# Table of Contents

What Is Amazon SageMaker? .......................................................... 1  
Amazon SageMaker Features ....................................................... 1  
Amazon SageMaker Pricing ....................................................... 3  
Are You a First-time User of Amazon SageMaker? .................. 3  
  How It Works ............................................................................. 4  
Machine Learning with Amazon SageMaker ......................... 4  
Explore, Analyze, and Process Data ........................................... 5  
Fairness and Model Explainability ............................................. 6  
  Best Practices for Evaluating Fairness and Explainability in the ML Lifecycle ............................................. 7  
Sample Notebooks .................................................................... 7  
Guide to the SageMaker Clarify Documentation .................... 8  
Model Training .......................................................................... 8  
Model Deployment ................................................................. 11  
Hosting Services ........................................................................ 11  
Batch Transform ....................................................................... 14  
Validating Models ................................................................... 15  
Model Monitoring ..................................................................... 16  
ML Frameworks, Python & R .................................................... 17  
  Apache MXNet ........................................................................ 17  
  Apache Spark ......................................................................... 18  
  Chainer ................................................................................... 26  
  PyTorch .................................................................................. 27  
  R ............................................................................................. 28  
  Scikit-learn .......................................................................... 30  
  SparkML Serving .................................................................... 31  
  TensorFlow .......................................................................... 31  
Supported Regions and Quotas .................................................. 32  
Get Started ................................................................................ 33  
  Set Up Amazon SageMaker .................................................. 33  
  Create an AWS Account ....................................................... 33  
  Create an IAM Administrator User and Group ................... 33  
Onboard to Studio ..................................................................... 34  
  Onboard Using Quick Start ................................................ 35  
  Onboard Using SSO ............................................................ 36  
  Onboard Using IAM ............................................................ 37  
  Choose a VPC ....................................................................... 38  
  Delete a Domain .................................................................... 39  
SageMaker JumpStart .............................................................. 41  
  Using JumpStart ................................................................. 43  
  Solutions ............................................................................... 44  
  Models .................................................................................... 44  
  Deploy a model ..................................................................... 44  
  Model Deployment Configuration .......................................... 44  
  Fine-Tune a Model .............................................................. 45  
  Fine-Tuning Data Source ..................................................... 45  
  Fine-Tuning deployment configuration .................................. 48  
  Hyperparameters ................................................................... 48  
  Training Output ...................................................................... 49  
  Next Steps ............................................................................. 49  
  Studio Tour ............................................................................ 50  
Get Started with Notebook Instances and SDKs .................... 58  
  Step 1: Create an Amazon S3 Bucket ................................... 59  
  Step 2: Create an Amazon SageMaker Notebook Instance .... 60  
  Step 3: Create a Jupyter Notebook ...................................... 60
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>SageMaker Studio</td>
<td>79</td>
</tr>
<tr>
<td>UI Overview</td>
<td>79</td>
</tr>
<tr>
<td>Set Up a Connection to an EMR Cluster</td>
<td>116</td>
</tr>
<tr>
<td>Perform Common Tasks</td>
<td>119</td>
</tr>
<tr>
<td>Notebook Instances</td>
<td>122</td>
</tr>
<tr>
<td>Notebook Pricing</td>
<td>94</td>
</tr>
<tr>
<td>Get Started</td>
<td>86</td>
</tr>
<tr>
<td>Compare Studio Notebooks with Notebook Instances</td>
<td>85</td>
</tr>
<tr>
<td>Create a Notebook Instance</td>
<td>124</td>
</tr>
<tr>
<td>Access Notebook Instances</td>
<td>126</td>
</tr>
<tr>
<td>Customize a Notebook Instance</td>
<td>127</td>
</tr>
<tr>
<td>Lifecycle Configuration Best Practices</td>
<td>128</td>
</tr>
<tr>
<td>Install External Libraries and Kernels in Notebook Instances</td>
<td>129</td>
</tr>
<tr>
<td>Notebook Instance Software Updates</td>
<td>132</td>
</tr>
<tr>
<td>Control an Amazon EMR Spark Instance Using a Notebook</td>
<td>132</td>
</tr>
<tr>
<td>Example Notebooks</td>
<td>134</td>
</tr>
<tr>
<td>Use or View Example Notebooks in Jupyter Classic</td>
<td>134</td>
</tr>
<tr>
<td>Use or View Example Notebooks in Jupyterlab</td>
<td>135</td>
</tr>
<tr>
<td>Set the Notebook Kernel</td>
<td>136</td>
</tr>
<tr>
<td>Git Repos</td>
<td>137</td>
</tr>
<tr>
<td>Add a Git Repository to Your Amazon SageMaker Account</td>
<td>138</td>
</tr>
<tr>
<td>Create a Notebook Instance with an Associated Git Repository</td>
<td>140</td>
</tr>
<tr>
<td>Associate a CodeCommit Repository in a Different AWS Account with a Notebook Instance</td>
<td>141</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>Get Started with Data Wrangler</td>
<td>527</td>
</tr>
<tr>
<td>Import</td>
<td>536</td>
</tr>
<tr>
<td>Create and Use a Data Wrangler Flow</td>
<td>543</td>
</tr>
<tr>
<td>Transform Data</td>
<td>545</td>
</tr>
<tr>
<td>Analyze and Visualize</td>
<td>562</td>
</tr>
<tr>
<td>Export</td>
<td>569</td>
</tr>
<tr>
<td>Security and Permissions</td>
<td>573</td>
</tr>
<tr>
<td>Troubleshoot</td>
<td>578</td>
</tr>
<tr>
<td>Process Data and Evaluate Models</td>
<td>580</td>
</tr>
<tr>
<td>Sample Notebooks</td>
<td>581</td>
</tr>
<tr>
<td>CloudWatch Logs and Metrics</td>
<td>582</td>
</tr>
<tr>
<td>Data Processing with Apache Spark</td>
<td>582</td>
</tr>
<tr>
<td>Data Processing with scikit-learn</td>
<td>583</td>
</tr>
<tr>
<td>Detect Pretraining Data Bias</td>
<td>583</td>
</tr>
<tr>
<td>Use Your Own Processing Code</td>
<td>599</td>
</tr>
<tr>
<td>Create, Store, and Share Features</td>
<td>605</td>
</tr>
<tr>
<td>How Feature Store Works</td>
<td>605</td>
</tr>
<tr>
<td>Create Feature Groups</td>
<td>606</td>
</tr>
<tr>
<td>Find, Discover, and Share Features</td>
<td>606</td>
</tr>
<tr>
<td>Real-Time Inference for Features Stored in the Online Store</td>
<td>606</td>
</tr>
<tr>
<td>Offline Store for Model Training and Batch Inference</td>
<td>607</td>
</tr>
<tr>
<td>Feature Data Ingestion</td>
<td>607</td>
</tr>
<tr>
<td>Access Control</td>
<td>607</td>
</tr>
<tr>
<td>Get started with Amazon SageMaker Feature Store</td>
<td>607</td>
</tr>
<tr>
<td>Feature Store Concepts</td>
<td>608</td>
</tr>
<tr>
<td>Create Feature Groups</td>
<td>608</td>
</tr>
<tr>
<td>Use Feature Store with Studio</td>
<td>614</td>
</tr>
<tr>
<td>Data Sources and Ingestion</td>
<td>621</td>
</tr>
<tr>
<td>Stream Ingestion</td>
<td>622</td>
</tr>
<tr>
<td>Data Wrangler with Feature Store</td>
<td>622</td>
</tr>
<tr>
<td>Athena and AWS Glue with Feature Store</td>
<td>624</td>
</tr>
<tr>
<td>Security and Access Control</td>
<td>625</td>
</tr>
<tr>
<td>Using KMS Permissions for Amazon SageMaker Feature Store</td>
<td>625</td>
</tr>
<tr>
<td>Authorizing Use of Your CMK for Your Online Store</td>
<td>626</td>
</tr>
<tr>
<td>Using Grants to Authorize Feature Store</td>
<td>627</td>
</tr>
<tr>
<td>Monitoring Feature Store interaction with AWS KMS</td>
<td>628</td>
</tr>
<tr>
<td>Accessing Data in Your Online Store</td>
<td>628</td>
</tr>
<tr>
<td>Authorizing Use of Your CMK for Offline Store</td>
<td>628</td>
</tr>
<tr>
<td>Amazon SageMaker Feature Store Offline Store Data Format</td>
<td>628</td>
</tr>
<tr>
<td>Amazon SageMaker Feature Store Notebook Examples</td>
<td>629</td>
</tr>
<tr>
<td>Training</td>
<td>630</td>
</tr>
<tr>
<td>Choose an algorithm</td>
<td>630</td>
</tr>
<tr>
<td>Choose an algorithm implementation</td>
<td>631</td>
</tr>
<tr>
<td>Problem types for the basic machine learning paradigisms</td>
<td>633</td>
</tr>
<tr>
<td>Use built-in algorithms</td>
<td>635</td>
</tr>
<tr>
<td>Use reinforcement learning</td>
<td>838</td>
</tr>
<tr>
<td>Experiments</td>
<td>845</td>
</tr>
<tr>
<td>SageMaker Experiments Features</td>
<td>846</td>
</tr>
<tr>
<td>Create an Experiment</td>
<td>847</td>
</tr>
<tr>
<td>View and Compare Experiments, Trials, and Trial Components</td>
<td>850</td>
</tr>
<tr>
<td>Track and Compare Tutorial</td>
<td>853</td>
</tr>
<tr>
<td>Search Experiments Using Studio</td>
<td>859</td>
</tr>
<tr>
<td>Clean Up Experiment Resources</td>
<td>864</td>
</tr>
<tr>
<td>Search Using the Console and API</td>
<td>866</td>
</tr>
<tr>
<td>Debugger</td>
<td>871</td>
</tr>
<tr>
<td>New Debugger Features</td>
<td>871</td>
</tr>
<tr>
<td>Feature</td>
<td>Page</td>
</tr>
<tr>
<td>------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>Model Parallel Training</td>
<td>1066</td>
</tr>
<tr>
<td>Data Parallel Training</td>
<td>1054</td>
</tr>
<tr>
<td>SageMaker Built-In Distributed Training Features</td>
<td>1054</td>
</tr>
<tr>
<td>Data Parallel Training</td>
<td>1054</td>
</tr>
<tr>
<td>Model Parallel Training</td>
<td>1054</td>
</tr>
<tr>
<td>Jupyter Notebook Examples</td>
<td>1083</td>
</tr>
<tr>
<td>Troubleshooting</td>
<td>1084</td>
</tr>
<tr>
<td>Detect Posttraining Data and Model Bias</td>
<td>1086</td>
</tr>
<tr>
<td>Sample Notebooks</td>
<td>1086</td>
</tr>
<tr>
<td>Measure Posttraining Data and Model Bias</td>
<td>1087</td>
</tr>
<tr>
<td>Configure SageMaker Clarify processing jobs</td>
<td>1105</td>
</tr>
<tr>
<td>Run SageMaker Clarify Processing Jobs</td>
<td>1112</td>
</tr>
<tr>
<td>Model Explainability</td>
<td>1115</td>
</tr>
<tr>
<td>Shapley Values</td>
<td>1116</td>
</tr>
<tr>
<td>SHAP Baselines for Explainability</td>
<td>1117</td>
</tr>
<tr>
<td>Create Feature Attribute Baselines and Explainability Reports</td>
<td>1117</td>
</tr>
<tr>
<td>Incremental Training</td>
<td>1118</td>
</tr>
<tr>
<td>Perform Incremental Training (Console)</td>
<td>1118</td>
</tr>
<tr>
<td>Perform Incremental Training (API)</td>
<td>1120</td>
</tr>
<tr>
<td>Managed Spot Training</td>
<td>1122</td>
</tr>
<tr>
<td>Using Managed Spot Training</td>
<td>1123</td>
</tr>
<tr>
<td>Managed Spot Training Lifecycle</td>
<td>1123</td>
</tr>
<tr>
<td>Use Checkpoints</td>
<td>1123</td>
</tr>
<tr>
<td>Use Augmented Manifest Files</td>
<td>1124</td>
</tr>
<tr>
<td>Augmented Manifest File format</td>
<td>1124</td>
</tr>
<tr>
<td>Stream Augmented Manifest File Data</td>
<td>1124</td>
</tr>
<tr>
<td>Use an Augmented Manifest File (Console)</td>
<td>1125</td>
</tr>
<tr>
<td>Use an Augmented Manifest File (API)</td>
<td>1127</td>
</tr>
<tr>
<td>Monitor and Analyze Using Metrics</td>
<td>1127</td>
</tr>
</tbody>
</table>

Amazon SageMaker Developer Guide
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inference</td>
<td>1135</td>
</tr>
<tr>
<td>Prerequisites</td>
<td>1135</td>
</tr>
<tr>
<td>What do you want to do?</td>
<td>1135</td>
</tr>
<tr>
<td>Manage Model Deployments</td>
<td>1136</td>
</tr>
<tr>
<td>Deploy Your Own Inference Code</td>
<td>1136</td>
</tr>
<tr>
<td>Guide to SageMaker</td>
<td>1136</td>
</tr>
<tr>
<td>Register and Deploy Models with Model Registry</td>
<td>1136</td>
</tr>
<tr>
<td>Create a Model Group</td>
<td>1137</td>
</tr>
<tr>
<td>Register a Model Version</td>
<td>1140</td>
</tr>
<tr>
<td>View Model Groups and Versions</td>
<td>1141</td>
</tr>
<tr>
<td>View Model Version Details</td>
<td>1143</td>
</tr>
<tr>
<td>Update Model Approval Status</td>
<td>1148</td>
</tr>
<tr>
<td>Deploy a Model in the Registry</td>
<td>1153</td>
</tr>
<tr>
<td>Deploy a Model Version from a Different Account</td>
<td>1154</td>
</tr>
<tr>
<td>View the Deployment History of a Model</td>
<td>1156</td>
</tr>
<tr>
<td>Multi-Model Endpoints</td>
<td>1160</td>
</tr>
<tr>
<td>Supported Algorithms and Frameworks</td>
<td>1161</td>
</tr>
<tr>
<td>Sample Notebooks</td>
<td>1161</td>
</tr>
<tr>
<td>Instance Recommendations for Multi-Model Endpoint Deployments</td>
<td>1162</td>
</tr>
<tr>
<td>Create a Multi-Model Endpoint</td>
<td>1163</td>
</tr>
<tr>
<td>Invoke a Multi-Model Endpoint</td>
<td>1166</td>
</tr>
<tr>
<td>Add or Remove Models</td>
<td>1167</td>
</tr>
<tr>
<td>Bring Your Own Container</td>
<td>1167</td>
</tr>
<tr>
<td>How Amazon SageMaker multi-model endpoints work</td>
<td>1172</td>
</tr>
<tr>
<td>Security</td>
<td>1173</td>
</tr>
<tr>
<td>CloudWatch Metrics for Multi-Model Endpoint Deployments</td>
<td>1173</td>
</tr>
<tr>
<td>Model Monitor</td>
<td>1174</td>
</tr>
<tr>
<td>How It Works</td>
<td>1174</td>
</tr>
<tr>
<td>Monitor Data Quality</td>
<td>1176</td>
</tr>
<tr>
<td>Monitor Model Quality</td>
<td>1181</td>
</tr>
<tr>
<td>Monitor Bias Drift</td>
<td>1187</td>
</tr>
<tr>
<td>Monitor Feature Attribution Drift</td>
<td>1191</td>
</tr>
<tr>
<td>Capture Data</td>
<td>1195</td>
</tr>
<tr>
<td>Schedule Monitoring Jobs</td>
<td>1196</td>
</tr>
<tr>
<td>Prebuilt Container</td>
<td>1198</td>
</tr>
<tr>
<td>Interpret Results</td>
<td>1199</td>
</tr>
<tr>
<td>Visualize Results</td>
<td>1201</td>
</tr>
<tr>
<td>Advanced Topics</td>
<td>1207</td>
</tr>
<tr>
<td>Inference Pipelines</td>
<td>1220</td>
</tr>
<tr>
<td>Sample Notebooks</td>
<td>1221</td>
</tr>
<tr>
<td>Process Features with Spark ML and Scikit-learn</td>
<td>1221</td>
</tr>
<tr>
<td>Create a Pipeline Model</td>
<td>1222</td>
</tr>
<tr>
<td>Real-time Inference</td>
<td>1224</td>
</tr>
<tr>
<td>Batch Transform</td>
<td>1226</td>
</tr>
<tr>
<td>Logs and Metrics</td>
<td>1227</td>
</tr>
<tr>
<td>Troubleshooting</td>
<td>1232</td>
</tr>
<tr>
<td>Compile and Deploy Models with Neo</td>
<td>1234</td>
</tr>
<tr>
<td>What is SageMaker Neo?</td>
<td>1234</td>
</tr>
<tr>
<td>How it Works</td>
<td>1234</td>
</tr>
<tr>
<td>Sample Notebooks</td>
<td>1235</td>
</tr>
<tr>
<td>Compile Models</td>
<td>1236</td>
</tr>
<tr>
<td>Cloud Instances</td>
<td>1242</td>
</tr>
</tbody>
</table>
What Is Amazon SageMaker?

Amazon SageMaker is a fully managed machine learning service. With SageMaker, data scientists and developers can quickly and easily build and train machine learning models, and then directly deploy them into a production-ready hosted environment. It provides an integrated Jupyter authoring notebook instance for easy access to your data sources for exploration and analysis, so you don't have to manage servers. It also provides common machine learning algorithms that are optimized to run efficiently against extremely large data in a distributed environment. With native support for bring-your-own-algorithms and frameworks, SageMaker offers flexible distributed training options that adjust to your specific workflows. Deploy a model into a secure and scalable environment by launching it with a few clicks from SageMaker Studio or the SageMaker console. Training and hosting are billed by minutes of usage, with no minimum fees and no upfront commitments.

This guide includes information and tutorials on SageMaker features. For additional information, see Amazon SageMaker developer resources.

Topics

- Amazon SageMaker Features (p. 1)
- Amazon SageMaker Pricing (p. 3)
- Are You a First-time User of Amazon SageMaker? (p. 3)

Amazon SageMaker Features

Amazon SageMaker includes the following features:

SageMaker Studio (p. 79)

An integrated machine learning environment where you can build, train, deploy, and analyze your models all in the same application.

SageMaker Model Registry (p. 1136)

Versioning, artifact and lineage tracking, approval workflow, and cross account support for deployment of your machine learning models.

SageMaker Projects (p. 1439)

Create end-to-end ML solutions with CI/CD by using SageMaker projects.

SageMaker Model Building Pipelines (p. 1397)

Create and manage machine learning pipelines integrated directly with SageMaker jobs.

SageMaker ML Lineage Tracking (p. 1463)

Track the lineage of machine learning workflows.

SageMaker Data Wrangler (p. 526)

Import, analyze, prepare, and featurize data in SageMaker Studio. You can integrate Data Wrangler into your machine learning workflows to simplify and streamline data pre-processing and feature
engineering using little to no coding. You can also add your own Python scripts and transformations to customize your data prep workflow.

SageMaker Feature Store (p. 605)

A centralized store for features and associated metadata so features can be easily discovered and reused. You can create two types of stores, an Online or Offline store. The Online Store can be used for low latency, real-time inference use cases and the Offline Store can be used for training and batch inference.

SageMaker JumpStart (p. 41)

Learn about SageMaker features and capabilities through curated 1-click solutions, example notebooks, and pretrained models that you can deploy. You can also fine-tune the models and deploy them.

SageMaker Clarify (p. 6)

Improve your machine learning models by detecting potential bias and help explain the predictions that models make.

SageMaker Edge Manager (p. 1280)

Optimize custom models for edge devices, create and manage fleets, run models with an efficient runtime, and monitor each model on each device.

SageMaker Ground Truth (p. 161)

High-quality training datasets by using workers along with machine learning to create labeled datasets.

Amazon Augmented AI (p. 1515)

Build the workflows required for human review of ML predictions. Amazon A2I brings human review to all developers, removing the undifferentiated heavy lifting associated with building human review systems or managing large numbers of human reviewers.

SageMaker Studio Notebooks (p. 84)

The next generation of SageMaker notebooks that include AWS Single Sign-On (AWS SSO) integration, fast start-up times, and single-click sharing.

SageMaker Experiments (p. 845)

Experiment management and tracking. You can use the tracked data to reconstruct an experiment, incrementally build on experiments conducted by peers, and trace model lineage for compliance and audit verifications.

SageMaker Debugger (p. 871)

Inspect training parameters and data throughout the training process. Automatically detect and alert users to commonly occurring errors such as parameter values getting too large or small.

SageMaker Autopilot (p. 146)

Users without machine learning knowledge can quickly build classification and regression models.

SageMaker Model Monitor (p. 1174)

Monitor and analyze models in production (endpoints) to detect data drift and deviations in model quality.

SageMaker Neo (p. 1234)

Train machine learning models once, then run anywhere in the cloud and at the edge.
SageMaker Elastic Inference (p. 1312)

Speed up the throughput and decrease the latency of getting real-time inferences.

Reinforcement Learning (p. 838)

Maximize the long-term reward that an agent receives as a result of its actions.

Preprocessing (p. 580)

Analyze and preprocess data, tackle feature engineering, and evaluate models.

Batch Transform (p. 1323)

Preprocess datasets, run inference when you don’t need a persistent endpoint, and associate input records with inferences to assist the interpretation of results.

Amazon SageMaker Pricing

As with other AWS products, there are no contracts or minimum commitments for using Amazon SageMaker. For more information about the cost of using SageMaker, see SageMaker Pricing.

Are You a First-time User of Amazon SageMaker?

If you are a first-time user of SageMaker, we recommend that you do the following:

1. Read How Amazon SageMaker Works (p. 4) – This section provides an overview of SageMaker, explains key concepts, and describes the core components involved in building AI solutions with SageMaker. We recommend that you read this topic in the order presented.

2. Set Up Amazon SageMaker (p. 33) – This section explains how to set up your AWS account and onboard to SageMaker Studio.

3. Amazon SageMaker Autopilot simplifies the machine learning experience by automating machine learning tasks. If you are new to SageMaker, it provides the easiest learning path. It also serves as an excellent ML learning tool that provides visibility into the code with notebooks generated for each of the automated ML tasks. For an introduction to its capabilities, see Automate model development with Amazon SageMaker Autopilot (p. 146). To get started building, training, and deploying machine learning models, Autopilot provides:
   - Samples: Explore modeling with Amazon SageMaker Autopilot (p. 147)
   - Videos: Use Autopilot to automate and explore the machine learning process (p. 147)
   - Tutorials: Get started with Amazon SageMaker Autopilot (p. 148)

4. Get Started with Amazon SageMaker (p. 33) – This section walks you through training your first model using SageMaker Studio, or the SageMaker console and the SageMaker API. You use training algorithms provided by SageMaker.

5. Explore other topics – Depending on your needs, do the following:
   - Submit Python code to train with deep learning frameworks – In SageMaker, you can use your own training scripts to train models. For information, see Use Machine Learning Frameworks, Python, and R with Amazon SageMaker (p. 17).
   - Use SageMaker directly from Apache Spark – For information, see Use Apache Spark with Amazon SageMaker (p. 18).
   - Use SageMaker to train and deploy your own custom algorithms – Package your custom algorithms with Docker so you can train and/or deploy them in SageMaker. To learn how SageMaker interacts with Docker containers, and for the SageMaker requirements for Docker images, see Using Docker containers with SageMaker (p. 1356).

6. View the API Reference – This section describes the SageMaker API operations.
How Amazon SageMaker Works

SageMaker is a fully managed service that enables you to quickly and easily integrate machine learning-based models into your applications. This section provides an overview of machine learning and explains how SageMaker works. If you are a first-time user of SageMaker, we recommend that you read the following sections in order:

1. Machine Learning with Amazon SageMaker (p. 4)
2. Explore, Analyze, and Process Data (p. 5)
3. Train a Model with Amazon SageMaker (p. 8)
4. Deploy a Model in Amazon SageMaker (p. 11)
5. Use Machine Learning Frameworks, Python, and R with Amazon SageMaker (p. 17)
6. Get Started with Amazon SageMaker (p. 33)

Machine Learning with Amazon SageMaker

This section describes a typical machine learning workflow and summarizes how you accomplish those tasks with Amazon SageMaker.

In machine learning, you "teach" a computer to make predictions, or inferences. First, you use an algorithm and example data to train a model. Then you integrate your model into your application to generate inferences in real time and at scale. In a production environment, a model typically learns from millions of example data items and produces inferences in hundreds to less than 20 milliseconds.

The following diagram illustrates the typical workflow for creating a machine learning model:

As the diagram illustrates, you typically perform the following activities:

1. **Generate example data**—To train a model, you need example data. The type of data that you need depends on the business problem that you want the model to solve (the inferences that you want the model to generate). For example, suppose that you want to create a model to predict a number given an input image of a handwritten digit. To train such a model, you need example images of handwritten numbers.
Data scientists often spend a lot of time exploring and preprocessing, or "wrangling," example data before using it for model training. To preprocess data, you typically do the following:

a. **Fetch the data**—You might have in-house example data repositories, or you might use datasets that are publicly available. Typically, you pull the dataset or datasets into a single repository.

b. **Clean the data**—To improve model training, inspect the data and clean it as needed. For example, if your data has a country name attribute with values United States and US, you might want to edit the data to be consistent.

c. **Prepare or transform the data**—To improve performance, you might perform additional data transformations. For example, you might choose to combine attributes. If your model predicts the conditions that require de-icing an aircraft, instead of using temperature and humidity attributes separately, you might combine those attributes into a new attribute to get a better model.

In SageMaker, you preprocess example data in a Jupyter notebook on your notebook instance. You use your notebook to fetch your dataset, explore it, and prepare it for model training. For more information, see **Explore, Analyze, and Process Data** (p. 5). For more information about preparing data in AWS Marketplace, see data preparation.

2. **Train a model**—Model training includes both training and evaluating the model, as follows:

   • **Training the model**—To train a model, you need an algorithm. The algorithm you choose depends on a number of factors. For a quick, out-of-the-box solution, you might be able to use one of the algorithms that SageMaker provides. For a list of algorithms provided by SageMaker and related considerations, see **Use Amazon SageMaker Built-in Algorithms** (p. 635).

   You also need compute resources for training. Depending on the size of your training dataset and how quickly you need the results, you can use resources ranging from a single general-purpose instance to a distributed cluster of GPU instances. For more information, see **Train a Model with Amazon SageMaker** (p. 8).

   • **Evaluating the model**—After you've trained your model, you evaluate it to determine whether the accuracy of the inferences is acceptable. In SageMaker, you use either the AWS SDK for Python (Boto) or the high-level Python library that SageMaker provides to send requests to the model for inferences.

     You use a Jupyter notebook in your SageMaker notebook instance to train and evaluate your model.

3. **Deploy the model**—You traditionally re-engineer a model before you integrate it with your application and deploy it. With SageMaker hosting services, you can deploy your model independently, decoupling it from your application code. For more information, see **Deploy a Model on SageMaker Hosting Services** (p. 11).

Machine learning is a continuous cycle. After deploying a model, you monitor the inferences, collect "ground truth," and evaluate the model to identify drift. You then increase the accuracy of your inferences by updating your training data to include the newly collected ground truth. You do this by retraining the model with the new dataset. As more and more example data becomes available, you continue retraining your model to increase accuracy.

---

### Explore, Analyze, and Process Data

Before using a dataset to train a model, data scientists typically explore, analyze, and preprocess it. For example, in one of the exercises in this guide, you use the MNIST dataset, a commonly available dataset.
of handwritten numbers, for model training. Before you begin training, you transform the data into a format that is more efficient for training. For more information, see Step 4.3: Transform the Training Dataset and Upload It to Amazon S3 (p. 63).

To preprocess data use one of the following methods:

- Use a Jupyter notebook on a Amazon SageMaker notebook instance. You can also use the notebook instance to do the following:
  - Write code to create model training jobs
  - Deploy models to SageMaker hosting
  - Test or validate your models

  For more information, see Use Amazon SageMaker Notebook Instances (p. 124)

- You can use a model to transform data by using SageMaker batch transform. For more information, see Step 6.2: Deploy the Model with Batch Transform (p. 70).

Amazon SageMaker Processing enables running jobs to preprocess and postprocess data, perform feature engineering, and evaluate models on SageMaker easily and at scale. When combined with the other critical machine learning tasks provided by SageMaker, such as training and hosting, Processing provides you with the benefits of a fully managed machine learning environment, including all the security and compliance support built into SageMaker. With Processing, you have the flexibility to use the built-in data processing containers or to bring your own containers and submit custom jobs to run on managed infrastructure. After you submit a job, SageMaker launches the compute instances, processes and analyzes the input data, and releases the resources upon completion. For more information, see Process Data and Evaluate Models (p. 580).

- For information about how to run your own data processing scripts, Data Processing with scikit-learn (p. 583).

- For information about how to build your own processing container to run scripts, see Build Your Own Processing Container (Advanced Scenario) (p. 600).

## What Is Fairness and Model Explainability for Machine Learning Predictions?

Amazon SageMaker Clarify helps improve your machine learning (ML) models by detecting potential bias and helping explain the predictions that models make. It helps you identify various types of bias in pretraining data and in posttraining that can emerge during model training or when the model is in production. SageMaker Clarify helps explain how these models make predictions using a feature attribution approach. It also monitors inferences models make in production for bias or feature attribution drift. The fairness and explainability functionality provided by SageMaker Clarify provides components that help AWS customers build less biased and more understandable machine learning models. It also provides tools to help you generate model governance reports that you can use to inform risk and compliance teams, and external regulators.

Machine learning models and data-driven systems are being increasingly used to help make decisions across domains such as financial services, healthcare, education, and human resources. Machine learning applications provide benefits such as improved accuracy, increased productivity, and cost savings to help meet regulatory requirements, improve business decisions, and provide better insights into data science procedures.

- **Regulatory** – In many situations, it is important to understand why an ML model made a specific prediction and also whether the prediction it made was impacted by any bias, either during training
or at inference. Recently, policymakers, regulators, and advocates have raised awareness about the ethical and policy challenges posed by ML and data-driven systems. In particular, they have expressed concerns about the potentially discriminatory impact of such systems (for example, inadvertently encoding of bias into automated decisions).

- **Business** – The adoption of AI systems in regulated domains requires trust, which can be built by providing reliable explanations of the behavior of trained models and how the deployed models make predictions. Model explainability may be particularly important to certain industries with reliability, safety, and compliance requirements, such as financial services, human resources, healthcare, and automated transportation. To take a common financial example, lending applications that incorporate the use of ML models might need to provide explanations about how those models made certain predictions to internal teams of loan officers, customer service representatives, and forecasters, in addition to end users/customers.

- **Data Science** – Data scientists and ML engineers need tools to generate the insights required to debug and improve ML models through better feature engineering, to determine whether a model is making inferences based on noisy or irrelevant features, and to understand the limitations of their models and failure modes their models may encounter.

**Best Practices for Evaluating Fairness and Explainability in the ML Lifecycle**

**Fairness as a Process** – The notions of bias and fairness are highly dependent on the application and the choice of attributes for which bias is to be measured, in addition to the choice of bias metrics. These choices should be guided by ethical, business, and regulatory considerations. Building consensus and achieving collaboration across key stakeholders (such as product, policy, legal, public relations (PR), engineering, AI/ML teams, end users, and communities) is important for the successful adoption of fair and transparent ML applications.

**Fairness and Explainability by Design in the ML Lifecycle** – You should consider fairness and explainability during each stage of the ML lifecycle: problem formation, dataset construction, algorithm selection, model training process, testing process, deployment, and monitoring/feedback. It is important to have the right tools to do this analysis. To encourage engaging with these considerations, here are a few example questions we recommend you ask during each of these stages.

**Sample Notebooks**

Amazon SageMaker Clarify provides the following sample notebooks:
• **Explainability and bias detection with Amazon SageMaker Clarify** – Use SageMaker Clarify to create a processing job for the detecting bias and explaining model predictions with feature attributions.

• **Monitoring bias drift and feature attribution drift Amazon SageMaker Clarify** – Use Amazon SageMaker Model Monitor to monitor bias drift and feature attribution drift over time.

These notebooks have been verified to run in Amazon SageMaker Studio only. If you need instructions on how to open a notebook in Amazon SageMaker Studio, see Create or Open an Amazon SageMaker Studio Notebook (p. 87). If you’re prompted to choose a kernel, choose **Python 3 (Data Science)**.

**Guide to the SageMaker Clarify Documentation**

Bias can occur and be measured in the data at each stage of the machine learning lifecycle: before training a model and after model training. SageMaker Clarify can provide feature attribution explanations of model predictions for trained models and for models deployed to production, where models can be monitored for any drift from their baseline explanatory attributions. The documentation for SageMaker Clarify is embedding throughout the larger SageMaker documentation set at the relevant ML stages as follows:

• For further information on detecting bias in preprocessing data before it’s used to train a model, see Detect Pretraining Data Bias (p. 583).

• For further information on detecting posttraining data and model bias, see Detect Posttraining Data and Model Bias (p. 1086).

• For further information on the model-agnostic feature attribution approach to explain model predictions after training, see Model Explainability (p. 1115).

• For further information on monitoring for bias in production model inferences due to the drift of data away from the baseline used to train the model, see Monitor Bias Drift for Models in Production (p. 1187).

• For further information on monitoring for the drift of features’ contributions away from the baseline that was established during model training, see Monitor Feature Attribution Drift for Models in Production (p. 1191).

**Train a Model with Amazon SageMaker**

The following diagram shows how you train and deploy a model with Amazon SageMaker:
The area labeled SageMaker highlights the two components of SageMaker: model training and model deployment.

To train a model in SageMaker, you create a training job. The training job includes the following information:

- The URL of the Amazon Simple Storage Service (Amazon S3) bucket where you've stored the training data.
- The compute resources that you want SageMaker to use for model training. Compute resources are ML compute instances that are managed by SageMaker.
- The URL of the S3 bucket where you want to store the output of the job.
- The Amazon Elastic Container Registry path where the training code is stored. For more information, see Docker Registry Paths for SageMaker Built-in Algorithms (p. 640).

You have the following options for a training algorithm:

- **Use an algorithm provided by SageMaker**—SageMaker provides training algorithms. If one of these meets your needs, it's a great out-of-the-box solution for quick model training. For a list of algorithms...
provided by SageMaker, see Use Amazon SageMaker Built-in Algorithms (p. 635). To try an exercise that uses an algorithm provided by SageMaker, see Get Started with Amazon SageMaker (p. 33).

- **Use SageMaker Debugger**—to inspect training parameters and data throughout the training process when working with the TensorFlow, PyTorch, and Apache MXNet learning frameworks or the XGBoost algorithm. Debugger automatically detects and alerts users to commonly occurring errors such as parameter values getting too large or small. For more information about using Debugger, see Amazon SageMaker Debugger (p. 871). Debugger sample notebooks are available at Amazon SageMaker Debugger Samples.

- **Use Apache Spark with SageMaker**—SageMaker provides a library that you can use in Apache Spark to train models with SageMaker. Using the library provided by SageMaker is similar to using Apache Spark MLLib. For more information, see Use Apache Spark with Amazon SageMaker (p. 18).

- **Submit custom code to train with deep learning frameworks**—You can submit custom Python code that uses TensorFlow, PyTorch, or Apache MXNet for model training. For more information, see Use TensorFlow with Amazon SageMaker (p. 31), Use PyTorch with Amazon SageMaker (p. 27), and Use Apache MXNet with Amazon SageMaker (p. 17).

- **Use your own custom algorithms**—Put your code together as a Docker image and specify the registry path of the image in a SageMaker CreateTrainingJob API call. For more information, see Using Docker containers with SageMaker (p. 1356).

- **Use an algorithm that you subscribe to from AWS Marketplace**—For information, see Find and Subscribe to Algorithms and Model Packages on AWS Marketplace (p. 1610).

After you create the training job, SageMaker launches the ML compute instances and uses the training code and the training dataset to train the model. It saves the resulting model artifacts and other output in the S3 bucket you specified for that purpose.

You can create a training job with the SageMaker console or the API. For information about creating a training job with the API, see the CreateTrainingJob API.

When you create a training job with the API, SageMaker replicates the entire dataset on ML compute instances by default. To make SageMaker replicate a subset of the data on each ML compute instance, you must set the S3DataDistributionType field to ShardedByS3Key. You can set this field using the low-level SDK. For more information, see S3DataDistributionType in S3DataSource.

**Important**
To prevent your algorithm container from contending for memory, we reserve memory for our SageMaker critical system processes on your ML compute instances and therefore you cannot expect to see all the memory for your instance type.
Deploy a Model in Amazon SageMaker

After you train your model, you can deploy it using Amazon SageMaker to get predictions in any of the following ways:

• To set up a persistent endpoint to get one prediction at a time, use SageMaker hosting services.
• To get predictions for an entire dataset, use SageMaker batch transform.

Topics
• Deploy a Model on SageMaker Hosting Services (p. 11)

Deploy a Model on SageMaker Hosting Services

For an example of how to deploy a model to the SageMaker hosting service, see Step 6.1: Deploy the Model to SageMaker Hosting Services (p. 68).

Or, if you prefer, watch the following video tutorial:

Deploy Your ML Models to Production at Scale with Amazon SageMaker

SageMaker provides model hosting services for model deployment, as shown in the following diagram. SageMaker provides an HTTPS endpoint where your machine learning model is available to provide inferences.
Deploying a model using SageMaker hosting services is a three-step process:

1. **Create a model in SageMaker**—By creating a model, you tell SageMaker where it can find the model components. This includes the S3 path where the model artifacts are stored and the Docker registry path for the image that contains the inference code. In subsequent deployment steps, you specify the model by name. For more information, see the [CreateModel API](#).

   **Note**
   The S3 bucket where the model artifacts are stored must be in the same region as the model that you are creating.

2. **Create an endpoint configuration for an HTTPS endpoint**—You specify the name of one or more models in production variants and the ML compute instances that you want SageMaker to launch to host each production variant.

   **Note**
   SageMaker supports running (a) multiple models, (b) multiple variants of a model, or (c) combinations of models and variants on the same endpoint. Model variants can reference the same model inference container (i.e. run the same algorithm), but use different
model artifacts (e.g., different model weight values based on other hyper-parameter configurations). In contrast, two different models may use the same algorithm, but focus on different business problems or underlying goals and may operate on different data sets.

When hosting models in production, you can configure the endpoint to elastically scale the deployed ML compute instances. For each production variant, you specify the number of ML compute instances that you want to deploy. When you specify two or more instances, SageMaker launches them in multiple Availability Zones. This ensures continuous availability. SageMaker manages deploying the instances. For more information, see the CreateEndpointConfig API.

3. Create an HTTPS endpoint—Provide the endpoint configuration to SageMaker. The service launches the ML compute instances and deploys the model or models as specified in the configuration. For more information, see the CreateEndpoint API. To get inferences from the model, client applications send requests to the SageMaker Runtime HTTPS endpoint. For more information about the API, see the InvokeEndpoint API.

Note
Endpoints are scoped to an individual AWS account, and are not public. The URL does not contain the account ID, but SageMaker determines the account ID from the authentication token that is supplied by the caller. For an example of how to use Amazon API Gateway and AWS Lambda to set up and deploy a web service that you can call from a client application that is not within the scope of your account, see Call a SageMaker model endpoint using Amazon API Gateway and AWS Lambda in the AWS Machine Learning Blog.

Note
When you create an endpoint, SageMaker attaches an Amazon EBS storage volume to each ML compute instance that hosts the endpoint. The size of the storage volume depends on the instance type. For a list of instance types that SageMaker hosting service supports, see AWS Service Limits. For a list of the sizes of the storage volumes that SageMaker attaches to each instance, see Host Instance Storage Volumes (p. 1355).

To increase a model's accuracy, you might choose to save the user's input data and ground truth, if available, as part of the training data. You can then retrain the model periodically with a larger, improved training dataset.

Best Practices for Deploying Models on SageMaker Hosting Services

When hosting models using SageMaker hosting services, consider the following:

- Typically, a client application sends requests to the SageMaker HTTPS endpoint to obtain inferences from a deployed model. You can also send requests to this endpoint from your Jupyter notebook during testing.

- You can deploy a model trained with SageMaker to your own deployment target. To do that, you need to know the algorithm-specific format of the model artifacts that were generated by model training. For more information about output formats, see the section corresponding to the algorithm you are using in Common Data Formats for Training (p. 646).

- You can deploy multiple variants of a model to the same SageMaker HTTPS endpoint. This is useful for testing variations of a model in production. For example, suppose that you've deployed a model into production. You want to test a variation of the model by directing a small amount of traffic, say 5%, to the new model. To do this, create an endpoint configuration that describes both variants of the
You can configure a ProductionVariant to use Application Auto Scaling. For information about configuring automatic scaling, see [Automatically Scale Amazon SageMaker Models](https://docs.aws.amazon.com/sagemaker/latest/dg/automatically-scale-endpoints.html) (p. 1330).

You can modify an endpoint without taking models that are already deployed into production out of service. For example, you can add new model variants, update the ML Compute instance configurations of existing model variants, or change the distribution of traffic among model variants. To modify an endpoint, you provide a new endpoint configuration. SageMaker implements the changes without any downtime. For more information see, `UpdateEndpoint` and `UpdateEndpointWeightsAndCapacities`.

Changing or deleting model artifacts or changing inference code after deploying a model produces unpredictable results. If you need to change or delete model artifacts or change inference code, modify the endpoint by providing a new endpoint configuration. Once you provide the new endpoint configuration, you can change or delete the model artifacts corresponding to the old endpoint configuration.

If you want to get inferences on entire datasets, consider using batch transform as an alternative to hosting services. For information, see [Get Inferences for an Entire Dataset with Batch Transform](https://docs.aws.amazon.com/sagemaker/latest/dg/batch-transform.html) (p. 14)

---

### Get Inferences for an Entire Dataset with Batch Transform

To get inferences for an entire dataset, use batch transform. With batch transform, you create a batch transform job using a trained model and the dataset, which must be stored in Amazon S3. Amazon SageMaker saves the inferences in an S3 bucket that you specify when you create the batch transform job. Batch transform manages all of the compute resources required to get inferences. This includes launching instances and deleting them after the batch transform job has completed. Batch transform manages interactions between the data and the model with an object within the instance node called an agent.

Use batch transform when you:

- Want to get inferences for an entire dataset and index them to serve inferences in real time
- Don't need a persistent endpoint that applications (for example, web or mobile apps) can call to get inferences
- Don't need the subsecond latency that SageMaker hosted endpoints provide

You can also use batch transform to preprocess your data before using it to train a new model or generate inferences.

The following diagram shows the workflow of a batch transform job:
To perform a batch transform, create a batch transform job using either the SageMaker console or the API. Provide the following:

- The path to the S3 bucket where you've stored the data that you want to transform.
- The compute resources that you want SageMaker to use for the transform job. Compute resources are machine learning (ML) compute instances that are managed by SageMaker.
- The path to the S3 bucket where you want to store the output of the job.
- The name of the SageMaker model that you want to use to create inferences. You must use a model that you have already created either with the `CreateModel` operation or the console.

The following is an example of what a dataset file might look like.

```
An example of input file content:
Record1-Attribute1, Record1-Attribute2, Record1-Attribute3, ..., Record1-AttributeM
Record2-Attribute1, Record2-Attribute2, Record2-Attribute3, ..., Record2-AttributeM
Record3-Attribute1, Record3-Attribute2, Record3-Attribute3, ..., Record3-AttributeM
...  
RecordN-Attribute1, RecordN-Attribute2, RecordN-Attribute3, ..., RecordN-AttributeM
```

A record is a single input data unit. For information about how to delimit records for batch transform jobs, see `SplitType`.

For an example of how to use batch transform, see Step 6.2: Deploy the Model with Batch Transform (p. 70).

**Validate a Machine Learning Model**

After training a model, evaluate it to determine whether its performance and accuracy enable you to achieve your business goals. You might generate multiple models using different methods and evaluate each. For example, you could apply different business rules for each model, and then apply various measures to determine each model's suitability. You might consider whether your model needs to be more sensitive than specific (or vice versa).
You can evaluate your model using historical data (offline) or live data:

- **Offline testing**—Use historical, not live, data to send requests to the model for inferences.

  Deploy your trained model to an alpha endpoint, and use historical data to send inference requests to it. To send the requests, use a Jupyter notebook in your Amazon SageMaker notebook instance and either the AWS SDK for Python (Boto) or the high-level Python library provided by SageMaker.

- **Online testing with live data**—SageMaker supports A/B testing for models in production by using production variants. Production variants are models that use the same inference code and are deployed on the same SageMaker endpoint. You configure the production variants so that a small portion of the live traffic goes to the model that you want to validate. For example, you might choose to send 10% of the traffic to a model variant for evaluation. After you are satisfied with the model’s performance, you can route 100% traffic to the updated model. For an example of testing models in production, see Test models in production (p. 1346).

For more information, see articles and books about how to evaluate models, for example, Evaluating Machine Learning Models.

Options for offline model evaluation include:

- **Validating using a holdout set**—Machine learning practitioners often set aside a part of the data as a "holdout set." They don’t use this data for model training.

  With this approach, you evaluate how well your model provides inferences on the holdout set. You then assess how effectively the model generalizes what it learned in the initial training, as opposed to using model memory. This approach to validation gives you an idea of how often the model is able to infer the correct answer.

  In some ways, this approach is similar to teaching elementary school students. First, you provide them with a set of examples to learn, and then test their ability to generalize from their learning. With homework and tests, you pose problems that were not included in the initial learning and determine whether they are able to generalize effectively. Students with perfect memories could memorize the problems, instead of learning the rules.

  Typically, the holdout dataset is of 20-30% of the training data.

- **k-fold validation**—In this validation approach, you split the example dataset into $k$ parts. You treat each of these parts as a holdout set for $k$ training runs, and use the other $k-1$ parts as the training set for that run. You produce $k$ models using a similar process, and aggregate the models to generate your final model. The value $k$ is typically in the range of 5-10.

## Monitoring a Model in Production

After you deploy a model into your production environment, use Amazon SageMaker model monitor to continuously monitor the quality of your machine learning models in real time. Amazon SageMaker model monitor enables you to set up an automated alert triggering system when there are deviations in the model quality, such as data drift and anomalies. Amazon CloudWatch Logs collects log files of monitoring the model status and notifies when the quality of your model hits certain thresholds that you preset. AWS CloudTrail stores the log files to an Amazon S3 bucket you specify. Early and pro-active detection of model deviations through AWS model monitor products enables you to take prompt actions to maintain and improve the quality of your deployed model.
For more information about SageMaker model monitoring products, see Amazon SageMaker Model Monitor (p. 1174).

To start your machine learning journey with SageMaker, sign up for an AWS account at Set Up SageMaker.

Use Machine Learning Frameworks, Python, and R with Amazon SageMaker

You can use Python and R natively in Amazon SageMaker notebook kernels. There are also kernels that support specific frameworks. A very popular way to get started with SageMaker is to use the Amazon SageMaker Python SDK. It provides open source Python APIs and containers that make it easy to train and deploy models in SageMaker, as well as examples for use with several different machine learning and deep learning frameworks.

For information about using specific frameworks or how to use R in SageMaker, see the following topics.

Languages SDKs and user guides:

- Amazon SageMaker Python SDK
- R (p. 28)
- API Reference Guide for Amazon SageMaker (p. 1723)

Machine learning and deep learning frameworks guides:

- Apache MXNet (p. 17)
- Apache Spark (p. 18)
- Chainer (p. 26)
- PyTorch (p. 27)
- Scikit-learn (p. 30)
- SparkML Serving (p. 31)
- TensorFlow (p. 31)

Use Apache MXNet with Amazon SageMaker

You can use SageMaker to train and deploy a model using custom MXNet code. The Amazon SageMaker Python SDK MXNet estimators and models and the SageMaker open-source MXNet container make writing a MXNet script and running it in SageMaker easier.

What do you want to do?

I want to train a custom MXNet model in SageMaker.

For a sample Jupyter notebook, see the MXNet example notebooks in the Amazon SageMaker Examples GitHub repository.

For documentation, see Train a Model with MXNet.

I have an MXNet model that I trained in SageMaker, and I want to deploy it to a hosted endpoint.

For more information, see Deploy MXNet models.
I have an MXNet model that I trained outside of SageMaker, and I want to deploy it to a SageMaker endpoint. For more information, see Deploy Endpoints from Model Data. I want to see the API documentation for Amazon SageMaker Python SDK MXNet classes. For more information, see MXNet Classes. I want to find the SageMaker MXNet container repository. For more information, see SageMaker MXNet Container GitHub repository. I want to find information about MXNet versions supported by AWS Deep Learning Containers. For more information, see Available Deep Learning Container Images.

For general information about writing MXNet script mode training scripts and using MXNet script mode estimators and models with SageMaker, see Using MXNet with the SageMaker Python SDK.

**Use Apache Spark with Amazon SageMaker**

This section provides information for developers who want to use Apache Spark for preprocessing data and Amazon SageMaker for model training and hosting. For information about supported versions of Apache Spark, see the Getting SageMaker Spark page in the SageMaker Spark GitHub repository.

SageMaker provides an Apache Spark library, in both Python and Scala, that you can use to easily train models in SageMaker using org.apache.spark.sql.DataFrame data frames in your Spark clusters. After model training, you can also host the model using SageMaker hosting services.

The SageMaker Spark library, com.amazonaws.services.sagemaker.sparksdk, provides the following classes, among others:

- **SageMakerEstimator**—Extends the org.apache.spark.ml.Estimator interface. You can use this estimator for model training in SageMaker.
- **SageMakerModel**—Extends the org.apache.spark.ml.Model class. You can use this SageMakerModel for model hosting and obtaining inferences in SageMaker.

**Download the SageMaker Spark Library**

You have the following options for downloading the Spark library provided by SageMaker:

- You can download the source code for both PySpark and Scala libraries from the SageMaker Spark GitHub repository.
- For the Python Spark library, you have the following additional options:
  - Use pip install:

    ```bash
    # pip install sagemaker_pyspark
    ```
  - In a notebook instance, create a new notebook that uses either the Sparkmagic (PySpark) or the Sparkmagic (PySpark3) kernel and connect to a remote Amazon EMR cluster.

    **Note**
    The EMR cluster must be configured with an IAM role that has the AmazonSageMakerFullAccess policy attached. For information about configuring roles...
• You can get the Scala library from Maven. Add the Spark library to your project by adding the following dependency to your pom.xml file:

```xml
<dependency>
  <groupId>com.amazonaws</groupId>
  <artifactId>sagemaker-spark_2.11</artifactId>
  <version>spark_2.2.0-1.0</version>
</dependency>
```

## Integrate Your Apache Spark Application with SageMaker

The following is high-level summary of the steps for integrating your Apache Spark application with SageMaker.

1. Continue data preprocessing using the Apache Spark library that you are familiar with. Your dataset remains a DataFrame in your Spark cluster. Load your data into a DataFrame and preprocess it so that you have a features column with `org.apache.spark.ml.linalg.Vector` of `Doubles`, and an optional label column with values of `Double` type.

2. Use the estimator in the SageMaker Spark library to train your model. For example, if you choose the k-means algorithm provided by SageMaker for model training, you call the `KMeansSageMakerEstimator.fit` method.

   Provide your DataFrame as input. The estimator returns a `SageMakerModel` object.

   **Note**
   
   SageMakerModel extends the `org.apache.spark.ml.Model`.

   The `fit` method does the following:

   a. Converts the input DataFrame to the protobuf format by selecting the features and label columns from the input DataFrame and uploading the protobuf data to an Amazon S3 bucket. The protobuf format is efficient for model training in SageMaker.

   b. Starts model training in SageMaker by sending a SageMaker `CreateTrainingJob` request. After model training has completed, SageMaker saves the model artifacts to an S3 bucket. SageMaker assumes the IAM role that you specified for model training to perform tasks on your behalf. For example, it uses the role to read training data from an S3 bucket and to write model artifacts to a bucket.

   c. Creates and returns a `SageMakerModel` object. The constructor does the following tasks, which are related to deploying your model to SageMaker.

      i. Sends a `CreateModel` request to SageMaker.

      ii. Sends a `CreateEndpointConfig` request to SageMaker.

      iii. Sends a `CreateEndpoint` request to SageMaker, which then launches the specified resources, and hosts the model on them.

3. You can get inferences from your model hosted in SageMaker with the `SageMakerModel.transform`.

   Provide an input DataFrame with features as input. The `transform` method transforms it to a DataFrame containing inferences. Internally, the `transform` method sends a request to the `InvokeEndpoint` SageMaker API to get inferences. The `transform` method appends the inferences to the input DataFrame.
Example 1: Use Amazon SageMaker for Training and Inference with Apache Spark

Topics
- Use Custom Algorithms for Model Training and Hosting on Amazon SageMaker with Apache Spark (p. 24)
- Use the SageMakerEstimator in a Spark Pipeline (p. 25)

Amazon SageMaker provides an Apache Spark library (in both Python and Scala) that you can use to integrate your Apache Spark applications with SageMaker. For example, you might use Apache Spark for data preprocessing and SageMaker for model training and hosting. For more information, see Use Apache Spark with Amazon SageMaker (p. 18). This section provides example code that uses the Apache Spark Scala library provided by SageMaker to train a model in SageMaker using DataFrames in your Spark cluster. The example also hosts the resulting model artifacts using SageMaker hosting services. Specifically, this example does the following:

- Uses the KMeansSageMakerEstimator to fit (or train) a model on data

Because the example uses the k-means algorithm provided by SageMaker to train a model, you use the KMeansSageMakerEstimator. You train the model using images of handwritten single-digit numbers (from the MNIST dataset). You provide the images as an input DataFrame. For your convenience, SageMaker provides this dataset in an S3 bucket.

In response, the estimator returns a SageMakerModel object.

- Obtains inferences using the trained SageMakerModel

To get inferences from a model hosted in SageMaker, you call the SageMakerModel.transform method. You pass a DataFrame as input. The method transforms the input DataFrame to another DataFrame containing inferences obtained from the model.

For a given input image of a handwritten single-digit number, the inference identifies a cluster that the image belongs to. For more information, see K-Means Algorithm (p. 711).

This is the example code:

```java
import org.apache.spark.sql.SparkSession
import com.amazonaws.services.sagemaker.sparksdk.IAMRole
import com.amazonaws.services.sagemaker.sparksdk.algorithms.KMeansSageMakerEstimator

val spark = SparkSession.builder.getOrCreate

// load mnist data as a dataframe from libsvm
val region = "us-east-1"
val trainingData = spark.read.format("libsvm")
  .option("numFeatures", "784")
  .load(s"s3://sagemaker-sample-data-$region/spark/mnist/train/")
```
```scala
val testData = spark.read.format("libsvm")
 .option("numFeatures", "784")
 .load(s"s3://sagemaker-sample-data-$region/spark/mnist/test/")

val roleArn = "arn:aws:iam::account-id:role/rolename"

val estimator = new KMeansSageMakerEstimator(
  sagemakerRole = IAMRole(roleArn),
  trainingInstanceType = "ml.p2.xlarge",
  trainingInstanceCount = 1,
  endpointInstanceType = "ml.c4.xlarge",
  endpointInitialInstanceCount = 1)
 .setK(10).setFeatureDim(784)

val model = estimator.fit(trainingData)

val transformedData = model.transform(testData)
transformedData.show
```

The code does the following:

- Loads the MNIST dataset from an S3 bucket provided by SageMaker (awsai-sparksdk-dataset) into a Spark DataFrame (mnistTrainingDataFrame):

``` scala
// Get a Spark session.
val spark = SparkSession.builder.getOrCreate

// load mnist data as a dataframe from libsvm
val region = "us-east-1"
val trainingData = spark.read.format("libsvm")
 .option("numFeatures", "784")
 .load(s"s3://sagemaker-sample-data-$region/spark/mnist/train/")
val testData = spark.read.format("libsvm")
 .option("numFeatures", "784")
 .load(s"s3://sagemaker-sample-data-$region/spark/mnist/test/")

val roleArn = "arn:aws:iam::account-id:role/rolename"
trainingData.show()
```

The `show` method displays the first 20 rows in the data frame:

```
+-----+--------------------+
|label|            features|
+-----+--------------------+
|  5.0| (784,[152,153,154...|
|  0.0| (784,[127,128,129...|
|  4.0| (784,[160,161,162...|
|  1.0| (784,[158,159,160...|
|  9.0| (784,[208,209,210...|
|  2.0| (784,[155,156,157...|
|  1.0| (784,[124,125,126...|
|  3.0| (784,[151,152,153...|
|  1.0| (784,[152,153,154...|
|  4.0| (784,[134,135,161...|
|  3.0| (784,[123,124,125...|
|  5.0| (784,[216,217,218...|
|  3.0| (784,[143,144,145...|
|  6.0| (784,[72,73,74,99...|
|  1.0| (784,[151,152,153...|
|  7.0| (784,[211,212,213...|
|  2.0| (784,[151,152,153...|
```
In each row:

- The **label** column identifies the image's label. For example, if the image of the handwritten number is the digit 5, the label value is 5.

- The **features** column stores a vector (org.apache.spark.ml.linalg.Vector) of Double values. These are the 784 features of the handwritten number. (Each handwritten number is a 28 x 28-pixel image, making 784 features.)

- Creates a SageMaker estimator (**KMeansSageMakerEstimator**)

  The `fit` method of this estimator uses the k-means algorithm provided by SageMaker to train models using an input DataFrame. In response, it returns a `SageMakerModel` object that you can use to get inferences.

  **Note**
  
  The `KMeansSageMakerEstimator` extends the SageMaker `SageMakerEstimator`, which extends the Apache Spark `Estimator`.

```scala
val estimator = new KMeansSageMakerEstimator(
  sagemakerRole = IAMRole(roleArn),
  trainingInstanceType = "ml.p2.xlarge",
  trainingInstanceCount = 1,
  endpointInstanceType = "ml.c4.xlarge",
  endpointInitialInstanceCount = 1)
  .setK(10).setFeatureDim(784)
```

The constructor parameters provide information that is used for training a model and deploying it on SageMaker:

- `trainingInstanceType` and `trainingInstanceCount`—Identify the type and number of ML compute instances to use for model training.

- `endpointInstanceType`—Identifies the ML compute instance type to use when hosting the model in SageMaker. By default, one ML compute instance is assumed.

- `endpointInitialInstanceCount`—Identifies the number of ML compute instances initially backing the endpoint hosting the model in SageMaker.

- `sagemakerRole`—SageMaker assumes this IAM role to perform tasks on your behalf. For example, for model training, it reads data from S3 and writes training results (model artifacts) to S3.

  **Note**
  
  This example implicitly creates a SageMaker client. To create this client, you must provide your credentials. The API uses these credentials to authenticate requests to SageMaker. For example, it uses the credentials to authenticate requests to create a training job and API calls for deploying the model using SageMaker hosting services.

- After the `KMeansSageMakerEstimator` object has been created, you set the following parameters, are used in model training:
• The number of clusters that the k-means algorithm should create during model training. You specify 10 clusters, one for each digit, 0 through 9.

• Identifies that each input image has 784 features (each handwritten number is a 28 x 28-pixel image, making 784 features).

• Calls the estimator fit method

```scala
// train
val model = estimator.fit(trainingData)
```

You pass the input DataFrame as a parameter. The model does all the work of training the model and deploying it to SageMaker. For more information see, Integrate Your Apache Spark Application with SageMaker (p. 19). In response, you get a SageMakerModel object, which you can use to get inferences from your model deployed in SageMaker.

You provide only the input DataFrame. You don't need to specify the registry path to the k-means algorithm used for model training because the KMeansSageMakerEstimator knows it.

• Calls the SageMakerModel.transform method to get inferences from the model deployed in SageMaker.

The transform method takes a DataFrame as input, transforms it, and returns another DataFrame containing inferences obtained from the model.

```scala
val transformedData = model.transform(testData)
transformedData.show
```

For simplicity, we use the same DataFrame as input to the transform method that we used for model training in this example. The transform method does the following:

• Serializes the features column in the input DataFrame to protobuf and sends it to the SageMaker endpoint for inference.

• Deserializes the protobuf response into the two additional columns (distance_to_cluster and closest_cluster) in the transformed DataFrame.

The show method gets inferences to the first 20 rows in the input DataFrame:

```
+-----+--------------------+-------------------+---------------+
| label|            features|distance_to_cluster|closest_cluster|
+-----+--------------------+-------------------+---------------+
|  5.0|(784,[152,153,154...|  1767.897705078125|            4.0|
|  0.0|(784,[127,128,129...| 1392.157470703125|            5.0|
|  4.0|(784,[160,161,162...| 1671.5711669921875|            9.0|
|  1.0|(784,[158,159,160...| 1182.6082763671875|            6.0|
|  9.0|(784,[208,209,210...| 1390.4002685546875|            0.0|
|  2.0|(784,[155,156,157...| 1713.988037109375|            1.0|
|  1.0|(784,[124,125,126...| 1246.3016357421875|            2.0|
|  3.0|(784,[151,152,153...| 1753.229248046875|            4.0|
|  1.0|(784,[152,153,154...|  978.8394165039062|            2.0|
|  4.0|(784,[134,135,161...| 1623.176513671875|            3.0|
|  3.0|(784,[123,124,125...| 1533.863525390625|            4.0|
|  5.0|(784,[216,217,218...| 1469.357177734375|            6.0|
|  3.0|(784,[143,144,145...| 1736.765869140625|            4.0|
|  6.0|(784,[72,73,74,99...| 1473.69384765625|            8.0|
```
You can interpret the data, as follows:

- A handwritten number with the label 5 belongs to cluster 4 (closest_cluster).
- A handwritten number with the label 0 belongs to cluster 5.
- A handwritten number with the label 4 belongs to cluster 9.
- A handwritten number with the label 1 belongs to cluster 6.

For more information on how to run these examples, see https://github.com/aws/sagemaker-spark/blob/master/README.md on GitHub.

Use Custom Algorithms for Model Training and Hosting on Amazon SageMaker with Apache Spark

In Example 1: Use Amazon SageMaker for Training and Inference with Apache Spark (p. 20), you use the kMeansSageMakerEstimator because the example uses the k-means algorithm provided by Amazon SageMaker for model training. You might choose to use your own custom algorithm for model training instead. Assuming that you have already created a Docker image, you can create your own SageMakerEstimator and specify the Amazon Elastic Container Registry path for your custom image.

The following example shows how to create a KMeansSageMakerEstimator from the SageMakerEstimator. In the new estimator, you explicitly specify the Docker registry path to your training and inference code images.

```java
import com.amazonaws.services.sagemaker.sparksdk.IAMRole
import com.amazonaws.services.sagemaker.sparksdk.SageMakerEstimator
import com.amazonaws.services.sagemaker.sparksdk.transformation.serializers.ProtobufRequestRowSerializer
import com.amazonaws.services.sagemaker.sparksdk.transformation.deserializers.KMeansProtobufResponseRowDeserializer
val estimator = new SageMakerEstimator(
    trainingImage = "811284229777.dkr.ecr.us-east-1.amazonaws.com/kmeans:1",
    modelImage = "811284229777.dkr.ecr.us-east-1.amazonaws.com/kmeans:1",
    requestRowSerializer = new ProtobufRequestRowSerializer(),
    responseRowDeserializer = new KMeansProtobufResponseRowDeserializer(),
    hyperParameters = Map("k" -> "10", "feature_dim" -> "784"),
    sagemakerRole = IAMRole(roleArn),
    trainingInstanceType = "ml.p2.xlarge",
    trainingInstanceCount = 1,
    endpointInstanceType = "ml.c4.xlarge",
    endpointInitialInstanceCount = 1,
    trainingSparkDataFormat = "sagemaker")
```

In the code, the parameters in the SageMakerEstimator constructor include:

- `trainingImage` —Identifies the Docker registry path to the training image containing your custom code.
- `modelImage` —Identifies the Docker registry path to the image containing inference code.
• requestRowSerializer —Implements
com.amazonaws.services.sagemaker.sparksdk.transformation.RequestRowSerializer.

This parameter serializes rows in the input DataFrame to send them to the model hosted in
SageMaker for inference.

• responseRowDeserializer —Implements
com.amazonaws.services.sagemaker.sparksdk.transformation.ResponseRowDeserializer.

This parameter deserializes responses from the model, hosted in SageMaker, back into a DataFrame.

• trainingSparkDataFormat —Specifies the data format that Spark uses when uploading training
data from a DataFrame to S3. For example, "sagemaker" for protobuf format, "csv" for comma-
separated values, and "libsvm" for LibSVM format.

You can implement your own RequestRowSerializer and ResponseRowDeserializer to serialize
and deserialize rows from a data format that your inference code supports, such as .libsvm or .csv.

Use the SageMakerEstimator in a Spark Pipeline

You can use org.apache.spark.ml.Estimator estimators and org.apache.spark.ml.Model
models, and SageMakerEstimator estimators and SageMakerModel models in
org.apache.spark.ml.Pipeline pipelines, as shown in the following example:

```scala
import org.apache.spark.ml.Pipeline
import org.apache.spark.ml.feature.PCA
import org.apache.spark.sql.SparkSession
import com.amazonaws.services.sagemaker.sparksdk.IAMRole
import com.amazonaws.services.sagemaker.sparksdk.algorithms
import com.amazonaws.services.sagemaker.sparksdk.algorithms.KMeansSageMakerEstimator

val spark = SparkSession.builder.getOrCreate
// load mnist data as a dataframe from libsvm
val region = "us-east-1"
val trainingData = spark.read.format("libsvm")
  .option("numFeatures", "784")
  .load(s"s3://sagemaker-sample-data-$region/spark/mnist/train/")
val testData = spark.read.format("libsvm")
  .option("numFeatures", "784")
  .load(s"s3://sagemaker-sample-data-$region/spark/mnist/test/")
// substitute your SageMaker IAM role here
val roleArn = "arn:aws:iam::account-id:role/rolename"

val pcaEstimator = new PCA()
  .setInputCol("features")
  .setOutputCol("projectedFeatures")
  .setK(50)

val kMeansSageMakerEstimator = new KMeansSageMakerEstimator(
  sagemakerRole = IAMRole(integTestingRole),
  requestRowSerializer =
    new ProtobufRequestRowSerializer(featuresColumnName = "projectedFeatures"),
  trainingSparkDataFormatOptions = Map("featuresColumnName" -> "projectedFeatures"),
  trainingInstanceType = "ml.p2.xlarge",
  trainingInstanceCount = 1,
  endpointInstanceType = "ml.c4.xlarge",
  endpointInstanceCount = 1)
  .setK(10).setFeatureDim(50)

val pipeline = new Pipeline().setStages(Array(pcaEstimator, kMeansSageMakerEstimator))
```
// train
val pipelineModel = pipeline.fit(trainingData)

val transformedData = pipelineModel.transform(testData)
transformedData.show()

The parameter `trainingSparkDataFormatOptions` configures Spark to serialize to protobuf the "projectedFeatures" column for model training. Additionally, Spark serializes to protobuf the "label" column by default.

Because we want to make inferences using the "projectedFeatures" column, we pass the column name into the `ProtobufRequestRowSerializer`.

The following example shows a transformed DataFrame:

<table>
<thead>
<tr>
<th>label</th>
<th>features</th>
<th>projectedFeatures</th>
<th>distance_to_cluster</th>
<th>closest_cluster</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.0</td>
<td>[784, 152, 153, 154, ...</td>
<td>[880.731433034386, ...</td>
<td>1500.470703125</td>
<td>0.0</td>
</tr>
<tr>
<td>0.0</td>
<td>[784, 127, 128, 129, ...</td>
<td>[1768.5172232146, ...</td>
<td>1142.18359375</td>
<td>4.0</td>
</tr>
<tr>
<td>4.0</td>
<td>[784, 160, 161, 162, ...</td>
<td>[704.949236329314, ...</td>
<td>1386.246826171875</td>
<td>9.0</td>
</tr>
<tr>
<td>1.0</td>
<td>[784, 158, 159, 160, ...</td>
<td>[-42.328192193771, ...</td>
<td>1277.037608394375</td>
<td>5.0</td>
</tr>
<tr>
<td>9.0</td>
<td>[784, 208, 209, 210, ...</td>
<td>[374.043902028333, ...</td>
<td>1211.00927734375</td>
<td>3.0</td>
</tr>
<tr>
<td>2.0</td>
<td>[784, 155, 156, 157, ...</td>
<td>[941.26771452885, ...</td>
<td>1496.15795894375</td>
<td>8.0</td>
</tr>
<tr>
<td>1.0</td>
<td>[784, 124, 125, 126, ...</td>
<td>[30.284896410594, ...</td>
<td>1327.676357421875</td>
<td>5.0</td>
</tr>
<tr>
<td>3.0</td>
<td>[784, 151, 152, 153, ...</td>
<td>[1270.1437060252, ...</td>
<td>1570.7674560546875</td>
<td>0.0</td>
</tr>
<tr>
<td>1.0</td>
<td>[784, 152, 153, 154, ...</td>
<td>[-112.10792566485, ...</td>
<td>1037.568359375</td>
<td>5.0</td>
</tr>
<tr>
<td>4.0</td>
<td>[784, 134, 135, 161, ...</td>
<td>[452.06820676606, ...</td>
<td>1165.1236572265625</td>
<td>3.0</td>
</tr>
<tr>
<td>3.0</td>
<td>[784, 123, 124, 125, ...</td>
<td>[610.596447285937, ...</td>
<td>1325.953369140625</td>
<td>7.0</td>
</tr>
<tr>
<td>5.0</td>
<td>[784, 216, 217, 218, ...</td>
<td>[142.959601818422, ...</td>
<td>1353.490341921875</td>
<td>5.0</td>
</tr>
<tr>
<td>3.0</td>
<td>[784, 143, 144, 145, ...</td>
<td>[1036.7186253658, ...</td>
<td>1460.431585646875</td>
<td>7.0</td>
</tr>
<tr>
<td>6.0</td>
<td>[784, 72, 73, 74, 99, ...</td>
<td>[996.740157453754, ...</td>
<td>1159.8631591796875</td>
<td>2.0</td>
</tr>
<tr>
<td>1.0</td>
<td>[784, 151, 152, 153, ...</td>
<td>[-107.26076167417, ...</td>
<td>960.96362046875</td>
<td>5.0</td>
</tr>
<tr>
<td>7.0</td>
<td>[784, 211, 212, 213, ...</td>
<td>[619.77182043094, ...</td>
<td>1245.1362046875</td>
<td>6.0</td>
</tr>
<tr>
<td>2.0</td>
<td>[784, 151, 152, 153, ...</td>
<td>[850.152101817161, ...</td>
<td>1304.43774410625</td>
<td>8.0</td>
</tr>
<tr>
<td>8.0</td>
<td>[784, 159, 160, 161, ...</td>
<td>[370.04187230547, ...</td>
<td>1192.478144140625</td>
<td>0.0</td>
</tr>
<tr>
<td>6.0</td>
<td>[784, 100, 101, 102, ...</td>
<td>[546.674328209335, ...</td>
<td>1277.0908203125</td>
<td>2.0</td>
</tr>
<tr>
<td>9.0</td>
<td>[784, 209, 210, 211, ...</td>
<td>[-29.259112927426, ...</td>
<td>1245.818237046875</td>
<td>6.0</td>
</tr>
</tbody>
</table>

**SDK examples: Use Amazon SageMaker with Apache Spark**

The following list is a subset of available examples. Visit the examples website to see more.

- sagemaker-spark: a Spark library for SageMaker
- SageMaker PySpark K-Means Clustering MNIST Example
- Distributed Data Processing using Apache Spark and SageMaker Processing
- Feature processing with Spark, training with XGBoost and deploying as Inference Pipeline

**Note**
To run the notebooks on a notebook instance, see Example Notebooks (p. 134). To run the notebooks on Studio, see Create or Open an Amazon SageMaker Studio Notebook (p. 87).

**Use Chainer with Amazon SageMaker**

You can use SageMaker to train and deploy a model using custom Chainer code. The SageMaker Python SDK Chainer estimators and models and the SageMaker open-source Chainer container make writing a Chainer script and running it in SageMaker easier.
What do you want to do?

I want to train a custom Chainer model in SageMaker.

For a sample Jupyter notebook, see the Chainer example notebooks in the Amazon SageMaker Examples GitHub repository.

For documentation, see Train a Model with Chainer.
I have a Chainer model that I trained in SageMaker, and I want to deploy it to a hosted endpoint.

For more information, see Deploy Chainer models.
I have a Chainer model that I trained outside of SageMaker, and I want to deploy it to a SageMaker endpoint

For more information, see Deploy Endpoints from Model Data.
I want to see the API documentation for Amazon SageMaker Python SDK Chainer classes.

For more information, see Chainer Classes.
I want to find information about SageMaker Chainer containers.

For more information, see the SageMaker Chainer Container GitHub repository.

For information about supported Chainer versions, and for general information about writing Chainer training scripts and using Chainer estimators and models with SageMaker, see Using Chainer with the SageMaker Python SDK.

Use PyTorch with Amazon SageMaker

You can use Amazon SageMaker to train and deploy a model using custom PyTorch code. The SageMaker Python SDK PyTorch estimators and models and the SageMaker open-source PyTorch container make writing a PyTorch script and running it in SageMaker easier.

What do you want to do?

I want to train a custom PyTorch model in SageMaker.

For a sample Jupyter notebook, see the PyTorch example notebook in the Amazon SageMaker Examples GitHub repository.

For documentation, see Train a Model with PyTorch.
I have a PyTorch model that I trained in SageMaker, and I want to deploy it to a hosted endpoint.

For more information, see Deploy PyTorch models.
I have a PyTorch model that I trained outside of SageMaker, and I want to deploy it to a SageMaker endpoint

For more information, see Deploy Endpoints from Model Data.
I want to see the API documentation for Amazon SageMaker Python SDK PyTorch classes.

For more information, see PyTorch Classes.
I want to find the SageMaker PyTorch container repository.

For more information, see SageMaker PyTorch Container GitHub repository.
I want to find information about PyTorch versions supported by AWS Deep Learning Containers.

For more information, see Available Deep Learning Container Images.

For general information about writing PyTorch training scripts and using PyTorch estimators and models with SageMaker, see Using PyTorch with the SageMaker Python SDK.

R User Guide to Amazon SageMaker

This document will walk you through ways of leveraging Amazon SageMaker features using R. This guide introduces SageMaker's built-in R kernel, how to get started with R on SageMaker, and finally several example notebooks.

The examples are organized in three levels, Beginner, Intermediate, and Advanced. They start from Getting Started with R on SageMaker, continue to end-to-end machine learning with R on SageMaker, and then finish with more advanced topics such as SageMaker Processing with R script, and Bring-Your-Own (BYO) R algorithm to SageMaker.

R Kernel in SageMaker

SageMaker notebook instances support R using a pre-installed R kernel. Also, the R kernel has the reticulate library, an R to Python interface, so you can use the features of SageMaker Python SDK from within an R script. paws is an optional library that you can add to your R kernel to get further functionality.

- **reticulatelib**rary: provides an R interface to the Amazon SageMaker Python SDK. The reticulate package translates between R and Python objects.
- **pawslib**rary: provides an R interface to make API calls to AWS services, similar to how boto3 works. paws enables Python developers to create, configure, and manage AWS services, such as EC2 and S3 using R.

Get Started with R in SageMaker

- Create a Notebook Instance using the t2.medium instance type and default storage size. You can pick a faster instance and more storage if you plan to continue using the instance for more advanced examples, or create a bigger instance later.
- Wait until the status of the notebook is In Service, and then click Open Jupyter.

- Create a new notebook with R kernel from the list of available environments.
• When the new notebook is created, you should see an R logo in the upper right corner of the notebook environment, and also R as the kernel under that logo. This indicates that SageMaker has successfully launched the R kernel for this notebook.

• Alternatively, when you are in a Jupyter notebook, you can use Kernel menu, and then select R from Change Kernel option.

Example Notebooks

Prerequisites

Getting Started with R on SageMaker: This sample notebook describes how you can develop R scripts using Amazon SageMaker’s R kernel. In this notebook you set up your SageMaker environment and permissions, download the abalone dataset from the UCI Machine Learning Repository, do some basic processing and visualization on the data, then save the data as .csv format to S3.

Beginner Level

End-to-End Machine Learning with R on SageMaker: This sample notebook extends the previous prerequisite getting started notebook. You learn how to train a model on the abalone dataset that predicts abalone age as measured by the number of rings in the shell. After you train your model, you create an endpoint and deploy your model to the endpoint. With your endpoint in place, you can test the model and generate predictions. The reticulate package will be used as an R interface to the Amazon SageMaker Python SDK.
SageMaker Batch Transform using R Kernel: This sample Notebook describes how to conduct a batch transform job using SageMaker’s Transformer API and the XGBoost algorithm. The notebook also uses the Abalone dataset.

Intermediate Level

Hyperparameter Optimization for XGBoost in R: This sample notebook extends the previous beginner notebooks that use the abalone dataset and XGBoost. It describes how to do model tuning with hyperparameter optimization. You will also learn how to use batch transform for batching predictions, as well as how to create a model endpoint to make real-time predictions.

Amazon SageMaker Processing with R: SageMaker Processing lets you preprocess, post-process and run model evaluation workloads. This example shows you how to create an R script to orchestrate a Processing job.

Advanced Level

Train and Deploy Your Own R Algorithm in SageMaker: Do you already have an R algorithm, and you want to bring it into SageMaker to tune, train, or deploy it? This example walks you through how to customize SageMaker containers with custom R packages, all the way to using a hosted endpoint for inference on your R-origin model.

Use Scikit-learn with Amazon SageMaker

You can use Amazon SageMaker to train and deploy a model using custom Scikit-learn code. The SageMaker Python SDK Scikit-learn estimators and models and the SageMaker open-source Scikit-learn container make writing a Scikit-learn script and running it in SageMaker easier.

What do you want to do?

I want to use Scikit-learn for data processing, feature engineering, or model evaluation in SageMaker.

For a sample Jupyter notebook, see https://github.com/awslabs/amazon-sagemaker-examples/tree/master/sagemaker_processing/scikit_learn_data_processing_and_model_evaluation.

For documentation, see Amazon SageMaker Python SDK ReadTheDocs.

I want to train a custom Scikit-learn model in SageMaker.


For documentation, see Train a Model with Scikit-learn.

I have a Scikit-learn model that I trained in SageMaker, and I want to deploy it to a hosted endpoint.

For more information, see Deploy Scikit-learn models.

I have a Scikit-learn model that I trained outside of SageMaker, and I want to deploy it to a SageMaker endpoint.

For more information, see Deploy Endpoints from Model Data.

I want to see the API documentation for Amazon SageMaker Python SDK Scikit-learn classes.

For more information, see Scikit-learn Classes.

I want to see information about SageMaker Scikit-learn containers.

For more information, see SageMaker Scikit-learn Container GitHub repository.

For general information about writing Scikit-learn training scripts and using Scikit-learn estimators and models with SageMaker, see Using Scikit-learn with the SageMaker Python SDK.
Scikit-learn versions supported by the Amazon SageMaker Scikit-learn container: 0.20.0, 0.23-1.

Use SparkML Serving with Amazon SageMaker

The Amazon SageMaker Python SDK SparkML Serving model and predictor and the Amazon SageMaker open-source SparkML Serving container support deploying Apache Spark ML pipelines serialized with MLeap in SageMaker to get inferences.

For information about using the SparkML Serving container to deploy models to SageMaker, see SageMaker Spark ML Container GitHub repository. For information about the Amazon SageMaker Python SDK SparkML Serving model and predictors, see the SparkML Serving Model and Predictor API documentation.

Use TensorFlow with Amazon SageMaker


Use TensorFlow Version 1.11 and Later

For TensorFlow versions 1.11 and later, the Amazon SageMaker Python SDK supports script mode training scripts.

What do you want to do?

I want to train a custom TensorFlow model in SageMaker.

For a sample Jupyter notebook, see TensorFlow script mode training and serving.

For documentation, see Train a Model with TensorFlow.

I have a TensorFlow model that I trained in SageMaker, and I want to deploy it to a hosted endpoint.

For more information, see Deploy TensorFlow Serving models.

I have a TensorFlow model that I trained outside of SageMaker, and I want to deploy it to a SageMaker endpoint.

For more information, see Deploying directly from model artifacts.

I want to see the API documentation for Amazon SageMaker Python SDK TensorFlow classes.

For more information, see TensorFlow Estimator.

I want to find the SageMaker TensorFlow container repository.

For more information, see SageMaker TensorFlow Container GitHub repository.

I want to find information about TensorFlow versions supported by AWS Deep Learning Containers.

For more information, see Available Deep Learning Container Images.

For general information about writing TensorFlow script mode training scripts and using TensorFlow script mode estimators and models with SageMaker, see Using TensorFlow with the SageMaker Python SDK.

Use TensorFlow Legacy Mode for Versions 1.11 and Earlier

The Amazon SageMaker Python SDK provides a legacy mode that supports TensorFlow versions 1.11 and earlier. Use legacy mode TensorFlow training scripts to run TensorFlow jobs in SageMaker if:
• You have existing legacy mode scripts that you do not want to convert to script mode.
• You want to use a TensorFlow version earlier than 1.11.

For information about writing legacy mode TensorFlow scripts to use with the SageMaker Python SDK, see TensorFlow SageMaker Estimators and Models.

## Supported Regions and Quotas

For the AWS Regions supported by Amazon SageMaker and the Amazon Elastic Compute Cloud (Amazon EC2) instance types that are available in each Region, see Amazon SageMaker Pricing.

For a list of the SageMaker service endpoints for each Region and the SageMaker service quotas for each instance type, see Amazon SageMaker endpoints and quotas.

You can request a quota increase using Service Quotas or the AWS Support Center. To request an increase, see AWS Service Quotas in the AWS General Reference.

Amazon SageMaker Studio is available in all the AWS Regions supported by Amazon SageMaker except the AWS GovCloud (US) Regions. In the supported Regions, Studio is available in the same Availability Zones as notebook instances.

Amazon SageMaker Pipelines is available in all the AWS Regions supported by AWS except the AWS GovCloud (US) Regions. SageMaker Projects is available in the AWS regions where CodePipeline is available. For more information about CodePipeline region availability, see the AWS Regional Services List.

For the Availability Zones supported by SageMaker, see Availability Zones (p. 1722).
Get Started with Amazon SageMaker

This video shows you how to setup and use SageMaker Studio. (Length: 19:14)

Before you can use Amazon SageMaker, you must sign up for an AWS account, create an IAM admin user, and onboard to Amazon SageMaker Studio.

After you complete these tasks, try out the Get Started guides. The guides walk you through training your first model using SageMaker Studio, or the SageMaker console and the SageMaker API.

Topics
- Set Up Amazon SageMaker (p. 33)
- Onboard to Amazon SageMaker Studio (p. 34)
- SageMaker JumpStart (p. 41)
- Get Started with Amazon SageMaker Notebook Instances and SDKs (p. 58)

Set Up Amazon SageMaker

In this section, you sign up for an AWS account, create an IAM admin user, and onboard to Amazon SageMaker Studio.

If you’re new to SageMaker, we recommend that you read How Amazon SageMaker Works (p. 4).

Topics
- Create an AWS Account (p. 33)
- Create an IAM Administrator User and Group (p. 33)

Create an AWS Account

In this section, you sign up for an AWS account. If you already have an AWS account, skip this step.

When you sign up for Amazon Web Services (AWS), your AWS account is automatically signed up for all AWS services, including SageMaker. You are charged only for the services that you use.

To create an AWS account

2. Follow the online instructions.

   Part of the sign-up procedure involves receiving a phone call and entering a verification code on the phone keypad.

   Write down your AWS account ID because you'll need it for the next task.

Create an IAM Administrator User and Group

When you create an AWS account, you get a single sign-in identity that has complete access to all of the AWS services and resources in the account. This identity is called the AWS account root user. Signing in to the AWS console using the email address and password that you used to create the account gives you complete access to all of the AWS resources in your account.
We strongly recommend that you *not* use the root user for everyday tasks, even the administrative ones. Instead, adhere to the Create Individual IAM Users, an AWS Identity and Access Management (IAM) administrator user. Then securely lock away the root user credentials and use them to perform only a few account and service management tasks.

**To create an administrator user**

- Create an administrator user in your AWS account. For instructions, see [Creating Your First IAM User and Administrators Group](#) in the [IAM User Guide](#).

  **Note**
  We assume that you use administrator user credentials for the exercises and procedures in this guide. If you choose to create and use another IAM user, grant that user minimum permissions. For more information, see [Authenticating with Identities (p. 1619)](#).

---

**Onboard to Amazon SageMaker Studio**

To use Amazon SageMaker Studio and Amazon SageMaker Studio Notebooks, you must complete the Studio onboarding process using the SageMaker console.

When onboarding, you can choose to use either AWS Single Sign-On (AWS SSO) or AWS Identity and Access Management (IAM) for authentication methods. When you use IAM authentication, you can choose either the **Quick start** or the **Standard setup** procedure.

  **Note**
  If you onboard using IAM authentication and want to switch to AWS SSO authentication later, you must delete the domain created for you by SageMaker Studio. Then, you need to manually re-import all notebooks and other user data that you created. For more information, see [Delete an Amazon SageMaker Studio Domain (p. 39)](#).

The simplest way to create a Amazon SageMaker Studio account is to follow the **Quick start** procedure.

For more control, including the option of using AWS SSO authentication, use the **Standard setup** procedures.

**AWS SSO Authentication**

To use AWS SSO authentication with Amazon SageMaker Studio, you must onboard to an AWS SSO organization.

  **Note**
  The SSO organization account must be in the same AWS Region as Amazon SageMaker Studio.

AWS SSO authentication provides the following benefits over IAM authentication:

- Members given access to Studio have a unique sign-in URL that directly opens Studio, and they sign in with their SSO credentials. When you use IAM authentication, you must sign in through the SageMaker console.
- Organizations manage their members in AWS SSO instead of Studio. You can assign multiple members access to Studio at the same time. When you use IAM authentication, you must add and manage members manually one at a time using the Studio Control Panel.

**Topics**

- [Onboard to Amazon SageMaker Studio Using Quick Start (p. 35)](#)
- [Onboard to Amazon SageMaker Studio Using AWS SSO (p. 36)](#)
- [Onboard to Amazon SageMaker Studio Using IAM (p. 37)](#)
Onboard to Amazon SageMaker Studio Using Quick Start

This topic describes how to onboard to Amazon SageMaker Studio using the Quick start procedure, which uses AWS Identity and Access Management (IAM) authentication. For information on how to onboard using the standard IAM procedure, see Onboard Using IAM (p. 37).

For information on how to onboard using AWS Single Sign-On (AWS SSO), see Onboard Using SSO (p. 36).

To onboard to Studio using Quick start

1. Open the SageMaker console.
2. Choose Amazon SageMaker Studio at the top left of the page.
3. On the SageMaker Studio page, under Get started, choose Quick start.
4. For User name, keep the default name or create a new name. The name can be up to 63 characters. Valid characters: A-Z, a-z, 0-9, and -(hyphen).
5. For Execution role, choose an option from the role selector.
   - If you choose Enter a custom IAM role ARN, the role must have the AmazonSageMakerFullAccess policy attached.
   - If you choose Create a new role, the Create an IAM role dialog opens:
     - For S3 buckets you specify, specify additional S3 buckets that users of your notebooks can access. If you don't want to add access to more buckets, choose None.
     - Choose Create role. SageMaker creates a new IAM AmazonSageMaker-ExecutionPolicy role with the AmazonSageMakerFullAccess policy attached.
6. For Projects, see SageMaker Studio Permissions Required to Use Projects (p. 1443). For more control, use the standard setup procedure.
7. Choose Submit.

   Note
   If you receive an error message that you need to create a VPC, see Choose a VPC (p. 38).

On the SageMaker Studio Control Panel, under Studio Summary, wait for Status to change to Ready.

When Status is Ready, the user name that you specified is enabled and chosen. The Add user and Delete user buttons, and the Open Studio link are also enabled.


   When Studio opens you can start using it.

Now that you've onboarded to SageMaker Studio, use the following steps to access Studio later.

To access Studio after you onboard

1. Open the SageMaker console.
2. Choose Amazon SageMaker Studio at the top left of the page.
3. On the **Amazon SageMaker Studio Control Panel**, choose your user name and then choose **Open Studio**.

**To add more users**

1. On the **Amazon SageMaker Studio Control Panel**, choose **Add user**.
2. Repeat steps 4 and 5 from the first procedure, "To onboard to Studio using Quick start."
3. Choose **Submit**.

For information about using SageMaker Studio, see **SageMaker Studio** (p. 79).

**Onboard to Amazon SageMaker Studio Using AWS SSO**

This topic describes how to onboard to Amazon SageMaker Studio using AWS SSO authentication. For information on how to onboard using AWS Identity and Access Management (IAM) authentication, see **Onboard Using Quick Start** (p. 35) or **Onboard Using IAM** (p. 37).

**To onboard to Studio using AWS SSO**

1. Open the **SageMaker console**.
2. Choose **Amazon SageMaker Studio** at the top left of the page.
3. On the **SageMaker Studio** page, under **Get started**, choose **Standard setup**.
4. For **Authentication method**, choose **AWS Single Sign-On (SSO)**. A message tells you whether you have an AWS SSO account in an AWS Region supported by SageMaker Studio.
5. If you don't have an AWS SSO account in a supported Region, you must create an AWS SSO account in a supported Region before proceeding. To continue to onboard without creating a new AWS SSO account, choose the **AWS Identity and Access Management (IAM)** authentication method or the **Quick start** procedure, which also uses IAM.

For information about setting up AWS SSO for use with Studio, see **Set Up AWS SSO for Use with Amazon SageMaker Studio** (p. 37).

6. Under **Permission**, for **Execution role for all users**, choose an option from the role selector. If you choose **Create a new role**, the **Create an IAM role** dialog opens:
   a. For **S3 buckets you specify**, specify additional S3 buckets that users of your notebooks can access. If you don't want to add access to more buckets, choose **None**.
   b. Choose **Create role**. SageMaker creates a new IAM AmazonSageMaker-ExecutionPolicy role with the AmazonSageMakerFullAccess policy attached.
7. For **Projects**, see **SageMaker Studio Permissions Required to Use Projects** (p. 1443).
8. Under **Network and storage**, specify the following:
   - Your VPC information – For more information, see **Choose a VPC** (p. 38).
   - (Optional) **Storage encryption key** – SageMaker uses an AWS managed customer master key (CMK) to encrypt your Amazon Elastic File System (Amazon EFS) and Amazon Elastic Block Store (Amazon EBS) file systems by default. To use a customer managed CMK, enter its key ID or Amazon Resource Name (ARN). For more information, see **Protect Data at Rest Using Encryption** (p. 1614).
9. Choose **Assign users**. The **Assign users** page opens and displays a list of your organization's members.
10. To assign users access to SageMaker Studio, choose the check box next to their user name and choose **Assign users**.
11. Send each assigned user the **Studio address** link shown under **Studio Summary**. Your AWS SSO users go to this address to access Studio.

**To access Studio after onboarding**

After you are given access to Studio, you are sent an email inviting you to create a password and activate your AWS SSO account. The email also contains the URL to sign in to Studio.

After you activate your account, go to the Studio URL, sign in, and wait for your user profile to be created. On subsequent visits, you only need to wait for Studio to load.

Bookmark the Studio URL. The URL is also available in the Studio Control Panel.

For information about using SageMaker Studio, see [SageMaker Studio](p. 79).

**Set Up AWS SSO for Use with Amazon SageMaker Studio**

To use AWS SSO authentication, you must belong to an AWS SSO organization. If you don't belong to an AWS SSO organization, you can create one with the following procedure.

**To create an AWS SSO organization**

1. On the **SageMaker Studio** page, under **Get started**, choose **Standard setup**.
2. Choose the **set up an account** link to open the AWS Single Sign-On (AWS SSO) console.
3. Choose **Enable AWS SSO**, and then choose **Create AWS organization**.

   When your organization has been created, the AWS SSO dashboard opens. AWS also sends you email to verify the email address associated with the organization.
4. To add a user to your AWS SSO organization, in the navigation pane, choose **Users**.
5. On the **Users** page, choose **Add user**. Under **User details**, specify all required fields. For **Password**, choose **Send an email to the user**.
6. Choose **Next: Groups** and then choose **Add user**. AWS sends an email to the user inviting them to create a password and activate their AWS SSO account.
7. To add more users, repeat steps 4 through 6.

Return to SageMaker Studio to continue to onboard using AWS SSO authentication.

**Onboard to Amazon SageMaker Studio Using IAM**

This topic describes how to onboard to Amazon SageMaker Studio using the standard setup procedure for AWS Identity and Access Management (IAM) authentication. To onboard faster using IAM, see [Onboard Using Quick Start](p. 35).

For information on how to onboard using AWS Single Sign-On (AWS SSO), see [Onboard Using SSO](p. 36).

**To onboard to Studio using IAM**

1. Open the **SageMaker console**.
2. Choose **Amazon SageMaker Studio** at the top left of the page.
3. On the **SageMaker Studio** page, under **Get started**, choose **Standard setup**.
4. For **Authentication method**, choose **AWS Identity and Access Management (IAM)**.
5. Under **Permission**, for **Execution role for all users**, choose an option from the role selector. If you choose **Create a new role**, the **Create an IAM role** dialog opens:
Choose a VPC

This topic provides detailed information on choosing an Amazon Virtual Private Cloud (VPC) when you onboard to Amazon SageMaker Studio. For more information on onboarding to SageMaker Studio, see Onboard to Amazon SageMaker Studio (p. 34).

By default, SageMaker Studio uses two VPCs. One VPC is managed by Amazon SageMaker and provides direct internet access. You specify the other VPC, which provides encrypted traffic between the domain and your Amazon Elastic File System (EFS) volume.

For information about using SageMaker Studio, see SageMaker Studio (p. 79).
You can change this behavior so that Studio sends all traffic over your specified VPC. When you choose this option, you must provide the subnets, security groups, and interface endpoints that are necessary to communicate with the SageMaker API and SageMaker runtime, and various AWS services, such as Amazon Simple Storage Service (Amazon S3) and Amazon CloudWatch, that are used by Studio and your Studio notebooks.

When you onboard to Studio, you tell Studio to send all traffic over your VPC by setting the network access type to **VPC only**.

**To specify the VPC information**

When you specify the VPC entities (that is, the VPC, subnet, or security group) in the following procedure, one of three options is presented based on the number of entities you have in the current AWS Region. The behavior is as follows:

- One entity – Studio uses that entity. This can’t be changed.
- Multiple entities – You must choose the entities from the dropdown list.
- No entities – You must create one or more entities in order to use Studio. Choose Create <entity> to open the VPC console in a new browser tab. After you create the entities, return to the Studio Get started page to continue the onboarding process.

This procedure is part of the SageMaker Studio onboarding process when you choose **Standard setup**. Your VPC information is specified under the **Network** section.

1. Choose the VPC.
2. Choose one or more subnets. If you don’t choose any subnets, SageMaker uses all the subnets in the VPC.
3. Select the network access type.
   - **Public internet only** – Non-EFS traffic goes through a SageMaker managed VPC, which allows internet access. Traffic between the domain and your Amazon EFS volume is through the specified VPC.
   - **VPC only** – All Studio traffic is through the specified VPC and subnets. Internet access is disabled by default.
4. Choose the security groups. If you chose **Public internet only**, this step is optional. If you chose **VPC only**, this step is required.

For VPC requirements in **VPC only** mode, see Connect SageMaker Studio Notebooks to Resources in a VPC (p. 1679).

**Delete an Amazon SageMaker Studio Domain**

When you onboard to Amazon SageMaker Studio using IAM authentication, Studio creates a domain for your account. A domain consists of a list of authorized users, configuration settings, and an Amazon Elastic File System (Amazon EFS) volume, which contains data for the users, including notebooks, resources, and artifacts. A user can have multiple applications (apps) which support the reading and execution experience of the user’s notebooks, terminals, and consoles.

To return Studio to the state it was in before you onboarded, you must delete this domain. Before a domain is deleted, the Amazon EFS volume is detached but not deleted. To delete the EFS volume, see Manage Your EFS Storage Volume (p. 120).

You must delete the domain if you want to switch authentication modes from IAM to AWS SSO.

To delete a domain, the domain cannot contain any user profiles. To delete a user profile, the profile cannot contain any non-failed apps.
When you delete these resources, the following occurs:

- **App** – The data (files and notebooks) in a user's home directory is saved. Unsaved notebook data is lost.

- **User profile** – The user is no longer able to sign in to Studio and loses access to their home directory but the data is not deleted. An admin can retrieve the data from the Amazon EFS volume where it is stored under the user's AWS account.

**Note**
You must have admin permission to delete a domain.

You can only delete an app whose status is **InService**, which is displayed as **Ready** in Studio. An app whose status is **Failed** doesn't need to be deleted to delete the containing domain. In Studio, an attempt to delete an app in the failed state results in an error.

**Topics**
- Delete a SageMaker Studio Domain (Studio) (p. 40)
- Delete a SageMaker Studio Domain (CLI) (p. 40)

**Delete a SageMaker Studio Domain (Studio)**

**To delete a domain**

1. Open the SageMaker console.
2. Choose **Amazon SageMaker Studio** at the top left of the page to open the **Amazon SageMaker Studio Control Panel**.
3. Repeat the following steps for each user in the **User name** list.
   a. Choose the user.
   b. On the **User Details** page, for each non-failed app in the **Apps** list, choose **Delete app**.
   c. On the **Delete app** dialog, choose **Yes, delete app**, type **delete** in the confirmation field, and then choose **Delete**.
   d. When the **Status** for all apps show as **Deleted**, choose **Delete user**.

   **Important**
   When a user is deleted, they lose access to the Amazon EFS volume that contains their data, including notebooks and other artifacts.

4. When all users are deleted, choose **Delete Studio**.
5. On the **Delete Studio** dialog, choose **Yes, delete Studio**, type **delete** in the confirmation field, and then choose **Delete**.

**Delete a SageMaker Studio Domain (CLI)**

For a list of AWS Regions supported by Amazon SageMaker Studio, see Onboard to Amazon SageMaker Studio (p. 34).

**To delete a domain**

1. Retrieve the list of domains in your account.

```
aws --region Region sagemaker list-domains
```
2. Retrieve the list of applications for the domain to be deleted.

```bash
aws --region Region sagemaker list-apps \
    --domain-id-equals DomainId
```

3. Delete each application in the list.

```bash
aws --region Region sagemaker delete-app \
    --domain-id DomainId \
    --app-name AppName \
    --app-type AppType \
    --user-profile-name UserProfileName
```

4. Retrieve the list of user profiles in the domain.

```bash
aws --region Region sagemaker list-user-profiles \
    --domain-id-equals DomainId
```

5. Delete each user profile in the list.

```bash
aws --region Region sagemaker delete-user-profile \
    --domain-id DomainId \
    --user-profile-name UserProfileName
```

6. Delete the domain.

```bash
aws --region Region sagemaker delete-domain \
    --domain-id DomainId
```

---

**SageMaker JumpStart**

**Important**

To use new features with an existing notebook instance or Studio app, you must restart the notebook instance or the Studio app to get the latest updates.

You can use SageMaker JumpStart to learn about SageMaker features and capabilities through curated 1-click solutions, example notebooks, and pretrained models that you can deploy. You can also fine-tune the models and deploy them.

To access JumpStart, you must first launch SageMaker Studio. JumpStart features are not available in SageMaker notebook instances, and you can't access them through SageMaker APIs or the AWS CLI.

Open JumpStart by choosing the JumpStart icon (🚀) in the left sidebar.

In the file and resource browser (the left pane), you can find JumpStart options. From here you can choose to browse JumpStart for solutions, models, notebooks, and other resources, or you can view your currently launched solutions, endpoints, and training jobs.

To see what JumpStart has to offer, choose the JumpStart icon, and then choose **Browse JumpStart**. JumpStart opens in a new tab in the main work area. Here you can browse 1-click solutions, models, example notebooks, blogs, and video tutorials.
SAGEMAKER JUMPSTART LAUNCHED ASSETS

View launched solutions, deployed model endpoints and training jobs created with JumpStart.

Browse JumpStart

Solutions (1)

half a minute ago

Your launched solutions.

Fraud Detection in Financial Transactions

Ready · 39 minutes ago
Important
Amazon SageMaker JumpStart makes certain content available from third-party sources. This content may be subject to separate license terms. You are responsible for reviewing and complying with any applicable license terms and making sure they are acceptable for your use case before downloading or using the content.

Using JumpStart
At the top of the JumpStart page you can use search to look for topics of interest.

SageMaker JumpStart: Discover, train, and deploy ML models
One-click solutions, pretrained models, and example notebooks

You can find JumpStart resources by using search, or by browsing each category that follows the search panel:

- **Solutions** – Launch end-to-end machine learning solutions that tie SageMaker to other AWS services with one click.
- **Text models** – Deploy and fine-tune pretrained transformers for various natural language processing use cases.
- **Vision models** – Deploy and fine-tune pretrained models for image classification and object detection with one click.
• **SageMaker algorithms** – Train and deploy SageMaker built-in Algorithms for various problem types with these example notebooks.

• **Example notebooks** – Run example notebooks that use SageMaker features like spot instance training and experiments over a large variety of model types and use cases.

• **Blogs** – Read deep dives and solutions from machine learning experts hosted by Amazon.

• **Video tutorials** – Watch video tutorials for SageMaker features and machine learning use cases from machine learning experts hosted by Amazon.

## Solutions

When you choose a solution, JumpStart provides a description of the solution and a **Launch** button. There are no configuration options. Solutions launch all of the resources necessary to run the solution, including training and model hosting instances. After JumpStart launches the solution, JumpStart provides a link to a notebook that you can use to explore the solution’s features. You can delete all of the solution’s resources by choosing **Delete solution resources**.

## Models

Models are available for quick deployment directly from JumpStart. You can also fine-tune some of these models. When you browse the models, you can scroll to the deploy and fine-tune sections to the **Description** section. In the **Description** section, you can learn more about the model, including what it can do with the model, what kind of inputs and outputs are expected, and the kind of data you need if you want to use transfer learning to fine-tune the model.

### Deploy a model

When you deploy a model from JumpStart, SageMaker hosts the model and deploys an endpoint that you can use for inference. JumpStart also provides an example notebook that you can use to access the model after it's deployed.

### Model Deployment Configuration

After you choose a model, the **Deploy Model** pane opens. Choose **Deployment Configuration** to configure your model deployment.
The default Machine Type for deploying a model depends on the model. The machine type is the hardware that the training job runs on. In the following example, the ml.m5.large instance is the default for this particular BERT model.

You can also change the Endpoint Name.

---

**Deploy Model**

Deploy a pretrained model to an endpoint for inference. Deploying on SageMaker hosts a specified compute instance and creates an internal API endpoint. JumpStart will provide a notebook to access the model after it is deployed. Learn more.

- **Deployment Configuration**
  Customize the machine type and endpoint name. Learn more.

- **Machine Type**
  [ML.M5.Large]

- **Endpoint Name**
  [tf-tc-bert-en-uncased-l-12-h-768-a-12-2]

- **Reset to default**

---

**Fine-Tune a Model**

Fine-tuning trains a pretrained model on a new dataset without training from scratch. This process, also known as transfer learning, can produce accurate models with smaller datasets and less training time.

**Fine-Tuning Data Source**

When you fine-tune a model, you can use the default dataset or choose your own data, which is located in an S3 bucket.
Fine-tune the Model on a New Dataset

The model can be fine-tuned to any given dataset comprising images belonging to any number of classes.

The model available for fine-tuning attaches a classification layer to the corresponding model available on TensorFlow, and initializes the layer parameters to random values. The dimension of the classification layer is determined based on the number of classes in the new fine-tuning step fine-tunes the classification layer parameters while keeping the parameters of the lower-level feature extractor model frozen, and returns the fine-tuned model. The objective is to minimize the cross-entropy error on the input data. The model returned by fine-tuning can be further deployed for inference.

- **Input**: A directory with as many sub-directories as the number of classes.
  - Each sub-directory should have images belonging to that class in .jpg format.
- **Output**: A trained model that can be deployed for inference.
  - A label mapping file is saved along with the trained model file on the s3 bucket.

We provide tf_flowers dataset as a default dataset for fine-tuning the model. tf_flowers comprises images of five types of flowers. The dataset has been downloaded from TensorFlow :point_up: Apache. Citation: @ONLINE {tfflowers, author = "The TensorFlow Team", title = "Flowers", month = "2019", url = "http://download.tensorflow.org/example_images/flower_photos.tgz"}
To browse the buckets available to you, choose **Find S3 bucket**. These buckets are limited by the permissions used to set up your Studio account. You can also specify an S3 URI by choosing **Enter S3 bucket location**.

**Fine-tune Model**

Create a training job to fine-tune this pretrained model to fit your own data. Fine-tuning this model on a new dataset without training from scratch. It can produce accurate models with less training time. [Learn more](#).

- **Data Source**

Select the default dataset, or use your own data to fine-tune this model.

- Default dataset
- Find S3 bucket
- Enter S3 bucket location

This option will fit the model to the default dataset. [Learn more](#).

**Deployment Configuration**

**Hyper-parameters**

**Tip**

To find out how to format the data in your bucket, choose **Learn more**. Also, the description section for the model has detailed information about inputs and outputs.

For text models:

- The bucket must have a data.csv file.
- The first column must be a unique integer for the class label. For example: 1, 2, 3, 4, n.
- The second column must be a string.
- The second column should have the corresponding text that matches the type and language for the model.
For vision models:

- The bucket must have as many subdirectories as the number of classes.
- Each subdirectory should contain images that belong to that class in .jpg format.

Note
The S3 bucket must be in the same AWS Region where you're running SageMaker Studio because SageMaker doesn't allow cross-region requests.

Fine-Tuning deployment configuration

The p3 family is recommended as the fastest for deep learning training, and this is recommended for fine-tuning a model. The following chart shows the number of GPUs in each instance type. There are other available options that you can choose from, including p2 and g4 instance types.

<table>
<thead>
<tr>
<th>Instance type</th>
<th>GPUs</th>
</tr>
</thead>
<tbody>
<tr>
<td>p3.2xlarge</td>
<td>1</td>
</tr>
<tr>
<td>p3.8xlarge</td>
<td>4</td>
</tr>
<tr>
<td>p3.16xlarge</td>
<td>8</td>
</tr>
<tr>
<td>p3dn.24xlarge</td>
<td>8</td>
</tr>
</tbody>
</table>

Hyperparameters

You can customize the hyperparameters of the training job that are used to fine-tune the model.

If you use the default dataset for text models without changing the hyperparameters, you get a nearly identical model as a result. For vision models, the default dataset is different from the dataset used to train the pretrained models, so your model is different as a result.

You have the following hyperparameter options:

- **Epochs** – One epoch is one cycle through the entire dataset. Multiple intervals complete a batch, and multiple batches eventually complete an epoch. Multiple epochs are run until the accuracy of the model reaches an acceptable level, or in other words, when the error rate drops below an acceptable level.

- **Learning rate** – The amount that values should be changed between epochs. As the model is refined, its internal weights are being nudged and error rates are checked to see if the model improves. A typical learning rate is 0.1 or 0.01, where 0.01 is a much smaller adjustment and could cause the training to take a long time to converge, whereas 0.1 is much larger and can cause the training to overshoot. It is one of the primary hyperparameters that you might adjust for training your model. Note that for text models, a much smaller learning rate (5e-5 for BERT) can result in a more accurate model.

- **Batch size** – The number of records from the dataset that to be selected for each interval to send to the GPUs available in training. In an image example, you might send out 32 images per GPU, so 32 would be your batch size. If you choose an instance type with more than one GPU, the batch is divided by the number of GPUs. Suggested batch size varies depending on the data and the model that you are using. For example, how you optimize for image data differs from how you handle language data. In the instance type chart in the deployment configuration section, you can see the number of GPUs per instance type. Start with a standard recommended batch size (for example, 32 for a vision model). Then, multiply this by the number of GPUs in the instance type that you selected. For example, if
you're using a p3.8xlarge, this would be 32(batch size)*4(GPUs), for a total of 128 as your batch size adjusted for the number of GPUs. For a text model like BERT, try starting with a batch size of 64, and then reduce as needed.

**Hyper-parameters**

Customize the hyper-parameters for the training job from the default values we've pre-filled.

**Epochs**

3

**Learning Rate**

0.00002

**Batch Size**

8

Next Steps

For a deep dive into Studio features:

- Amazon SageMaker Studio Tour (p. 50) – An end-to-end tour of the main features of SageMaker Studio.

Training Output

When the fine-tuning process is complete, JumpStart provides information about the model: parent model, training job name, training job Amazon Resource Name (ARN), training time, and output path. The output path is where you can find your new model in an S3 bucket. The folder structure uses the model name you provided and the model file is in an /output subfolder and it's always named model.tar.gz.

Example: s3://bucket/model-name/output/model.tar.gz
Amazon SageMaker Studio Tour

This walkthrough takes you on a tour of the main features of Amazon SageMaker Studio using the xgboost_customer_churn_studio.ipynb sample notebook from the aws/amazon-sagemaker-examples repository. It is intended that you proceed through the walkthrough and run the notebook in Studio at the same time.

The code in the notebook trains multiple models and sets up the SageMaker Debugger and SageMaker Model Monitor. The walkthrough shows you how to view the trials, compare the resulting models, show the debugger results, and deploy the best model using the SageMaker Studio UI. You don’t need to understand the code to follow this walkthrough.

For a series of videos that shows how to use the main features of SageMaker Studio, see NEW! Amazon SageMaker Studio on YouTube.

Prerequisites

To run the notebook for this tour, you need:

- An AWS SSO or IAM account to sign in to Studio. For information, see Onboard to Amazon SageMaker Studio (p. 34).
- Basic familiarity with the Studio user interface and Jupyter notebooks. For information, see Amazon SageMaker Studio UI Overview (p. 79).
- A copy of the aws/amazon-sagemaker-examples repository in your Studio environment.

To clone the repository

1. Sign in to SageMaker Studio. For AWS SSO users, sign in using the URL from your invitation email. For IAM users, follow these steps.
   a. Sign in to the SageMaker console.
   b. Choose Amazon SageMaker Studio in the left navigation pane.
   c. Choose Open Studio in the row next to your user name.
2. On the top menu, choose File then New then Terminal.
3. At the command prompt, run the following command.

   `git clone https://github.com/aws/amazon-sagemaker-examples.git`

   **Note**
   If you encounter an error when you run the sample notebook, and some time has passed from when you cloned the repository, review the notebook on the remote repository for updates.

Topics

- Open the Amazon SageMaker Studio Notebook (p. 51)
- Keep Track of the Machine Learning Experiment (p. 51)
- Create a Chart to Visualize Data (p. 54)
- Debug the Training Job (p. 55)
- Deploy the Best Model (p. 56)
- Monitor the Deployed Model (p. 57)
- Clean Up Resources (p. 58)
Open the Amazon SageMaker Studio Notebook

Amazon SageMaker Studio notebooks are collaborative Jupyter notebooks that are built into SageMaker Studio. You can launch Studio notebooks without setting up compute instances and file storage, so you can get started fast.

You can share notebooks with others in your organization, so that they can easily reproduce your results and collaborate while building models and exploring your data.

For more information about SageMaker Studio notebooks, see Use Amazon SageMaker Studio Notebooks (p. 84).

To open the xgboost_customer_churn_studio notebook

1. Sign in to Studio. For more information, see Onboard to Amazon SageMaker Studio (p. 34).
2. Choose the file browser icon (文件浏览器).
4. Double-click xgboost_customer_churn_studio.ipynb to open the notebook.
5. In the Select Kernel dialog, choose Python 3 (Data Science), then choose Select.

Your screen should resemble the following:

Next, you use the notebook to create an experiment.

Keep Track of the Machine Learning Experiment

Amazon SageMaker Experiments lets you organize, track, compare, and evaluate your machine learning experiments. An experiment is composed of multiple trials with the same objective. Each trial is composed of multiple trial components such as a preprocessing job and a training job.

First you create an experiment, then you create a trial which assigns it to the experiment. Next, you create a training job as a trial component and associate the component with the trial. For more information, see Manage Machine Learning with Amazon SageMaker Experiments (p. 853).

To create an experiment and a trial with a training job

1. Scroll down the notebook and choose the section titled Amazon SageMaker Experiments.
2. In the Studio main menu, choose Run and then Run All Above Selected Cell.
3. Hold down the Shift key and press Enter to run the next code cell, which creates an experiment by calling the create method of the Experiment class.

```python
sess = sagemaker.session.Session()
create_date = strftime("%Y-%m-%d-%H-%M-%S", gmtime())
```
customer_churn_experiment = Experiment.create(experiment_name="customer-churn-prediction-xgboost-{}").format(create_date),
    description="Using xgboost to predict customer churn",
    sagemaker_boto_client=boto3.client('sagemaker'))

4. In the left sidebar, choose the SageMaker Experiment List icon ( ) to see the experiment (named customer-churn-prediction-xgboost...) in the experiments list. You might need to refresh the list.

5. Run the next two code cells. The first cell defines the hyperparameters to use in the training job. The second cell creates a trial that is assigned to the experiment that was created in the previous step. Next, the cell creates a training job as a trial component, then runs the trial by calling the fit method of the Estimator class. It can take several minutes for the training job to complete.

   Note
   The output of the training job includes a long list of messages like [0]#011train-error:0.077154#011validation-error:0.099099. These aren't errors due to the training job but are the results from the model training process.

trial = Trial.create(trial_name="algorithm-mode-trial-{}").format(strftime("%Y-%m-%d-%H-%M-%S", gmtime())),
    experiment_name=customer_churn_experiment.experiment_name,
    sagemaker_boto_client=boto3.client('sagemaker'))

xgb = sagemaker.estimator.Estimator(image_name=docker_image_name,
    role=role,
    hyperparameters=hyperparams,
    train_instance_count=1,
    train_instance_type='ml.m4.xlarge',
    output_path='s3://{}/output'.format(bucket,
    base_job_name="demo-xgboost-customer-churn",
    sagemaker_session=sess)

xgb.fit({'train': s3_input_train,
    'validation': s3_input_validation},
        experiment_config={
    "ExperimentName": customer_churn_experiment.experiment_name,
    "TrialName": trial.trial_name,
    "TrialComponentDisplayName": "Training",
})

6. In the experiments list, double-click the experiment name to see the trial (named algorithm-mode-trial...).
7. Double-click the trial name to see the associated trial component (named Training).
8. Double-click the Training trial component to open the Describe Trial Component tab. You can follow the progress of the training job here.

After the trial finishes, you can see details about the training job, such as metrics and hyperparameters, charts that visualize the training results. To see the billable time and instance type, choose the AWS Settings heading.

Next, the notebook creates and compares multiple trials that use different values for the min_child_weight hyperparameter.

To create and compare multiple trials

1. Scroll to the section of the notebook titled Trying other hyperparameter values.
2. Run the following cell that creates and runs five trials, each with a different value of the min_child_weight hyperparameter.

Note
In the previous step of creating a single trial, the output of the training job is displayed. Here, the output is suppressed as it would display about three thousand lines.

```python
min_child_weights = [1, 2, 4, 8, 10]
for weight in min_child_weights:
    hyperparams["min_child_weight"] = weight
    trial = Trial.create(trial_name="algorithm-mode-trial-{}-weight-{}").format(strftime("%Y-%m-%d-%H-%M-%S", gmtime()), weight),
    experiment_name=customer_churn_experiment.experiment_name,
    sagemaker_boto_client=boto3.client('sagemaker'))
    t_xgb = sagemaker.estimator.Estimator(image_name=docker_image_name,
    role=role,
    hyperparameters=hyperparams,
    train_instance_count=1,
    train_instance_type='ml.m4.xlarge',
    output_path='s3://{}/'.format(bucket, prefix),
    base_job_name="demo-xgboost-customer-churn",
    sagemaker_session=sess)

    t_xgb.fit({'train': s3_input_train,
        'validation': s3_input_validation},
        wait=False,
        experiment_config={
          "ExperimentName": customer_churn_experiment.experiment_name,
          "TrialName": trial.trial_name,
          "TrialComponentDisplayName": "Training",
        })
```
3. To follow the progress and view the results in Studio, choose the **Home** icon above **TRIAL COMPONENTS**.

4. Right-click the experiment name and choose **Open in trial component list**. You can see details about the trials, compare trials to find the best performing model, and create charts to visualize training results.

![Image of Amazon SageMaker Studio interface showing the **TRIAL COMPONENTS** list with a scatter plot of validation error against min_child_weight hyperparameter values.

After all the trials finish, sort the trials by choosing the **validation:error** heading. In a later section, you will deploy the trial with the lowest **validation:error**.

### Create a Chart to Visualize Data

To visualize data after the training jobs run, you can create charts in Amazon SageMaker Studio. In this notebook, the training jobs run for a very short time, so they don't create much data. Because of this, you create a scatter plot of the **validation:error_last** metric (final validation error) for each of the **min_child_weight** hyperparameter values that were specified in the training jobs.

**To create the scatter plot**

1. In the **TRIAL COMPONENTS** list, multi-select the five trials from the previous step, then choose **Add chart**.

2. If the **CHART PROPERTIES** pane isn't open, choose the **Settings** icon ( espanol ) in the upper right corner to open it. Choose the **Settings** icon again when you want to close the pane.

3. Configure the chart properties as follows:
   - For **Data type**, choose **Summary statistics**.
   - For **Chart type**, choose **Scatter plot**.
   - For **X-axis**, choose **min_child_weight**.
   - For **Y-axis**, choose **validation:error_last**.
   - For **Color**, choose **trialComponentName**.

Studio displays the scatter plot.
Next, the notebook sets up the SageMaker Debugger.

**Debug the Training Job**

Amazon SageMaker Debugger helps you analyze your training jobs and find problems. It monitors, records, and analyzes tensor data from training jobs and checks the training tensors against a set of rules that you specify. You can choose from a list of built-in rules, or create your own custom rules. For more information, see Amazon SageMaker Debugger (p. 871).

**To debug a training job**

1. To specify the rules to use to analyze your training job, run the following cell in the section titled Amazon SageMaker Debugger.

   ```python
   debug_rules = [Rule.sagemaker(rule_configs.loss_not_decreasing()),
                  Rule.sagemaker(rule_configs.overtraining()),
                  Rule.sagemaker(rule_configs.overfit())
   ]
   ```

2. Run the remaining cells in the section to create a new trial using the debug rules. Note the `rules=debug_rules` argument that is added to the `fit` call.

   ```python
   entry_point_script = "xgboost_customer_churn.py"
   trial = Trial.create(...)

   framework_xgb = sagemaker.xgboost.XGBoost(image_name=docker_image_name,
                                             entry_point=entry_point_script,
                                             role=role,
                                             framework_version="0.90-2",
                                             py_version="py3",
                                             hyperparameters=hyperparams,
                                             train_instance_count=1,
                                             train_instance_type="ml.m4.xlarge",
                                             output_path='s3://{}/{}')
   ```
base_job_name="demo-xgboost-customer-churn",
sagemaker_session=sess,
    rules=debug_rules
)

framework_xgb.fit(...) 

3. In the experiments list, double-click the experiment name to see the trials list.
4. In the trials list, right-click the debug trial (named framework-mode-trial...) and choose Open in trial component list.
5. In the trial components list, right-click the trial and choose Open in trial details.
6. To see the results for each debug rule that you specified, choose the Debugger heading.

Notice that the training job passed all three of the rules that were configured for the job. If Debugger had found issues, you could choose the rule in the list to see more information in the Debugger Details tab.

Next, you deploy the model.

**Deploy the Best Model**

You can create an endpoint and deploy a model using the SDK or the Amazon SageMaker Studio UI. The notebook shows you how to deploy using the SDK. For this tour, we show you how to deploy using the Studio UI. After you deploy the model, you can set up the SageMaker Model Monitor to monitor the endpoint.

**To deploy a model using the Studio UI**

1. In the experiments list, right-click the experiment and choose Open in trial component list.
2. To sort the trials, choose the validation: error heading.
3. Right-click the trial with the lowest validation: error and choose Deploy model.
4. Under Deploy model, specify a name for the endpoint that will host the model.
5. Choose Deploy model.
Next, the notebook sets up the SageMaker Model Monitor.

**Monitor the Deployed Model**

Monitor the quality of your deployed models with Amazon SageMaker Model Monitor. Model Monitor runs monitoring jobs on the endpoints where models are deployed. You can use its built-in monitoring capabilities, which don't require coding, or you can write code for custom analysis. For more information, see Amazon SageMaker Model Monitor (p. 1174).

The notebook first creates a processing job to generate baseline statistics. Next, it creates a monitoring schedule, with a polling time of one hour, that compares the recent data captures to the baseline.

**To monitor the deployed model**

1. Run all the code cells in the notebook sections titled **Host the model** and **Amazon SageMaker Model Monitor**. This can take some time. Note that this creates a different endpoint than the endpoint you created in the previous section.

2. In Studio, choose the **SageMaker Endpoint List** icon ( ).

3. Double-click the endpoint (named `demo-xgboost-customer-churn...`) that was created by the notebook.
In the Monitoring job history list, you can see any issues that the monitoring jobs found. To see details about an issue, choose the issue.

**Important**
You must shut down the kernel to stop monitoring. To shut down the kernel, from the top Studio menu, choose **Kernel** then **Shut Down Kernel**.

**Clean Up Resources**

To stop incurring charges, you should clean up the resources that were created.

To clean up the resources, follow the instructions in the notebook section titled **Clean up**.

**Get Started with Amazon SageMaker Notebook Instances and SDKs**

This video shows you how to setup and use SageMaker notebook instances. (Length: 26:04)

The best way to learn how to use Amazon SageMaker is to create, train, and deploy a simple machine learning model. To do this, you need the following:

- A dataset. You use the MNIST (Modified National Institute of Standards and Technology database) dataset of images of handwritten, single digit numbers. This dataset provides a training set of 50,000 example images of handwritten single-digit numbers, a validation set of 10,000 images, and a test dataset of 10,000 images. You provide this dataset to the algorithm for model training. For more information about the MNIST dataset, see **MNIST Dataset**.

- An algorithm. You use the XGBoost algorithm provided by SageMaker to train the model using the MNIST dataset. During model training, the algorithm assigns example data of handwritten numbers into 10 clusters: one for each number, 0 through 9. For more information about the algorithm, see **XGBoost Algorithm** (p. 813).

You also need a few resources for storing your data and running the code in this exercise:

- An Amazon Simple Storage Service (Amazon S3) bucket to store the training data and the model artifacts that SageMaker creates when it trains the model.

- A SageMaker notebook instance to prepare and process data and to train and deploy a machine learning model.
Step 1: Create an Amazon S3 Bucket

Training a model produces the following:

- The model training data
- Model artifacts, which Amazon SageMaker generates during model training

You save these in an Amazon Simple Storage Service (Amazon S3) bucket: You can store datasets that you use as your training data and model artifacts that are the output of a training job in a single bucket or in two separate buckets. For this exercise and others in this guide, one bucket is sufficient. If you already have S3 buckets, you can use them, or you can create new ones.

To create a bucket, follow the instructions in Create a Bucket in the Amazon Simple Storage Service Console User Guide. Include sagemaker in the bucket name. For example, sagemaker-datetime.

Note

Amazon SageMaker needs permission to access these buckets. You grant permission with an IAM role, which you create in the next step when you create a SageMaker notebook instance. This IAM role automatically gets permissions to access any bucket that has sagemaker in the name. It gets these permissions through the AmazonSageMakerFullAccess policy, which SageMaker attaches to the role. If you add a policy to the role that grants the SageMaker service principal S3FullAccess permission, the name of the bucket does not need to contain sagemaker.
Next Step

Step 2: Create an Amazon SageMaker Notebook Instance (p. 60)

Step 2: Create an Amazon SageMaker Notebook Instance

An Amazon SageMaker notebook instance is a fully managed machine learning (ML) Amazon Elastic Compute Cloud (Amazon EC2) compute instance that runs the Jupyter Notebook App. You use the notebook instance to create and manage Jupyter notebooks that you can use to prepare and process data and to train and deploy machine learning models. For more information, see Explore, Analyze, and Process Data (p. 5).

Note
If necessary, you can change the notebook instance settings, including the ML compute instance type, later.

To create a SageMaker notebook instance

1. Open the Amazon SageMaker console at https://console.aws.amazon.com/sagemaker/.
2. Choose Notebook instances, then choose Create notebook instance.
3. On the Create notebook instance page, provide the following information (if a field is not mentioned, leave the default values):
   a. For Notebook instance name, type a name for your notebook instance.
   b. For Instance type, choose ml.t2.medium. This is the least expensive instance type that notebook instances support, and it suffices for this exercise. If a ml.t2.medium instance type isn't available in your AWS Region, choose ml.t3.medium.
   c. For IAM role, choose Create a new role, then choose Create role.
   d. Choose Create notebook instance.

   In a few minutes, Amazon SageMaker launches an ML compute instance—in this case, a notebook instance—and attaches an ML storage volume to it. The notebook instance has a preconfigured Jupyter notebook server and a set of Anaconda libraries.

Next Step

Step 3: Create a Jupyter Notebook (p. 60).

Step 3: Create a Jupyter Notebook

You can create a Jupyter notebook in the notebook instance you created in Step 2: Create an Amazon SageMaker Notebook Instance (p. 60), and create a cell that gets the IAM role that your notebook needs to run Amazon SageMaker APIs and specifies the name of the Amazon S3 bucket that you will use to store the datasets that you use for your training data and the model artifacts that a SageMaker training job outputs.

To create a Jupyter notebook

1. Open the notebook instance.
b. Open the Notebook Instances, and then open the notebook instance you created by choosing either Open Jupyter for classic Jupyter view or Open JupyterLab for JupyterLab view next to the name of the notebook instance.

**Note**
If you see Pending to the right of the notebook instance in the Status column, your notebook is still being created. The status will change to InService when the notebook is ready for use.

2. Create a notebook.
   a. If you opened the notebook in Jupyter classic view, on the Files tab, choose New, and conda_python3. This preinstalled environment includes the default Anaconda installation and Python 3.
   b. If you opened the notebook in JupyterLab view, on the File menu, choose New, and then choose Notebook. For Select Kernel, choose conda_python3. This preinstalled environment includes the default Anaconda installation and Python 3.

3. In the Jupyter notebook, choose File and Save as, and name the notebook.

**Next Step**

**Step 4: Download, Explore, and Transform the Training Data (p. 61)**

**Step 4: Download, Explore, and Transform the Training Data**

Download the MNIST dataset to your notebook instance, review the data, transform it, and upload it to your S3 bucket.

You transform the data by changing its format from `numpy.array` to comma-separated values (CSV). The XGBoost Algorithm (p. 813) expects input in either the LIBSVM or CSV format. LIBSVM is an open source machine learning library. In this exercise, you use CSV format because it's simpler.

**Topics**
- Step 4.1: Download the MNIST Dataset (p. 61)
- Step 4.2: Explore the Training Dataset (p. 62)
- Step 4.3: Transform the Training Dataset and Upload It to Amazon S3 (p. 63)

**Step 4.1: Download the MNIST Dataset**

To download the MNIST dataset, copy and paste the following code into the notebook and run it:

```python
import boto3
import pickle
# download preprocessed data from S3
s3 = boto3.client('s3')
bucket = 'sagemaker-sample-files'
s3.download_file(bucket, 'datasets/image/MNIST/mnist.pkl', './mnist.pkl')
with open('mnist.pkl', 'rb') as f:
    train_set, valid_set, test_set = pickle.load(f)
```

The code does the following:

1. Downloads the MNIST dataset (mnist.pkl) from the SageMaker public database S3 bucket to your notebook instance.
2. Unzips the file and reads the following datasets into the notebook instance's memory:
   • train_set – You use these images of handwritten numbers to train a model.
   • valid_set – The XGBoost Algorithm (p. 813) uses these images to evaluate the progress of the model during training.
   • test_set – You use this set to get inferences to test the deployed model.

Next Step

Step 4.2: Explore the Training Dataset (p. 62)

Step 4.2: Explore the Training Dataset

Typically, you explore training data to determine what you need to clean up and which transformations to apply to improve model training. For this exercise, you don't need to clean up the MNIST dataset.

To explore the dataset

• Type the following code in a cell in your notebook and run the cell to display the first 10 images in train_set:

```python
%matplotlib inline
import matplotlib.pyplot as plt
plt.rcParams['figure.figsize'] = (2,10)
for i in range(0, 10):
    img = train_set[0][i]
    label = train_set[1][i]
    img_reshape = img.reshape((28,28))
    imgplot = plt.imshow(img_reshape, cmap='gray')
    print('This is a {}'.format(label))
plt.show()
```

train_set contains the following structures:

• train_set[0] – Contains images.
• train_set[1] – Contains labels.

The code uses the matplotlib library to get and display the first 10 images from the training dataset.
Step 4.3: Transform the Training Dataset and Upload It to Amazon S3 (p. 63)

The XGBoost Algorithm (p. 813) expects comma-separated values (CSV) for its training input. The format of the training dataset is numpy.array. Transform the dataset from numpy.array format to the CSV format. Then upload it to the Amazon S3 bucket that you created in Step 1: Create an Amazon S3 Bucket (p. 59)

To convert the dataset to CSV format and upload it

Type the following code into a cell in your notebook and then run the cell.

```python
%%time
import os
import re
import copy
import time
import io
import struct
from time import gmtime, strftime
import boto3
import sagemaker

role = sagemaker.get_execution_role()
region = boto3.Session().region_name
bucket = sagemaker.Session().default_bucket()  # Default S3 URI. Replace it with your S3 URI if you want.
prefix = 'sagemaker/xgboost-mnist'  # Used as part of the path in the bucket where you store data

def convert_data():
    data_partitions = [('train', train_set), ('validation', valid_set), ('test', test_set)]
    for data_partition_name, data_partition in data_partitions:
        print('{}: {} {}'.format(data_partition_name, data_partition[0].shape, data_partition[1].shape))
        labels = [t.tolist() for t in data_partition[1]]
        features = [t.tolist() for t in data_partition[0]]
        if data_partition_name != 'test':
            examples = np.insert(features, 0, labels, axis=1)
        else:
            examples = features
        np.savetxt('data.csv', examples, delimiter=',

key = '{}/{}examples'.format(prefix, data_partition_name)
url = 's3://{}{}'.format(bucket, key)
boto3.Session().resource('s3').Bucket(bucket).Object(key).upload_file('data.csv')
print('Done writing to {}'.format(url))
```
Step 5: Train a Model

To train, deploy, and validate a model in Amazon SageMaker, you can use either the Amazon SageMaker Python SDK or the AWS SDK for Python (Boto3). (You can also use the console, but for this exercise, you will use the notebook instance and one of the SDKs.) This exercise provides code examples for each library.

The Amazon SageMaker Python SDK abstracts several implementation details, and is easy to use. If you're a first-time SageMaker user, we recommend that you use it to train, deploy, and validate the model. For more information, see https://sagemaker.readthedocs.io/en/stable/overview.html.

Choose the Training Algorithm

To choose the right algorithm for your model, you typically follow an evaluation process. In this tutorial, you use the SageMaker XGBoost Algorithm (p. 813), so no evaluation is required. For more information about choosing a SageMaker algorithms, see Use Amazon SageMaker Built-in Algorithms (p. 635).

Create and Run a Training Job (Amazon SageMaker Python SDK)

The Amazon SageMaker Python SDK includes the sagemaker.estimator.Estimator estimator. You can use this class, in the sagemaker.estimator module, with any algorithm. For more information about available built-in algorithms, see https://sagemaker.readthedocs.io/en/stable/estimators.html#sagemaker.estimator.Estimator.

To run a model training job (Amazon SageMaker Python SDK)

1. Import the Amazon SageMaker Python SDK and get the XGBoost container.

```python
import sagemaker
from sagemaker import image_uris
container = sagemaker.image_uris.retrieve("xgboost", region, "1.2-1")
```

Note

Check the SageMaker Python SDK version by running sagemaker.__version__. This tutorial is based on sagemaker>=2.15.2. Install the latest version by running the following command:

```
! pip install -qU sagemaker>=2.15.2
```
Step 5: Train a Model

If you run this installation in your exiting SageMaker Studio or Notebook instance JupyterLab environment, you need to manually refresh the kernel to finish applying the version update.

2. Download the training and validation data from the Amazon S3 location where you uploaded it in Step 4.3: Transform the Training Dataset and Upload It to Amazon S3 (p. 63), and set the location where you store the training output.

```python
train_data = 's3:///{}/{}' .format(bucket, prefix, 'train')
validation_data = 's3:///{}/{}' .format(bucket, prefix, 'validation')
s3_output_location = 's3:///{}/{}' .format(bucket, prefix, 'xgboost_model_sdk')
print(train_data)
```

3. Create an instance of the `sagemaker.estimator.Estimator` class.

```python
xgb_model = sagemaker.estimator.Estimator(container,
    role,
    instance_count=1,
    instance_type='ml.m4.xlarge',
    volume_size = 5,
    output_path=s3_output_location,
    sagemaker_session=sagemaker.Session())
```

In the constructor, you specify the following parameters:

- `role` – The AWS Identity and Access Management (IAM) role that SageMaker can assume to perform tasks on your behalf (for example, reading training results, called model artifacts, from the S3 bucket and writing training results to Amazon S3). This is the role that you got in Step 3: Create a Jupyter Notebook (p. 60).
- `train_instance_count` and `train_instance_type` – The type and number of ML compute instances to use for model training. For this exercise, you use only a single training instance.
- `train_volume_size` – The size, in GB, of the Amazon Elastic Block Store (Amazon EBS) storage volume to attach to the training instance. This must be large enough to store training data if you use File mode (File mode is the default).
- `output_path` – The path to the S3 bucket where SageMaker stores the training results.
- `sagemaker_session` – The session object that manages interactions with SageMaker APIs and any other AWS service that the training job uses.

4. Set the hyperparameter values for the XGBoost training job by calling the `set_hyperparameters` method of the estimator. For a description of XGBoost hyperparameters, see XGBoost Hyperparameters (p. 820).

```python
xgb_model.set_hyperparameters(max_depth = 5,
    eta = .2,
    gamma = 4,
    min_child_weight = 6,
    silent = 0,
    objective = "multi:softmax",
    num_class = 10,
    num_round = 10)
```

5. Create the training channels to use for the training job. For this example, we use both `train` and `validation` channels.

```python
from sagemaker.inputs import TrainingInput
train_channel = TrainingInput(train_data, content_type='text/csv')
```
valid_channel = TrainingInput(validation_data, content_type='text/csv')
data_channels = {'train': train_channel, 'validation': valid_channel}

6. To start model training, call the estimator's `fit` method.

   ```python
   xgb_model.fit(inputs=data_channels, logs=True)
   ```

   This is a synchronous operation. The method displays progress logs and waits until training completes before returning. For more information about model training, see Train a Model with Amazon SageMaker (p. 8).

   Model training for this exercise can take up to 15 minutes.

Next Step

Step 6: Deploy the Model to Amazon SageMaker (p. 68)

Create and Run a Training Job (AWS SDK for Python (Boto3))

To train a model, SageMaker uses the `CreateTrainingJob` API. The AWS SDK for Python (Boto3) provides the corresponding create_training_job method.

When using this method, you provide the following information:

- The training algorithm – Specify the registry path of the Docker image that contains the training code. For the registry paths for the algorithms provided by SageMaker, see Docker Registry Paths for SageMaker Built-in Algorithms (p. 640).
- Algorithm-specific hyperparameters – Specify algorithm-specific hyperparameters to influence the final quality of the model. For information, see XGBoost Hyperparameters (p. 820).
- The input and output configuration – Provide the S3 bucket where training data is stored and where SageMaker saves the results of model training (the model artifacts).

To run a model training job (AWS SDK for Python (Boto3))

1. Import the `get_image_url` utility function Amazon SageMaker Python SDK and get the location of the XGBoost container.

   ```python
   import sagemaker
   from sagemaker import image_uris
   container = sagemaker.image_uris.retrieve("xgboost", region, "1.2-1")
   ```

2. Set up the training information for the job. You pass this information when you call create_training_job. For more information about the information that you need to send to a training job, see CreateTrainingJob.

   ```python
   # Ensure that the train and validation data folders generated above are reflected in the "InputDataConfig" parameter below.
   common_training_params = 
   {   "AlgorithmSpecification": {   "TrainingImage": container,   "TrainingInputMode": "File"   },
   ```
"RoleArn": role,
"OutputDataConfig": {
  "S3OutputPath": bucket_path + "+ prefix + "/xgboost"
},
"ResourceConfig": {
  "InstanceCount": 1,
  "InstanceType": "ml.m4.xlarge",
  "VolumeSizeInGB": 5
},
"HyperParameters": {
  "max_depth": "5",
  "eta": "0.2",
  "gamma": "4",
  "min_child_weight": "6",
  "silent": "0",
  "objective": "multi:softmax",
  "num_class": "10",
  "num_round": "10"
},
"StoppingCondition": {
  "MaxRuntimeInSeconds": 86400
},
"InputDataConfig": [
  {
    "ChannelName": "train",
    "DataSource": {
      "S3DataSource": {
        "S3DataType": "S3Prefix",
        "S3Uri": bucket_path + "+ prefix+ '/train/',
        "S3DataDistributionType": "FullyReplicated"
      }
    },
    "ContentType": "text/csv",
    "CompressionType": "None"
  },
  {
    "ChannelName": "validation",
    "DataSource": {
      "S3DataSource": {
        "S3DataType": "S3Prefix",
        "S3Uri": bucket_path + "+ prefix+ '/validation/',
        "S3DataDistributionType": "FullyReplicated"
      }
    },
    "ContentType": "text/csv",
    "CompressionType": "None"
  }
]

3. Name your training job, and finish configuring the parameters that you send to it.

```python
# training job params
training_job_name = 'xgboost-mnist' + strftime("%Y-%m-%d-%H-%M-%S", gmttime())
print("Job name is: ", training_job_name)

training_job_params = copy.deepcopy(common_training_params)
training_job_params['TrainingJobName'] = training_job_name
training_job_params['ResourceConfig']['InstanceCount'] = 1
```

4. Call `create_training_job` to start the training job, and wait for it to complete. If the training job
fails, print the reason that it failed.

```python
%%time
```
You now have a trained model. SageMaker stores the resulting artifacts in your S3 bucket.

Next Step

Step 6: Deploy the Model to Amazon SageMaker (p. 68)

Step 6: Deploy the Model to Amazon SageMaker

To get predictions, deploy your model to Amazon SageMaker. The method you use depends on how you want to generate inferences:

- To get one inference at a time in real time, set up a persistent endpoint using SageMaker hosting services.
- To get inferences for an entire dataset, use SageMaker batch transform.

Topics

- Step 6.1: Deploy the Model to SageMaker Hosting Services (p. 68)
- Step 6.2: Deploy the Model with Batch Transform (p. 70)

Step 6.1: Deploy the Model to SageMaker Hosting Services

To deploy a model in SageMaker, hosting services, you can use either the Amazon SageMaker Python SDK or the AWS SDK for Python (Boto3). This exercise provides code examples for both libraries.

The Amazon SageMaker Python SDK abstracts several implementation details, and is easy to use. If you're a first-time SageMaker user, we recommend that you use it. For more information, see https://sagemaker.readthedocs.io/en/stable/overview.html.

Topics

- Deploy the Model to SageMaker Hosting Services (Amazon SageMaker Python SDK) (p. 69)
- Deploy the Model to SageMaker Hosting Services (AWS SDK for Python (Boto3)) (p. 69)
Deploy the Model to SageMaker Hosting Services (Amazon SageMaker Python SDK)

Deploy the model that you trained in Create and Run a Training Job (Amazon SageMaker Python SDK) (p. 64) by calling the deploy method of the sagemaker.estimator.Estimator object. This is the same object that you used to train the model. When you call the deploy method, specify the number and type of ML instances that you want to use to host the endpoint.

```python
xgb_predictor = xgb_model.deploy(initial_instance_count=1,
                                  content_type='text/csv',
                                  instance_type='ml.t2.medium')
```

The deploy method creates the deployable model, configures the SageMaker hosting services endpoint, and launches the endpoint to host the model. For more information, see https://sagemaker.readthedocs.io/en/stable/estimators.html#sagemaker.estimator.Estimator.deploy.

It also returns a sagemaker.predictor.RealTimePredictor object, which you can use to get inferences from the model. For information, see https://sagemaker.readthedocs.io/en/stable/predictors.html#sagemaker.predictor.RealTimePredictor.

Next Step

Step 7: Validate the Model (p. 72)

Deploy the Model to SageMaker Hosting Services (AWS SDK for Python (Boto3)).

Deploying a model using the AWS SDK for Python (Boto 3) is a three-step process:

1. Create a model in SageMaker – Send a CreateModel request to provide information such as the location of the S3 bucket that contains your model artifacts and the registry path of the image that contains inference code.

2. Create an endpoint configuration – Send a CreateEndpointConfig request to provide the resource configuration for hosting. This includes the type and number of ML compute instances to launch to deploy the model.

3. Create an endpoint – Send a CreateEndpoint request to create an endpoint. SageMaker launches the ML compute instances and deploys the model. SageMaker returns an endpoint. Applications can send requests for inference to this endpoint.

To deploy the model (AWS SDK for Python (Boto 3))

For each of the following steps, paste the code in a cell in the Jupyter notebook you created in Step 3: Create a Jupyter Notebook (p. 60) and run the cell.

1. Create a deployable model by identifying the location of model artifacts and the Docker image that contains the inference code.

```python
model_name = training_job_name + '-mod'

info = sm.describe_training_job(TrainingJobName=training_job_name)
model_data = info['ModelArtifacts'] ['S3ModelArtifacts']
print(model_data)

primary_container = {
    'Image': container,
    'ModelDataUrl': model_data
}
```
Step 6: Deploy the Model

2. Create a SageMaker endpoint configuration by specifying the ML compute instances that you want to deploy your model to.

```python
epsilon_point_config_name = 'DEMO-XGBoostEndpointConfig-' + strftime("%Y-%m-%d-%H-%M-%S", gmtime())
print(endpoint_config_name)
create_endpoint_config_response = sm.create_endpoint_config(EndpointConfigName = endpoint_config_name,
ProductionVariants=[
    {'InstanceType': 'ml.m4.xlarge',
    'InitialVariantWeight': 1,
    'InitialInstanceCount': 1,
    'ModelName': model_name,
    'VariantName': 'AllTraffic'}])
print("Endpoint Config Arn: " + create_endpoint_config_response["EndpointConfigArn"])
```

3. Create a SageMaker endpoint.

```python
%%time
import time
endpoint_name = 'DEMO-XGBoostEndpoint-' + strftime("%Y-%m-%d-%H-%M-%S", gmtime())
print(endpoint_name)
create_endpoint_response = sm.create_endpoint(EndpointName=endpoint_name,
EndpointConfigName=endpoint_config_name)
print(create_endpoint_response["EndpointArn"])
resp = sm.describe_endpoint(EndpointName=endpoint_name)
status = resp["EndpointStatus"]
print("Status: " + status)
while status=='Creating':
    time.sleep(60)
    resp = sm.describe_endpoint(EndpointName=endpoint_name)
    status = resp["EndpointStatus"]
    print("Status: " + status)
print("Arn: " + resp["EndpointArn"])
print("Status: " + status)
```

This code continuously calls the `describe_endpoint` command in a while loop until the endpoint either fails or is in service, and then prints the status of the endpoint. When the status changes to `InService`, the endpoint is ready to serve inference requests.

Next Step

Step 7: Validate the Model (p. 72)

**Step 6.2: Deploy the Model with Batch Transform**

To get inference for an entire dataset, use batch transform. SageMaker stores the results in Amazon S3.
For information about batch transforms, see Get Inferences for an Entire Dataset with Batch Transform (p. 14). For an example that uses batch transform, see the batch transform sample notebook at https://github.com/awslabs/amazon-sagemaker-examples/tree/master/sagemaker_batch_transform/introduction_to_batch_transform.

Topics
- Deploy a Model with Batch Transform (SageMaker High-level Python Library) (p. 71)
- Deploy a Model with Batch Transform (SDK for Python (Boto3)) (p. 71)

Deploy a Model with Batch Transform (SageMaker High-level Python Library)

The following code creates a sagemaker.transformer.Transformer object from the model that you trained in Create and Run a Training Job (Amazon SageMaker Python SDK) (p. 64). Then it calls that object's transform method to create a transform job. When you create the sagemaker.transformer.Transformer object, you specify the number and type of ML instances to use to perform the batch transform job, and the location in Amazon S3 where you want to store the inferences.

Paste the following code in a cell in the Jupyter notebook you created in Step 3: Create a Jupyter Notebook (p. 60) and run the cell.

```python
# The location of the test dataset
batch_input = 's3://{}/{}/test/examples'.format(bucket, prefix)

# The location to store the results of the batch transform job
batch_output = 's3://{}/{}/batch-inference'.format(bucket, prefix)

transformer = xgb_model.transformer(instance_count=1, instance_type='ml.m4.xlarge',
                                     output_path=batch_output)

transformer.transform(data=batch_input, data_type='S3Prefix', content_type='text/csv',
                      split_type='Line')

transformer.wait()
```


Next Step

Step 7: Validate the Model (p. 72)

Deploy a Model with Batch Transform (SDK for Python (Boto3))

To run a batch transform job, call the `create_transform_job` method using the model that you trained in Deploy the Model to SageMaker Hosting Services (AWS SDK for Python (Boto3)). (p. 69).

To create a batch transform job (SDK for Python (Boto3))

For each of the following steps, paste the code in a cell in the Jupyter notebook you created in Step 3: Create a Jupyter Notebook (p. 60) and run the cell.

1. Name the batch transform job and specify where the input data (the test dataset) is stored and where to store the job's output.

```python
batch_job_name = 'xgboost-mnist-batch' + strftime("%Y-%m-%d-%H-%M-%S", gmtime())

batch_input = 's3://{}/{}/test/examples'.format(bucket, prefix)
print(batch_input)
```
2. Configure the parameters that you pass when you call the `create_transform_job` method.

```python
request = {
    "TransformJobName": batch_job_name,
    "ModelName": model_name,
    "BatchStrategy": "MultiRecord",
    "TransformOutput": {
        "S3OutputPath": batch_output
    },
    "TransformInput": {
        "DataSource": {
            "S3DataSource": {
                "S3DataType": "S3Prefix",
                "S3Uri": batch_input
            }
        },
        "ContentType": "text/csv",
        "SplitType": "Line",
        "CompressionType": "None"
    },
    "TransformResources": {
        "InstanceType": "ml.m4.xlarge",
        "InstanceCount": 1
    }
}
```

For more information about the parameters, see CreateTransformJob.

3. Call the `create_transform_job` method, passing in the parameters that you configured in the previous step. Then call the `describe_transform_job` method in a loop until it completes.

Paste the following code in a cell in the Jupyter notebook you created in Step 3: Create a Jupyter Notebook (p. 60) and run the cell.

```python
sm.create_transform_job(**request)
while(True):
    response = sm.describe_transform_job(TransformJobName=batch_job_name)
    status = response['TransformJobStatus']
    if status == 'Completed':
        print("Transform job ended with status: " + status)
        break
    if status == 'Failed':
        message = response['FailureReason']
        print('Transform failed with the following error: {}'.format(message))
        raise Exception('Transform job failed')
    print("Transform job is still in status: " + status)
    time.sleep(30)

Next Step

Step 7: Validate the Model (p. 72)

Step 7: Validate the Model

Now that you have trained and deployed a model in Amazon SageMaker, validate it to ensure that it generates accurate predictions on new data. That is, on data that is different from the data that the
model was trained on. For this, use the test dataset that you created in Step 4: Download, Explore, and Transform the Training Data (p. 61).

Topics
- Step 7.1: Validate a Model Deployed to SageMaker Hosting Services (p. 73)
- Step 7.2: Validate a Model Deployed with Batch Transform (p. 75)

Step 7.1: Validate a Model Deployed to SageMaker Hosting Services

If you deployed a model to Amazon SageMaker hosting services in Step 6.1: Deploy the Model to SageMaker Hosting Services (p. 68), you now have an endpoint that you can invoke to get inferences in real time. To validate the model, invoke the endpoint with example images from the test dataset and check whether the inferences you get match the actual labels of the images.

Topics
- Validate a Model Deployed to SageMaker Hosting Services (SageMaker Python SDK) (p. 73)
- Validate a Model Deployed to SageMaker Hosting Services (AWS SDK for Python (Boto3)) (p. 74)

Validate a Model Deployed to SageMaker Hosting Services (SageMaker Python SDK)

To validate the model by using the Amazon SageMaker Python SDK, use the sagemaker.predictor.RealTimePredictor object that you created in Deploy the Model to SageMaker Hosting Services (Amazon SageMaker Python SDK) (p. 69). For information, see https://sagemaker.readthedocs.io/en/stable/predictors.html#sagemaker.predictor.RealTimePredictor.

To validate the model (Amazon SageMaker Python SDK)

1. Download the test data from Amazon S3.

   ```python
   s3 = boto3.resource('s3')
   test_key = "/{}/test/examples".format(prefix)
   s3.Bucket(bucket).download_file(test_key, 'test_data')
   ```

2. Plot the first 10 images from the test dataset with their labels.

   ```python
   %matplotlib inline
   for i in range (0, 10):
       img = test_set[0][i]
       label = test_set[1][i]
       img_reshape = img.reshape((28,28))
       imgplot = plt.imshow(img_reshape, cmap='gray')
       print('This is a {}'.format(label))
       plt.show()
   ```
3. To get inferences for the first 10 examples in the test dataset, call the predict method of the sagemaker.predictor.RealTimePredictor object.

```python
with open('test_data', 'r') as f:
    for j in range(0, 10):
        single_test = f.readline()
        result = xgb_predictor.predict(single_test)
        print(result)
```

To see if the model is making accurate predictions, check the output from this step against the numbers that you plotted in the previous step.

You have now trained, deployed, and validated your first model in SageMaker.

Next Step

Step 9: Clean Up (p. 77)

Validate a Model Deployed to SageMaker Hosting Services (AWS SDK for Python (Boto3))

To use the AWS SDK for Python (Boto3) to validate the model, call the invoke_endpoint method. This method corresponds to the InvokeEndpoint API provided by the SageMaker runtime.

To validate the model (AWS SDK for Python (Boto3))

1. Download the test data from Amazon S3.

```python
s3 = boto3.resource('s3')
test_key = "/{}/test/examples".format(prefix)
s3.Bucket(bucket).download_file(test_key, 'test_data')
```

2. Plot the first 10 images from the test dataset with their labels.

```python
%matplotlib inline
for i in range (0, 10):
```
Step 7: Validate the Model

```python
img = test_set[0][i]
label = test_set[1][i]
img reshape = img.reshape((28, 28))
imgplot = plt.imshow(img reshape, cmap='gray')
print('This is a {}'.format(label))
plt.show()
```

3. Get the SageMaker runtime client, which provides the `invoke_endpoint` method.

```python
runtime_client = boto3.client('runtime.sagemaker')
```

4. Get inferences from the first 10 examples in the test dataset by calling `invoke_endpoint`.

```python
with open('test_data', 'r') as f:
    for i in range(0, 10):
        single_test = f.readline()
        response = runtime_client.invoke_endpoint(EndpointName = endpoint_name,
                                                 ContentType = 'text/csv',
                                                 Body = single_test)
        result = response['Body'].read().decode('ascii')
        print('Predicted label is {}'.format(result))
```

5. To see if the model is making accurate predictions, check the output from this step against the numbers you plotted in the previous step.

You have now trained, deployed, and validated your first model in SageMaker.

**Next Step**

Step 7.2: Validate a Model Deployed with Batch Transform

You now have a file in Amazon S3 that contains inferences that you got by running a batch transform job in Step 6.2: Deploy the Model with Batch Transform (p. 70). To validate the model, check a subset of the inferences from the file to see whether they match the actual numbers from the test dataset.

**To validate the batch transform inferences**

1. Download the test data from Amazon S3.
2. Plot the first 10 images from the test dataset with their labels.

```python
#matplotlib inline
for i in range (0, 10):
    img = test_set[0][i]
    label = test_set[1][i]
    img_reshape = img.reshape((28,28))
    imgplot = plt.imshow(img_reshape, cmap='gray')
    print('This is a {}'.format(label))
    plt.show()
```

3. Download the output from the batch transform job from Amazon S3 to a local file.

```python
s3.Bucket(bucket).download_file(prefix + '/batch-inference/examples.out', 'batch_results')
```

4. Get the first 10 results from the batch transform job.

```python
with open('batch_results') as f:
    results = f.readlines()
for j in range (0, 10):
    print(results[j])
```

5. To see if the batch transform job made accurate predictions, check the output from this step against the numbers that you plotted from the test data.

You have now trained, deployed, and validated your first model in SageMaker.

**Next Step**

**Step 8: Integrating Amazon SageMaker Endpoints into Internet-facing Applications (p. 77)**
Step 8: Integrating Amazon SageMaker Endpoints into Internet-facing Applications

In a production environment, you might have an internet-facing application sending requests to the endpoint for inference. The following high-level example shows how to integrate your model endpoint into your application.

For an example of how to use Amazon API Gateway and AWS Lambda to set up and deploy a web service that you can call from a client application, see Call a SageMaker model endpoint using Amazon API Gateway and AWS Lambda in the AWS Machine Learning Blog.

1. Create an IAM role that the AWS Lambda service principal can assume. Give the role permissions to call the Amazon SageMaker InvokeEndpoint API.
2. Create a Lambda function that calls the SageMaker InvokeEndpoint API.
3. Call the Lambda function from a mobile application. For an example of how to call a Lambda function from a mobile application using Amazon Cognito for credentials, see Tutorial: Using AWS Lambda as Mobile Application Backend.

Next Step

Step 9: Clean Up (p. 77)

Step 9: Clean Up

To avoid incurring unnecessary charges, use the AWS Management Console to delete the resources that you created for this exercise.

Note
If you plan to explore other exercises in this guide, you might want to keep some of these resources, such as your notebook instance, S3 bucket, and IAM role.

1. Open the Amazon SageMaker console at https://console.aws.amazon.com/sagemaker/ and delete the following resources:
   • The endpoint. Deleting the endpoint also deletes the ML compute instance or instances that support it.
     1. Under Inference, choose Endpoints.
     2. Choose the endpoint that you created in the example, then choose Actions | Delete.
     3. Choose Delete.
   • The endpoint configuration.
     1. Under Inference, choose Endpoint configurations.
     2. Choose the endpoint configuration that you created in the example, then choose Actions | Delete.
     3. Choose Delete.
   • The model.
     1. Under Inference, choose Models.
     2. Choose the model that you created in the example, then choose Actions | Delete.
     3. Choose Delete.
   • The notebook instance. Before deleting the notebook instance, stop it.
     1. Under Notebook, choose Notebook instances.
2. Choose the notebook instance that you created in the example, then choose **Actions | Stop**. The notebook instance takes up to several minutes to stop. When **Status** changes to **Stopped**, move on to the next step.

3. Choose **Actions | Delete**.

4. Choose **Delete**.

2. Open the Amazon S3 console at [https://console.aws.amazon.com/s3/](https://console.aws.amazon.com/s3/) and delete the bucket that you created for storing model artifacts and the training dataset.

3. Open the Amazon CloudWatch console at [https://console.aws.amazon.com/cloudwatch/](https://console.aws.amazon.com/cloudwatch/) and delete all of the log groups that have names starting with `/aws/sagemaker/`.
Amazon SageMaker Studio

Amazon SageMaker Studio is a web-based, integrated development environment (IDE) for machine learning that lets you build, train, debug, deploy, and monitor your machine learning models. SageMaker Studio provides all the tools you need to take your models from experimentation to production while boosting your productivity. In a single unified visual interface, customers can perform the following tasks:

- Write and execute code in Jupyter notebooks
- Build and train machine learning models
- Deploy the models and monitor the performance of their predictions
- Track and debug the machine learning experiments

For information on the onboarding steps to sign in to SageMaker Studio, see Onboard to Amazon SageMaker Studio (p. 34).

For the AWS Regions supported by SageMaker Studio, see Supported Regions and Quotas (p. 32).

Topics

- Amazon SageMaker Studio UI Overview (p. 79)
- Use the Amazon SageMaker Studio Launcher (p. 82)
- Use Amazon SageMaker Studio Notebooks (p. 84)
- Bring your own SageMaker image (p. 102)
- Set Up a Connection to an Amazon EMR Cluster (p. 116)
- Perform Common Tasks in Amazon SageMaker Studio (p. 119)
- Amazon SageMaker Studio Pricing (p. 122)

Amazon SageMaker Studio UI Overview

Amazon SageMaker Studio extends the JupyterLab interface. Previous users of JupyterLab will notice the similarity of the user interface, including the workspace. Studio adds many additions to the interface. The most prominent additions are detailed in the following sections. For an overview of the basic JupyterLab interface, see The JupyterLab Interface.

The following image shows SageMaker Studio with the file browser open and the Studio Launcher displayed.
At the top of the screen is the **menu bar**. At the left of the screen is the **left sidebar** which contains icons to open file browsers, resource browsers, and tools. At the right of the screen is the **right sidebar**, represented by the **Settings** icon (⚙), which displays contextual property settings when open. At the bottom of the screen is the **status bar**.

Above the **Settings** icon, there's a button to provide feedback about your experiences with SageMaker Studio. To the left of the **Feedback** button there's the notification icon. Choose the icon to view notifications from Studio such as new Studio versions and new SageMaker features. To update to a new version of Studio, see Update SageMaker Studio and Studio Apps (p. 121).

The main work area is divided horizontally into two panes. The left pane is the **file and resource browser**. The right pane contains one or more tabs for resources such as notebooks, terminals, metrics, and graphs.

**Topics**

- Left sidebar (p. 80)
- File and resource browser (p. 81)
- Main work area (p. 82)
- Settings (p. 82)

**Left sidebar**

The left sidebar includes the following icons. When you hover over an icon, a tooltip displays the icon name. When you choose an icon, the file and resource browser displays the described functionality. For hierarchical entries, a selectable breadcrumb at the top of the browser shows your location in the hierarchy.
### Icon Description

- **File Browser**
  - Choose the **Upload Files** icon ( Upload ) to add files to Studio.
  - Double-click a file to open the file in a new tab.
  - To have adjacent files open, choose a tab that contains a notebook, Python, or text file, then choose **New View for File**.
  - Choose the plus (+) sign on the menu at the top of the file browser to open the Studio Launcher.

- **Git**
  - You can connect to a Git repository and then access a full range of Git tools and operations. For more information, see Clone a Git Repository in SageMaker Studio (p. 120).

- **Running Terminals and Kernels**
  - For more information, see Shut Down Resources (p. 98).

- **Commands (Ctrl + Shift + C)**
  - The majority of the menu commands are available here.

- **Notebook Tools**
  - You can access a notebook's metadata through the **Advanced Tools** section. This icon is displayed only when a notebook is open.

- **SageMaker Jumpstart**
  - Provides a list of solutions, model endpoints, or training jobs created with SageMaker Jumpstart.

- **Open Tabs**
  - Provides a list of open tabs, which is useful if you have multiple open tabs.

- **SageMaker Components and registries**
  - Provides a list of projects, data wrangler flows, pipelines, experiments, trials, models, or endpoints, or access to the feature store.

### File and resource browser

The file and resource browser displays lists of your notebooks, experiments, trials, trial components, and endpoints. On the menu at the top of the file browser, choose the plus (+) sign to open the Studio Launcher. The Launcher allows you to create a notebook, launch a Python interactive shell, or open a terminal.
Main work area

The main work area consists of multiple tabs that contain your open notebooks and terminals, and detailed information about your experiments and endpoints. One commonly used tab is the Trial Component List. This list is referred to as the Leaderboard because it’s where you can compare experiments and trials. For more information, see View and Compare Amazon SageMaker Experiments, Trials, and Trial Components (p. 850).

Settings

The settings pane allows you to adjust table and chart properties. By default, the pane is hidden on the far right of the screen. To open the pane, choose the Settings icon on the top right of the screen.

Use the Amazon SageMaker Studio Launcher

You can use the Amazon SageMaker Studio Launcher to create notebooks and text files, and launch terminals and interactive Python shells.

You can open Studio Launcher in any of the following ways:

- Choose Amazon SageMaker Studio at the top-left of Studio.
- Use the keyboard shortcut Ctrl + Shift + L.
- From the Studio menu, choose File and then choose New Launcher.
- If the Studio file browser is open, choose the plus (+) sign on the Studio file browser menu.

The Launcher opens in a new tab in Studio. Your screen should look similar to the following:
The Launcher consists of following sections:

- **Get started** – Provides material to get started using SageMaker Studio, such as videos and tutorials, and one-click solutions for machine learning problems.
- **ML tasks and components** – Create machine learning tasks and components, such as new feature groups, data flows, and projects.
- **Notebooks and compute resources** – Create a notebook, open an image terminal, or open a Python console.
- **Utilities and files** – Show contextual help from a notebook, create files, or open a system terminal.

### Topics

- Notebooks and compute resources (p. 83)
- Utilities and files (p. 83)

### Notebooks and compute resources

To create or launch an item, choose the SageMaker image that you want the item to run in from the **SageMaker image** dropdown menu. Next, choose the item. When you choose an item from this section, you might incur additional usage charges. For more information, see **Usage Metering** (p. 99).

The following items are available:

- **Notebook**
  
  Launches the notebook in a kernel session on the chosen SageMaker image. For more information, see **Change a Kernel** (p. 95).

  Creates the notebook in the folder that you have currently selected in the file browser. To view the file browser, in the left sidebar of Studio, choose the **File Browser** icon ( 文件夹).

- **Console**
  
  Launches the shell in a kernel session on the chosen SageMaker image.

  Opens the shell in the folder that you have currently selected in the file browser.

- **Image terminal**
  
  Launches the terminal in a terminal session on the chosen SageMaker image.

  Opens the terminal in the root folder for the user (as shown by the Home folder in the file browser).

### Utilities and files

Items in this section run in the context of SageMaker Studio and don't incur usage charges.

The following items are available:

- **Show Contextual Help**
  
  Opens a new tab that displays contextual help for functions in a Studio notebook. To display the help, choose a function in an active notebook. To make it easier to see the help in context, drag the help tab so that it's adjacent to the notebook tab. To open the help tab from within a notebook, press `Ctrl + I`.

  The following screenshot shows the contextual help for the `Experiment.create` method.
Use Amazon SageMaker Studio Notebooks

Amazon SageMaker Studio notebooks are collaborative notebooks that you can launch quickly because you don’t need to set up compute instances and file storage beforehand. A set of instance types, known as Fast launch types are designed to launch in under two minutes. SageMaker Studio notebooks provide persistent storage, which enables you to view and share notebooks even if the instances that the notebooks run on are shut down.

You can share your notebooks with others, so that they can easily reproduce your results and collaborate while building models and exploring your data. You provide access to a read-only copy of the notebook through a secure URL. Dependencies for your notebook are included in the notebook's metadata. When your colleagues copy the notebook, it opens in the same environment as the original notebook.

A SageMaker Studio notebook runs in an environment defined by the following:

- **EC2 instance type** – The hardware configuration the notebook runs on. The configuration includes the number and type of processors (vCPU and GPU), and the amount and type of memory. The instance type determines the pricing rate.

**• System terminal**

Opens a bash shell in the root folder for the user (as shown by the Home folder in the file browser).

**• Text File and Markdown File**

Creates a file of the associated type in the folder that you have currently selected in the file browser.

To view the file browser, in the left sidebar, choose the **File Browser** icon ().

Use Studio Notebooks

- System terminal
  - Opens a bash shell in the root folder for the user (as shown by the Home folder in the file browser).
- Text File and Markdown File
  - Creates a file of the associated type in the folder that you have currently selected in the file browser.

To view the file browser, in the left sidebar, choose the **File Browser** icon ( ).
• SageMaker image – A container image that is compatible with SageMaker Studio. The image consists of the kernels, language packages, and other files required to run a notebook in Studio. There can be multiple images in an instance. For more information, see Bring your own SageMaker image (p. 102).
• KernelGateway app – A SageMaker image runs as a KernelGateway app. The app provides access to the kernels in the image. There is a one-to-one correspondence between a SageMaker image and a SageMaker app.
• Kernel – The process that inspects and runs the code contained in the notebook. A kernel is defined by a kernel spec in the image. There can be multiple kernels in an image.

You can change any of these resources from within the notebook.

Sample SageMaker Studio notebooks are available in the aws_sagemaker_studio folder of the Amazon SageMaker example GitHub repository. Each notebook comes with the necessary SageMaker image that opens the notebook with the appropriate kernel.

We recommend that you familiarize yourself with the SageMaker Studio interface and the Studio notebook toolbar before creating or using a Studio notebook. For more information, see Amazon SageMaker Studio UI Overview (p. 79) and Use the SageMaker Studio Notebook Toolbar (p. 88).

Topics
• How Are Amazon SageMaker Studio Notebooks Different from Notebook Instances? (p. 85)
• Get Started (p. 86)
• Create or Open an Amazon SageMaker Studio Notebook (p. 87)
• Use the SageMaker Studio Notebook Toolbar (p. 88)
• Share and Use a Amazon SageMaker Studio Notebook (p. 90)
• Get Notebook and App Metadata (p. 92)
• Get Notebook Differences (p. 93)
• Manage Resources (p. 94)
• Usage Metering (p. 99)
• Available Resources (p. 99)

How Are Amazon SageMaker Studio Notebooks Different from Notebook Instances?

When you're starting a new notebook, we recommend that you create the notebook in Amazon SageMaker Studio instead of launching a notebook instance from the Amazon SageMaker console. There are many benefits to using a SageMaker Studio notebook, including the following:

• Starting a Studio notebook is faster than launching an instance-based notebook. Typically, it is 5-10 times faster than instance-based notebooks.
• Notebook sharing is an integrated feature in SageMaker Studio. Users can generate a shareable link that reproduces the notebook code and also the SageMaker image required to execute it, in just a few clicks.
• SageMaker Studio notebooks come pre-installed with the latest Amazon SageMaker Python SDK.
• SageMaker Studio notebooks are accessed from within Studio. This enables you to build, train, debug, track, and monitor your models without leaving Studio.
• Each member of a Studio team gets their own home directory to store their notebooks and other files. The directory is automatically mounted onto all instances and kernels as they're started, so their notebooks and other files are always available. The home directories are stored in Amazon Elastic File System (Amazon EFS) so that you can access them from other services.
When using AWS SSO, you use your SSO credentials through a unique URL to directly access SageMaker Studio. You don't have to interact with the AWS Management Console to run your notebooks.

Studio notebooks are equipped with a set of predefined SageMaker image settings to get you started faster.

**Note**

Studio notebooks don't support *local mode*. However, you can use a notebook instance to train a sample of your dataset locally, and then use the same code in a Studio notebook to train on the full dataset.

When you open a notebook in SageMaker Studio, the view is an extension of the JupyterLab interface. The primary features are the same, so you'll find the typical features of a Jupyter notebook and JupyterLab. For more information about the Studio interface, see *Amazon SageMaker Studio UI Overview* (p. 79).

---

## Get Started

To get started, you or your organization’s administrator need to complete the Amazon SageMaker Studio onboarding process. For more information, see *Onboard to Amazon SageMaker Studio* (p. 34).

You can access an SageMaker Studio notebook in any of the following ways:

- You receive an email invitation to access Studio through your organization's AWS SSO account, which includes a direct link to login to Studio without having to use the Amazon SageMaker console. You can proceed to the *the section called “Next Steps”* (p. 86).
- You receive a link to a shared Studio notebook, which includes a direct link to log in to Studio without having to use the SageMaker console. You can proceed to the *the section called “Next Steps”* (p. 86).
- You onboard to Studio and then log in to the SageMaker console. For more information, see *Onboard to Amazon SageMaker Studio* (p. 34).

---

## Log In from the Amazon SageMaker console

**To log in from the SageMaker console**

1. Onboard to Amazon SageMaker Studio. If you've already onboarded, skip to the next step.
2. Open the SageMaker console.
3. Choose *Amazon SageMaker Studio*.
4. The Amazon SageMaker Studio Control Panel opens.
5. In the Amazon SageMaker Studio Control Panel, you'll see a list of user names.
   
   Next to your user name, choose *Open Studio*.

---

## Next Steps

Now that you're in Studio, you can try any of the following options:

- Create a SageMaker Studio notebook – Continue to the next section.
- Familiarize yourself with the SageMaker Studio interface – See *Amazon SageMaker Studio UI Overview* (p. 79).
- Follow a Studio end-to-end tutorial – See *Amazon SageMaker Studio Tour* (p. 50).
Create or Open an Amazon SageMaker Studio Notebook

When you create a notebook in Amazon SageMaker Studio or open a non-shared notebook in Studio for the first time, you have to select a SageMaker image and kernel for the notebook. SageMaker launches the notebook on a default instance of a type based on the chosen SageMaker image. For CPU based images, the default instance type is `ml.t3.medium` (available as part of the AWS Free Tier). For GPU based images, the default instance type is `ml.g4dn.xlarge`.

If you create or open additional notebooks that use the same instance type, whether or not the notebooks use the same kernel, the notebooks run on the same instance of that instance type.

After a notebook is launched, you can change its instance type, and SageMaker image and kernel from within the notebook. For more information, see Change an Instance Type (p. 94) and Change a Kernel (p. 95).

You can have only one instance of each instance type. Each instance can have multiple SageMaker images running on it. Each SageMaker image can run multiple kernels or terminal instances.

Billing occurs per instance and starts when the first instance of a given instance type is launched. If you want to create or open a notebook without the risk of incurring charges, open the notebook from the File menu and choose No Kernel from the Select Kernel dialog. You can read and edit a notebook without a running kernel but you can't run cells.

Billing ends when the SageMaker image for the instance is shut down. For more information, see Usage Metering (p. 99).

For information on shutting down the notebook, see Shut Down Resources (p. 98).

Topics
- Open a Studio notebook (p. 87)
- Create a Notebook from the File Menu (p. 87)
- Create a Notebook from the Launcher (p. 88)

Open a Studio notebook

SageMaker Studio can only open notebooks listed in the Studio file browser. For instructions on uploading a notebook to the file browser, see Upload files to SageMaker Studio (p. 119) or Clone a Git Repository in SageMaker Studio (p. 120).

To open a notebook
1. In the left sidebar, choose the File Browser icon ( ) to display the file browser.
2. Browse to a notebook file and double-click it to open the notebook in a new tab.

Create a Notebook from the File Menu

To create a notebook from the File menu
1. From the Studio menu, choose File, choose New, and then choose Notebook.
2. On the Select Kernel dialog, to use the default kernel, Python 3 (Data Science), choose Select. Otherwise, use the dropdown menu to select a different kernel.
Create a Notebook from the Launcher

To create a notebook from the Launcher

1. To open the Launcher, use the keyboard shortcut Ctrl + Shift + L.
   
   Alternatively, from the File Browser, choose the plus (+) sign on the left of the menu.

2. On the Launcher, keep the default SageMaker image, Data Science, or use the dropdown menu to select a different image.


For a list of the available images, see Available Amazon SageMaker Images (p. 101).

After you choose the kernel or image, your notebook launches and opens in a new Studio tab. To view the notebook’s kernel session, in the left sidebar, choose the Running Terminals, Kernels, and Images icon ( ). You can stop the notebook’s kernel session from this view.

Use the SageMaker Studio Notebook Toolbar

Amazon SageMaker Studio notebooks extend the JupyterLab interface. For an overview of the basic JupyterLab interface, see The JupyterLab Interface.

The following image shows the toolbar and an empty cell from a SageMaker Studio notebook.

<table>
<thead>
<tr>
<th>Icon</th>
<th>Description</th>
</tr>
</thead>
</table>
| ![Save and checkpoint](icon.png) | **Save and checkpoint**  
Saves the notebook and updates the checkpoint file. For more information, see Get the Difference Between the Last Checkpoint (p. 93). |
| ![Insert cell](icon.png) | **Insert cell**  
Inserts a code cell below the current cell. The current cell is noted by the blue vertical marker in the left margin. |
| ![Cut, copy, and paste cells](icon.png) | **Cut, copy, and paste cells**  
Cuts, copies, and pastes the selected cells. |
<table>
<thead>
<tr>
<th>Icon</th>
<th>Description</th>
</tr>
</thead>
</table>
| ![Play](image) | **Run cells**
Runs the selected cells and then makes the cell that follows the last selected cell the new selected cell. |
| ![Stop](image) | **Interrupt kernel**
Interrupts the kernel, which cancels the currently running operation. The kernel remains active. |
| ![Reload](image) | **Restart kernel**
Restarts the kernel. Variables are reset. Unsaved information is not affected. |
| ![Code](image) | **Cell type**
Displays or changes the current cell type. The cell types are:
- Code – Code that the kernel runs.
- Markdown – Text rendered as markdown.
- Raw – Content, including Markdown markup, that’s displayed as text. |
| ![Terminal](image) | **Launch terminal**
Launches a terminal in the SageMaker image hosting the notebook. For an example, see [Get App Metadata](p. 92). |
| ![Clock](image) | **Checkpoint diff**
Opens a new tab that displays the difference between the notebook and the checkpoint file. For more information, see [Get the Difference Between the Last Checkpoint](p. 93). |
| ![Git](image) | **Git diff**
Only enabled if the notebook is opened from a Git repository. Opens a new tab that displays the difference between the notebook and the last Git commit. For more information, see [Get the Difference Between the Last Commit](p. 94). |

**Kernel: CPU: 0.00% MEM: 2.81%**

**CPU and memory of the kernel or instance**
Displays the CPU usage and memory usage. Double-click to toggle between the current kernel and the current instance.
### Share and Use a Notebook

You can share your Amazon SageMaker Studio notebooks with your colleagues. The shared notebook is a copy. After you share your notebook, any changes you make to your original notebook aren't reflected in the shared notebook and any changes your colleague's make in their shared copies of the notebook aren't reflected in your original notebook. If you want to share your latest version, you must create a new snapshot and then share it.

#### Topics
- Share a Notebook (p. 91)
Share a Notebook

The following screenshot shows the menu from a Studio notebook.

![Screenshot of a Studio notebook menu](image)

**To share a notebook**

1. In the upper-right corner of the notebook, choose **Share**.
2. (Optional) In **Create shareable snapshot**, choose any of the following items:
   - **Include Git repo information** – Includes a link to the Git repository that contains the notebook. This enables you and your colleague to collaborate and contribute to the same Git repository.
   - **Include output** – Includes all notebook output that has been saved.

   **Note**
   If you’re an AWS SSO user and you don’t see these options, your AWS SSO administrator probably disabled the feature. Contact your administrator.
3. Choose **Create**.
4. After the snapshot is created, choose **Copy link** and then choose **Close**.
5. Share the link with your colleague.

After selecting your sharing options, you are provided with a URL. You can share this link with users that have access to Amazon SageMaker Studio. When the user opens the URL, they’re prompted to log in using AWS SSO or IAM authentication. This shared notebook becomes a copy, so changes made by the recipient will not be reproduced in your original notebook.

**Use a Shared Notebook**

You use a shared notebook in the same way you would with a notebook that you created yourself. When you click a link to a shared notebook for the first time, a read-only version of the notebook opens. To edit the shared notebook, choose **Create a Copy**. This copies the shared notebook to your personal storage.

The copied notebook launches on an instance of the instance type and SageMaker image that the notebook was using when the sender shared it. If you aren’t currently running an instance of the instance type, a new instance is started. Customization to the SageMaker image isn’t shared. You can also inspect the notebook snapshot by choosing **Snapshot Details**.

The following are some important considerations about sharing and authentication:

- If you have an active session, you see a read-only view of the notebook until you choose **Create a Copy**.
- If you don’t have an active session, you need to log in.
- If you use IAM to login, after you login, select your user profile then choose **Open SageMaker Studio**. Then you need to choose the link you were sent.
- If you use SSO to login, after you login the shared notebook is opened automatically in Studio.
Get Notebook and App Metadata

You can access notebook metadata and App metadata using the Amazon SageMaker UI.

Topics
- Get Notebook Metadata (p. 92)
- Get App Metadata (p. 92)

Get Notebook Metadata

Jupyter notebooks contain optional metadata that you can access through the Amazon SageMaker UI.

To view the notebook metadata
1. In the left sidebar, choose the Notebook Tools icon.
2. Open the Advanced Tools section.

The metadata should look similar to the following.

```
{
    "instance_type": "ml.t3.medium",
    "kernelspec": {
        "display_name": "Python 3 (Data Science)",
        "language": "python",
        "name": "python3__SAGEMAKER_INTERNAL__arn:aws:sagemaker:us-west-2:<acct-id>:image/datascience-1.0"
    },
    "language_info": {
        "codemirror_mode": {
            "name": "ipython",
            "version": 3
        },
        "file_extension": ".py",
        "mimetype": "text/x-python",
        "name": "python",
        "nbconvert_exporter": "python",
        "pygments_lexer": "ipython3",
        "version": "3.7.6"
    }
}
```

Get App Metadata

When you create a notebook in Amazon SageMaker Studio, the App metadata is written to a file named `resource-metadata.json` in the folder `/opt/ml/metadata/`. You can get the App metadata by opening an Image terminal from within the notebook. The metadata gives you the following information, which includes the SageMaker image and instance type the notebook runs in:

- **AppType** – KernelGateway
- **DomainId** – Same as the StudioID
- **UserProfileName** – The profile name of the current user
- **ResourceArn** – The Amazon Resource Name (ARN) of the App, which includes the instance type
- **ResourceName** – The name of the SageMaker image
Additional metadata might be included for internal use by Studio and is subject to change.

**To get the App metadata**

1. In the center of the notebook menu, choose the **Launch Terminal** icon (нстру). This opens a terminal in the SageMaker image that the notebook runs in.
2. Run the following commands to display the contents of the `resource-metadata.json` file.

```
cd /opt/ml/metadata/
cat resource-metadata.json
```

The file should look similar to the following.

```
{
  "AppType": "KernelGateway",
  "DomainId": "d-xxxxx",
  "UserProfileName": "profile-name",
  "ResourceName": "datascience--1-0-ml"
}
```

**Get Notebook Differences**

You can display the difference between the current notebook and the last checkpoint or the last Git commit using the Amazon SageMaker UI.

The following screenshot shows the menu from a Studio notebook.

![Screenshot of Studio notebook menu](image)

**Topics**

- Get the Difference Between the Last Checkpoint (p. 93)
- Get the Difference Between the Last Commit (p. 94)

**Get the Difference Between the Last Checkpoint**

When you create a notebook, a hidden checkpoint file that matches the notebook is created. You can view changes between the notebook and the checkpoint file or revert the notebook to match the checkpoint file.

By default, a notebook is auto-saved every 120 seconds and also when you close the notebook. However, the checkpoint file isn't updated to match the notebook. To save the notebook and update the checkpoint file to match, you must choose the **Save notebook and create checkpoint** icon (нстру) on the left of the notebook menu or use the `Ctrl + S` keyboard shortcut.

To view the changes between the notebook and the checkpoint file, choose the **Checkpoint diff** icon (нстру) in the center of the notebook menu.
To revert the notebook to the checkpoint file, from the main Studio menu, choose File then Revert Notebook to Checkpoint.

**Get the Difference Between the Last Commit**

If a notebook is opened from a Git repository, you can view the difference between the notebook and the last Git commit.

To view the changes in the notebook from the last Git commit, choose the Git diff icon (git) in the center of the notebook menu.

**Manage Resources**

You can change the instance type, and SageMaker image and kernel from within an Amazon SageMaker Studio notebook. To create a custom kernel to use with your notebooks, see Bring your own SageMaker image (p. 102).

**Topics**

- Change an Instance Type (p. 94)
- Change a Kernel (p. 95)
- Shut Down Resources (p. 98)

**Change an Instance Type**

With Amazon SageMaker Studio notebooks, you can change the Amazon Elastic Compute Cloud (Amazon EC2) instance type that your notebook runs on from within the notebook.

When you open a new notebook for the first time, you are assigned a default instance type to run the notebook. When you open additional notebooks on the same instance type, the notebooks run on the same instance as the first notebook, even if the notebooks use different kernels.

**Important**

If you change the instance type, unsaved information and existing settings for the notebook are lost, and installed packages must be re-installed.

The following screenshot shows the menu from a Studio notebook. The processor and memory of the instance type powering the notebook are displayed as 2 vCPU + 4 GiB.

![Screenshot of Studio notebook menu showing instance type](image)

To change the instance type

1. Choose the instance type.
2. In Select instance, choose one of the fast launch instance types that are listed. Or to see all instance types, switch off Fast launch only. The list can be sorted by any column.
3. After choosing a type, choose **Save and continue**.

4. Wait for the new instance to become enabled, and then the new instance type information is displayed.

For a list of the available instance types, see [Available Instance Types](#) (p. 100).

### Change a Kernel

With Amazon SageMaker Studio notebooks, you can change the notebook's kernel from within the notebook.

The following screenshot shows the menu from a Studio notebook. The current SageMaker image and kernel are displayed as **Python 3 (Data Science)**, where Python 3 denotes the kernel and Data Science denotes the SageMaker image that contains the kernel. The color of the circle to the right indicates the kernel is idle or busy. The kernel is busy when the center and the edge of the circle are the same color.
To change a notebook's kernel

1. Choose the kernel name.
2. From the drop-down list, choose a kernel.
Start Preferred Kernel

R (Custom R Image) (r-image/latest)

Use No Kernel

No Kernel

Start Other Kernel

Python 3 (Base Python)
Python 3 (Data Science)
Python 3 (MXNet CPU Optimized)
Python 3 (MXNet GPU Optimized)
Python 3 (PyTorch CPU Optimized)
Python 3 (PyTorch GPU Optimized)
Python 3 (TensorFlow 2 CPU Optimized)
Python 3 (TensorFlow 2 GPU Optimized)
Python 3 (TensorFlow CPU Optimized)
Python 3 (TensorFlow GPU Optimized)

Use Kernel from Preferred Session

Use Kernel from Other Session

No Kernel
3. After choosing a kernel, choose Select.
4. Wait for the kernel's status to show as idle, which indicates the kernel has started.

For a list of available SageMaker kernels, see Available Amazon SageMaker Kernels (p. 101).

**Shut Down Resources**

You can shut down individual resources, including notebooks, terminals, kernels, apps, and instances. You can also shut down all resources in one of these categories at the same time.

**Topics**
- Shut Down an Open Notebook (p. 98)
- Shut Down Resources (p. 98)

**Shut Down an Open Notebook**

You can shut down an open notebook from the Amazon SageMaker Studio File menu or from the Running Terminal and Kernels pane.

**Note**
When you shut down a notebook, any unsaved information in the notebook is lost. The notebook is not deleted.

**To shut down an open notebook from the File menu**

1. Optionally, save the notebook contents by choosing the Disk icon on the left of the notebook menu.
2. Choose File then Close and Shutdown Notebook.
3. Choose OK.

**Shut Down Resources**

The Running Terminals and Kernels pane consists of four sections. Each section lists all the resources of that type. You can shut down each resource individually or shut down all the resources in a section at the same time.

When you choose to shut down all resources in a section, the following occurs:

- **RUNNING INSTANCES/RUNNING APPS** – All instances, apps, notebooks, kernel sessions, consoles/shells, and image terminals are shut down. System terminals aren't shut down. Choose this option to stop the accrual of all charges.
- **KERNEL SESSIONS** – All kernels, notebooks and consoles/shells are shut down.
- **TERMINAL SESSIONS** – All image terminals and system terminals are shut down.

**To shut down resources**

1. In the left sidebar, choose the Running Terminals and Kernels icon (●).
2. Do either of the following:
   - To shut down a specific resource, choose the SHUT DOWN icon (⇧) on the same row as the resource.
For running instances, a confirmation dialog lists all the resources that will be shut down. For running apps, a confirmation dialog is displayed. Choose **Shut down all** to proceed.

**Note**
No confirmation dialog is displayed for kernel sessions or terminal sessions.
- To shut down all resources in a section, choose the X to the right of the section label. A confirmation dialog is displayed. Choose **Shut down all** to proceed.

## Usage Metering

The costs incurred for running Amazon SageMaker Studio notebooks, interactive shells, consoles, and terminals are based on Amazon Elastic Compute Cloud (Amazon EC2) instance usage.

When you launch the following resources, you must choose a SageMaker image and kernel:

**From the Studio Launcher**

- Notebook
- Interactive Shell
- Image Terminal

**From the File menu**

- Notebook
- Console

When launched, the resource is run on an Amazon EC2 instance of an instance type based on the chosen SageMaker image and kernel. If an instance of that type was previously launched and is available, the resource is run on that instance. If an instance of that type is not available, the resource is run on a new default instance of that type.

For CPU based images, the default instance type is `ml.t3.medium`. For GPU based images, the default instance type is `ml.g4dn.xlarge`.

The costs incurred are based on the instance type and the number of instances of each instance type. You are billed separately for each instance.

Metering starts when an instance is created. Metering ends when the instance is shut down. For information on how to shut down an instance, see [Shut Down Resources](#) (p. 98).

**Important**
You must shut down the instance to stop incurring charges. If you shut down the notebook running on the instance but don’t shut down the instance, you will still incur charges.

When you open multiple notebooks on the same instance type, the notebooks run on the same instance even if they’re using different kernels. You are billed only for the time that one instance is running.

You can change the instance type from within the notebook after you open it. For more information, see [Change an Instance Type](#) (p. 94).

For information about billing along with pricing examples, see [Amazon SageMaker Pricing](#).

## Available Resources

The following sections list the available resources for Amazon SageMaker Studio notebooks.
Available Instance Types

The following Amazon Elastic Compute Cloud (Amazon EC2) instance types are available.

For detailed information on which instance types fit your use case, and their performance capabilities, see Amazon Elastic Compute Cloud Instance types.

Note
For most use cases, you should use a ml.t3.medium. This is the default instance type for CPU-based SageMaker images, and is available as part of the AWS Free Tier.

>> Fast launch instances types are optimized to start in under two minutes.

Default instance types

- CPU-based images: ml.t3.medium >> Fast launch
- GPU-based images: ml.g4dn.xlarge >> Fast launch

General purpose (no GPUs)

- ml.t3.medium >> Fast launch
- ml.t3.large
- ml.t3.xlarge
- ml.t3.2xlarge
- ml.m5.large >> Fast launch
- ml.m5.xlarge
- ml.m5.2xlarge
- ml.m5.4xlarge
- ml.m5.8xlarge
- ml.m5.12xlarge
- ml.m5.16xlarge
- ml.m5.24xlarge

Compute optimized (no GPUs)

- ml.c5.large >> Fast launch
- ml.c5.xlarge
- ml.c5.2xlarge
- ml.c5.4xlarge
- ml.c5.9xlarge
- ml.c5.12xlarge
- ml.c5.18xlarge
- ml.c5.24xlarge
**Available Resources**

**Accelerated computing (1+ GPUs)**
- ml.p3.2xlarge
- ml.p3.8xlarge
- ml.p3.16xlarge
- ml.g4dn.xlarge >> Fast launch
- ml.g4dn.2xlarge
- ml.g4dn.4xlarge
- ml.g4dn.8xlarge
- ml.g4dn.12xlarge
- ml.g4dn.16xlarge

**Available Amazon SageMaker Images**
The following SageMaker images are available in Amazon SageMaker Studio. SageMaker images contain the latest Amazon SageMaker Python SDK and the latest version of the kernel. The name in brackets ([ ]) is the resource identifier of the SageMaker image as specified in the Amazon Resource Name (ARN) for the SageMaker image.

- Data Science [datascience-1.0]
  
  Data Science is a Conda image with the most commonly used Python packages and libraries, such as NumPy and SciKit Learn.
- Base Python [python-3.6]
- MXNet (optimized for CPU) [mxnet-1.6-cpu-py36]
- MXNet (optimized for GPU) [mxnet-1.6-gpu-py36]
- PyTorch (optimized for CPU) [pytorch-1.4-cpu-py36]
- PyTorch (optimized for GPU) [pytorch-1.4-gpu-py36]
- TensorFlow (optimized for CPU) [tensorflow-1.15-cpu-py36]
- TensorFlow (optimized for GPU) [tensorflow-1.15-gpu-py36]
- TensorFlow 2 (optimized for CPU) [tensorflow-2.1-cpu-py36]
- TensorFlow 2 (optimized for GPU) [tensorflow-2.1-gpu-py36]

**Available Amazon SageMaker Kernels**
The following Amazon SageMaker kernels are available in SageMaker Studio. The name in parentheses is the SageMaker image hosting the kernel.

- Python 3 (Data Science)
- Python 3 (Base Python)
- Python 3 (MXNet CPU Optimized)
- Python 3 (MXNet GPU Optimized)
- Python 3 (PyTorch CPU Optimized)
- Python 3 (PyTorch GPU Optimized)
- Python 3 (TensorFlow CPU Optimized)
- Python 3 (TensorFlow GPU Optimized)
- Python 3 (TensorFlow 2 CPU Optimized)
- Python 3 (TensorFlow 2 GPU Optimized)
Bring your own SageMaker image

A SageMaker image consists of the kernels, language packages, and other files required to run a Jupyter notebook in Amazon SageMaker Studio. Amazon SageMaker provides many built-in images for you to use. If you need different functionality, you can bring your own custom images to Studio. For the list of built-in images, see Available Amazon SageMaker Images (p. 101).

A SageMaker image is a holder for a set of SageMaker image versions. An image version represents a container image that is compatible with SageMaker Studio and stored in an Amazon Elastic Container Registry (ECR) repository. Each image version is immutable.

To make a custom SageMaker image available to all users within a domain, you attach the image to the domain. To make an image available to a single user, you attach the image to the user's profile. When you attach an image, SageMaker uses the latest image version by default. You can also attach a specific image version. After you attach the version, you can choose the version from the SageMaker Launcher or the image selector when you launch a notebook.

You can create images and image versions, and attach image versions to your domain, using the SageMaker Studio control panel, the AWS SDK for Python (Boto3), and the AWS Command Line Interface (AWS CLI). You can also create images and image versions using the SageMaker console, even if you haven't onboarded to Studio.

SageMaker provides sample Dockerfiles to use as a starting point for your custom SageMaker images in the SageMaker Studio Custom Image Samples repository. These include Dockerfiles for the following images:

- Julia
- R
- Scala
- Tensorflow 2

**Topics**

- Create a custom SageMaker image (Console) (p. 102)
- Attach a custom SageMaker image (Control Panel) (p. 103)
- Launch a custom SageMaker image in SageMaker Studio (p. 105)
- Bring your own custom SageMaker image tutorial (p. 107)
- Custom SageMaker image specifications (p. 115)

Create a custom SageMaker image (Console)

This topic describes how you can create a custom SageMaker image using the SageMaker console. You can also create the image using the SageMaker Studio control panel. The steps are the same. For information on using the Studio control panel, see Attach a custom SageMaker image (Control Panel) (p. 103).

When you create an image, SageMaker also creates an initial image version. The image version represents a container image in Amazon Elastic Container Registry (ECR). The container image must satisfy the requirements to be used in Amazon SageMaker Studio. For more information, see Custom SageMaker image specifications (p. 115).

**To create an image**

1. Open the Amazon SageMaker console at https://console.aws.amazon.com/sagemaker/.
2. In the left navigation pane, choose Images.
3. On the **Custom images** page, choose **Create image**.

4. For **Image source**, enter the registry path to the container image in Amazon ECR. The path is in the following format:

   `acct-id.dkr.ecr.region.amazonaws.com/repo-name[:tag] or [@digest]`

5. Choose **Next**.

6. Under **Image properties**, enter the following:

   - **Image name** – The name must be unique to your account in the current AWS Region.
   - (Optional) **Display name** – The name displayed in the Studio user interface. When not provided, **Image name** is displayed.
   - (Optional) **Description** – A description of the image.
   - **IAM role** – The role must have the **AmazonSageMakerFullAccess** policy attached. Use the dropdown menu to choose one of the following options:
     - Create a new role – Specify any additional Amazon Simple Storage Service (Amazon S3) buckets that you want users of your notebooks to have access to. If you don't want to allow access to additional buckets, choose **None**. SageMaker attaches the **AmazonSageMakerFullAccess** policy to the role. The role allows users of your notebooks access to the S3 buckets listed next to the checkmarks.
     - Enter a custom IAM role ARN – Enter the Amazon Resource Name (ARN) of your IAM role.
     - Use existing role – Choose one of your existing roles from the list.
   - (Optional) **Image tags** – Choose **Add new tag**. You can add up to 50 tags. Tags are searchable using the Studio user interface, the SageMaker console, or the SageMaker **Search API**.

7. Choose **Submit**.

The new image is displayed in the **Custom images** list and briefly highlighted. After the image has been successfully created, you can choose the image name to view its properties or choose **Create version** to create another version.

**To create another image version**

1. Choose **Create version** on the same row as the image.
2. For **Image source**, enter the registry path to the ECR container image. The container image shouldn't be the same image as used in a previous version of the SageMaker image.

To use the custom image in Studio, you must attach it to your domain. For more information, see **Attach a custom SageMaker image (Control Panel)** (p. 103).

**Attach a custom SageMaker image (Control Panel)**

To use a custom SageMaker image, you must attach a version of the image to your domain. When you attach an image version, it appears in the SageMaker Studio Launcher and is available in the **Select image** dropdown list, which users use to launch an activity or change the image used by a notebook.

There is a limit to the number of image versions that can be attached at any given time. After you reach the limit, you must detach a version in order to attach another version of the image.

**Attach an existing image version to your domain**

This topic describes how you can attach an existing custom SageMaker image version to your domain using the SageMaker Studio control panel. You can also create a custom SageMaker image and image version, and then attach that version to your domain.
The steps to create an image and image version are the same whether you use the Studio control panel or the SageMaker console. For the procedure to create an image and image version, see Create a custom SageMaker image (Console) (p. 102).

To attach an existing image

1. Open the Amazon SageMaker console at https://console.aws.amazon.com/sagemaker/.
2. In the left navigation pane, choose Amazon SageMaker Studio.
3. On the SageMaker Studio Control Panel, under Custom images attached to domain, choose Attach image.
4. For Image source, choose Existing image.
5. Choose an existing image from the list.
6. Choose a version of the image from the list.
7. Choose Next.
8. Choose the IAM role. For more information, see Create a custom SageMaker image (Console) (p. 102).
9. Choose Next.
10. Under Studio configuration, enter or change the following settings. For information on how to get the kernel information from the image, see DEVELOPMENT in the SageMaker Studio Custom Image Samples repository.
   - EFS mount path – The path within the image to mount the user's Amazon Elastic File System (EFS) home directory.
   - Kernel:
     - For Kernel name, enter the name of an existing kernel in the image.
     - (Optional) For Kernel display name, enter the display name for the kernel.
     - Choose Add kernel.
   - (Optional) Configuration tags – Choose Add new tag and then add a configuration tag.

   **Note**
   For more information, see the Kernel discovery and User data sections of Custom SageMaker image specifications (p. 115).
11. Choose Submit

Wait for the image version to be attached to the domain. When attached, the version is displayed in the Custom images list and briefly highlighted.

**Detach a custom SageMaker image**

When you detach an image from a domain, all versions of the image are detached. When an image is detached, all users of the domain lose access to the image versions.

A running notebook that has a kernel session on an image version when the version is detached, continues to run. When the notebook is stopped or the kernel is shut down the image version becomes unavailable.

**To detach an image**

1. In the SageMaker Studio control panel, under Custom images attached to domain, choose the image and then choose Detach.
2. (Optional) To delete the image and all versions from SageMaker, select Also delete the selected images .... This does not delete the associated container images from Amazon ECR.
3. Choose **Detach**.

**Launch a custom SageMaker image in SageMaker Studio**

After you create your custom SageMaker image and attach it to your domain, the image appears in the image selector dialog box of the SageMaker Studio Launcher, and the kernel appears in the kernel selector dialog box.

**To launch and select your custom image**

2. Use the keyboard shortcut Ctrl + Shift + L to open Studio Launcher.

3. Open the **Select a SageMaker image** dropdown menu.
4. Choose your custom image.

5. Launch a notebook or interactive shell in the custom image.

6. In an open notebook, you can switch to the custom kernel by choosing a different kernel in the Select Kernel dialog box.
Note
If you encounter an error when launching the image, check your Amazon CloudWatch logs. The name of the log group is /aws/sagemaker/studio. The name of the log stream is $domainID/$userProfileName/KernelGateway/$appName.

Bring your own custom SageMaker image tutorial

In this tutorial, you create a custom SageMaker image and attach a version of the image to your domain for use in Amazon SageMaker Studio. The image version contains a selection of R packages, along with the AWS SDK for Python (Boto3) and the Amazon SageMaker Python SDK. After you complete this tutorial, you can select the version in Studio and use R to access the SDKs using the RStudio reticulate package. For more information, see R Interface to Python.

Two methods are presented to attach the image version to your domain. In the first method, you create a new domain with the version attached. This method is simpler but you need to specify the Amazon Virtual Private Cloud (VPC) information and execution role that's required to create the domain.

If you have onboarded to Studio, you can use the second method to attach the image version to your current domain. In this case, you don't need to specify the VPC information and execution role. After you attach the version, you must delete all the apps in your domain and reopen Studio.

You can't run this tutorial from Studio for the following reasons:

- Docker isn't available inside Studio.
- You can't create or update a domain within Studio.
Prerequisites

- The Docker application. For information about setting up Docker, see Orientation and setup.
- A local copy of the Dockerfile for creating a Studio compatible R image from the SageMaker Studio custom image samples repository.
  
  **Note**
  Building the R image from the Dockerfile installs dependencies that may be licensed under copyleft licenses such as GPLv3. You should review the license terms and make sure they are acceptable for your use case before proceeding and building this image.

- Permissions to access the Amazon Elastic Container Registry (Amazon ECR) service. For more information, see Amazon ECR Managed Policies.

- An AWS Identity and Access Management execution role that has the AmazonSageMakerFullAccess policy attached. If you have onboarded to Amazon SageMaker Studio, you can get the role from the Studio Summary section of the SageMaker Studio control panel.

Links

- Amazon Elastic Container Registry API Reference
- Docker Engine API
- Dockerfile reference

Topics

- Add a Studio-compatible container image to Amazon ECR (p. 108)
- Create a SageMaker image from the ECR container image (p. 110)
- Attach the SageMaker image to a new domain (p. 111)
- Attach the SageMaker image to your current domain (p. 113)
- View the attached image in the Studio control panel (p. 114)
- Clean up resources (p. 115)

Add a Studio-compatible container image to Amazon ECR

You perform the following steps to add a container image to Amazon ECR:

- Create an Amazon ECR repository.
- Authenticate to Amazon ECR.
- Build a Studio-compatible container image.
- Push the image to the Amazon ECR repository.

**Note**
The Amazon ECR repository must be in the same AWS Region as SageMaker Studio.

**To build and add a container image to Amazon ECR**

1. Create an Amazon ECR repository using the AWS CLI. To create the repository using the Amazon ECR console, see Creating a repository.

   ```bash
   aws ecr create-repository \
   --repository-name smstudio-custom \
   --image-scanning-configuration scanOnPush=true
   ```
2. Authenticate to Amazon ECR. Make sure the Docker application is running. For more information, see Registry Authentication.

   **Note**
   The `get-login-password` command was introduced in AWS CLI version 1.17.10. Use `aws --version` to determine your version of the AWS CLI. For information about upgrading, see Installing the AWS CLI.

   ```bash
   aws ecr get-login-password | \n   docker login --username AWS --password-stdin \\
   <acct-id>.dkr.ecr.<region>.amazonaws.com/smstudio-custom
   ```

   Response:

   Login Succeeded

3. Build the R image Dockerfile. The period (.) specifies that the Dockerfile should be in the context of the build command.

   **Note**
   You can't directly build a Dockerfile within Studio.

   ```bash
docker build . -t smstudio-r -t \\
<acct-id>.dkr.ecr.<region>.amazonaws.com/smstudio-custom:r
```

   Response:

   ```bash
   Successfully built f97aaaa805b1
   Successfully tagged smstudio-r:latest
   Successfully tagged acct-id.dkr.ecr.us-east-2.amazonaws.com/smstudio-custom:r
   ```

4. Push the container image to the Amazon ECR repository. For more information, see ImagePush and Pushing an image.

   ```bash
docker push <acct-id>.dkr.ecr.<region>.amazonaws.com/smstudio-custom:r
```

   Response:

   ```bash
   The push refers to repository [acct-id.dkr.ecr.us-east-2.amazonaws.com/smstudio-custom] 
   r: digest: sha256:7a5c8cb01944f3b10fe495a930b3682d15b7b1de9f1f5497d1a3e3634369cead 
   size: 3066
   ```
Create a SageMaker image from the ECR container image

You perform the following steps to create a SageMaker image from the container image:

- Create an Image.
- Create an ImageVersion.
- Create a configuration file.
- Create an AppImageConfig.

To create the SageMaker image entities

1. Create a SageMaker image.

   ```bash
   aws sagemaker create-image \
   --image-name r-image \
   --role-arn arn:aws:iam::<acct-id>:role/service-role/<execution-role>
   ```

   **Response:**

   ```json
   {
     "ImageArn": "arn:aws:sagemaker:us-east-2:<acct-id>:image/r-image"
   }
   ```

2. Create a SageMaker image version from the container image.

   ```bash
   aws sagemaker create-image-version \
   --image-name r-image \
   --base-image <acct-id>.dkr.ecr.<region>.amazonaws.com/smstudio-custom:r
   ```

   **Response:**

   ```json
   {
     "ImageVersionArn": "arn:aws:sagemaker:us-east-2:<acct-id>:image-version/r-image/1"
   }
   ```

3. Check that the image version was successfully created.

   ```bash
   aws sagemaker describe-image-version \
   --image-name r-image \
   --version 1
   ```

   **Response:**

   ```json
   {
     "ImageVersionStatus": "CREATED"
   }
   ```

   **Note**

   If the response is "ImageVersionStatus": "CREATED_FAILED", the response also includes the failure reason. A permissions issue is a common cause of failure. You can check your Amazon CloudWatch logs. The name of the log group is /aws/sagemaker/studio. The name of the log stream is $domainID/$userProfileName/KernelGateway/$appName.
4. Create a configuration file, named app-image-config-input.json. For information on how the get the following information from the image, see DEVELOPMENT in the SageMaker Studio Custom Image Samples repository.

```json
{
    "AppImageConfigName": "r-image-config",
    "KernelGatewayImageConfig": {
        "KernelSpecs": [
            {
                "Name": "ir",
                "DisplayName": "R (Custom R Image)"
            }
        ],
        "FileSystemConfig": {
            "MountPath": "/home/sagemaker-user",
            "DefaultUid": 1000,
            "DefaultGid": 100
        }
    }
}
```

5. Create the AppImageConfig using the file created in the previous step.

```
aws sagemaker create-app-image-config
   --cli-input-json file://app-image-config-input.json
```

Response:

```json
{
}
```

---

**Attach the SageMaker image to a new domain**

To use this method, you need to specify an execution role that has the AmazonSageMakerFullAccess policy attached.

**Note**

You can have only one domain. If you have onboarded to SageMaker Studio, you must delete your current domain before you can use this method. For more information, see Delete an Amazon SageMaker Studio Domain (p. 39).

You perform the following steps to create the domain and attach the custom SageMaker image:

- Get your default VPC ID and subnet IDs.
- Create the configuration file for the domain, which specifies the image.
- Create the domain with the configuration file.

**To add the custom SageMaker image to your domain**

1. Get your default VPC ID.

```
aws ec2 describe-vpcs
   --filters Name=isDefault,Values=true
   --query "Vpcs[0].VpcId" --output text
```
Response:

vpc-xxxxxxxx

2. Get your default subnet IDs using the VPC ID from the previous step.

```bash
aws ec2 describe-subnets
   --filters Name=vpc-id,Values=<vpc-id>
   --query "Subnets[*].SubnetId" --output json
```

Response:

```
[   "subnet-b55171dd",
    "subnet-8a5f99c6",
    "subnet-e88d1392"
]
```

3. Create a configuration file named `create-domain-input.json`. Insert the VPC ID, subnet IDs, ImageName, and AppImageConfigName from the previous steps. Because ImageVersionNumber isn’t specified, the latest version of the image is used, which is the only version in this case.

```json
{
   "DomainName": "domain-with-custom-r-image",
   "VpcId": "<vpc-id>",
   "SubnetIds": [
      "<subnet-ids>"
   ],
   "DefaultUserSettings": {
      "ExecutionRole": "<execution-role>",
      "KernelGatewayAppSettings": {
         "CustomImages": [
            {
               "ImageName": "r-image",
               "AppImageConfigName": "r-image-config"
            }
         ]
      }
   },
   "AuthMode": "IAM"
}
```

4. Create the domain with the attached custom SageMaker image.

```bash
aws sagemaker create-domain
   --cli-input-json file://create-domain-input.json
```

Response:

```
{
   "Url": "https://d-xxxxxxxxxxxx.studio.us-east-2.sagemaker.aws/..."
}
```
Attach the SageMaker image to your current domain

This method presumes you've already onboarded to Amazon SageMaker Studio. For more information, see Onboard to Amazon SageMaker Studio (p. 34).

Note
You must delete all the apps in your domain before you update the domain with the new image version. For information about deleting the apps, see Delete an Amazon SageMaker Studio Domain (p. 39).

You perform the following steps to add the SageMaker image to your current domain.

- Get your DomainID from SageMaker Studio.
- Use the DomainID to get the DefaultUserSettings for the domain.
- Add the ImageName and AppImageConfig as a CustomImage to the DefaultUserSettings.
- Update your domain to include the custom image.

To add the custom SageMaker image to your domain

1. Open the Amazon SageMaker console at https://console.aws.amazon.com/sagemaker/.
2. From the top left of the navigation pane, choose Amazon SageMaker Studio.
3. From the Studio Control Panel, under Studio Summary, find the Studio ID, which is also your DomainId. The ID is in the following format: d-xxxxxxxxxxxx.
4. Use the domain ID to get the description of the domain.

```bash
aws sagemaker describe-domain \
  --domain-id <d-xxxxxxxxxxxx>
```

Response:

```json
{}
```
"DomainId": "d-xxxxxxxxxxxxx",
"DefaultUserSettings": {
  "KernelGatewayAppSettings": {
    "CustomImages": [
    ],
  ...
  }
}

5. Save the default user settings section of the response to a file named `default-user-settings.json`.

6. Insert the `ImageName` and `AppImageConfigName` from the previous steps as a custom image. Because `ImageVersionNumber` isn't specified, the latest version of the image is used, which is the only version in this case.

```
{
  "DefaultUserSettings": {
    "KernelGatewayAppSettings": {
      "CustomImages": [
        {
          "ImageName": "string",
          "AppImageConfigName": "string"
        }
      ],
    ...
  }
}
```

7. Use the domain ID and default user settings file to update your domain.

```
aws sagemaker update-domain \\n--domain-id <d-xxxxxxxxxxxxx> \\n--cli-input-json file://default-user-settings.json
```

Response:

```
{
  "DomainArn": "arn:aws:sagemaker:us-east-2:acct-id:domain/d-xxxxxxxxxxxxx"
}
```

**View the attached image in the Studio control panel**

After you create the custom SageMaker image and attach it to your domain, the image appears in the custom images list in the SageMaker Studio control panel.

For information about how to launch the image in Amazon SageMaker Studio, see Launch a custom SageMaker image in SageMaker Studio (p. 105).
Clean up resources

You perform the following steps to clean up the resources you created in the previous sections:

- Detach the image and image versions from your domain.
- Delete the image, image version, and app image config.
- Delete the container image and repository from Amazon ECR. For more information, see Deleting a repository.

To clean up resources

1. Detach the image and image versions from your domain by passing an empty custom image list to the domain. Open the `default-user-settings.json` file you created in Attach the image to a current domain (p. 113).
2. Delete the custom images and then save the file.

```json
"DefaultUserSettings": {
    "KernelGatewayAppSettings": {
        "CustomImages": [
        ],
      ...,
    },
    ...
  }
```
3. Use the domain ID and default user settings file to update your domain.

```bash
aws sagemaker update-domain \
    --domain-id <d-yyyyyyyyyyyy> \
    --cli-input-json file://default-user-settings.json
```

Response:

```json
{
    "DomainArn": "arn:aws:sagemaker:us-east-2:acct-id:domain/d-yyyyyyyyyyyy"
}
```
4. Delete the app image config.

```bash
aws sagemaker delete-app-image-config \
    --app-image-config-name r-image-config
```
5. Delete the SageMaker image, which also deletes all image versions. The container images in ECR that are represented by the image versions are not deleted.

```bash
aws sagemaker delete-image \
    --image-name r-image
```

Custom SageMaker image specifications

The following specifications apply to the container image that is represented by a SageMaker image version.
Running the image

ENTRYPOINT and CMD instructions are overridden to enable the image to run as a KernelGateway app.

Port 8888 in the image is reserved for running the KernelGateway web server.

Stopping the image

The DeleteApp API issues the equivalent of a docker stop command. Other processes in the container won't get the SIGKILL/SIGTERM signals.

Kernel discovery

SageMaker recognizes kernels as defined by Jupyter kernel specs.

You can specify a list of kernels to display before running the image. If not specified, python3 is displayed. Use the DescribeAppImageConfig API to view the list of kernels.

Conda environments are recognized as kernel specs by default.

File system

The /opt/.sagemakerinternal and /opt/ml directories are reserved. Any data in these directories might not be visible at runtime.

User data

Each user in a Studio domain gets a user directory on a shared Amazon Elastic File System volume in the image. The location of the current user's directory on the Amazon EFS volume is configurable. By default, the location of the directory is /home/sagemaker-user.

SageMaker configures POSIX UID/GID mappings between the image and the host. This defaults to mapping the root user's UID/GID (0/0) to the UID/GID on the host.

You can specify these values using the CreateAppImageConfig API.

Metadata

A metadata file is located at /opt/ml/metadata/resource-metadata.json. No additional environment variables are added to the variables defined in the image. For more information, see Get App Metadata (p. 92).

GPU

On a GPU instance, the image is run with the --gpus option. Only the CUDA toolkit should be included in the image not the NVIDIA drivers. For more information, see NVIDIA User Guide.

Metrics and Logging

Logs from the KernelGateway process are sent to Amazon CloudWatch in the customer's account. The name of the log group is /aws/sagemaker/studio. The name of the log stream is $domainID/$userProfileName/KernelGateway/$appName.

Image size

Limited to 11 GB.

Set Up a Connection to an Amazon EMR Cluster

Amazon EMR is a big data platform for processing vast amounts of data. The central component of Amazon EMR is the cluster. A cluster is a collection of Amazon EC2 instances. Apache Spark is a
distributed processing framework that runs on Amazon EMR. For more information, see What Is Amazon EMR? and Apache Spark.

Amazon SageMaker Studio comes with a SageMaker SparkMagic image that contains a PySpark kernel. The SparkMagic image also contains an AWS CLI utility, `sm-sparkmagic`, that you can use to create the configuration files required for the PySpark kernel to connect to the Amazon EMR cluster. After creating the configuration files, the utility displays the steps required to finish the setup.

For added security, you can specify that the connection to the EMR cluster uses Kerberos authentication. For more information, see Use Kerberos Authentication.

**Prerequisites**

- Access to SageMaker Studio that's set up to use Amazon VPC mode. For more information, see Choose a VPC (p. 38).
- An Amazon EMR cluster in the same VPC as Studio or in a VPC that's connected to the same VPC as Studio.
- If you use the `sm-sparkmagic` utility, the IAM execution role associated with your Studio user profile must contain the following extra permissions. To find the execution role, choose your user name in the SageMaker Studio Control Panel.

```json
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Effect": "Allow",
         "Action": [
            "elasticmapreduce:DescribeCluster",
            "elasticmapreduce:DescribeSecurityConfiguration",
            "elasticmapreduce:ListInstances"
         ],
         "Resource": [
            "arn:aws:elasticmapreduce:*:*:cluster/*"
         ]
      }
   ]
}
```

**To set up a connection to an EMR cluster**

1. Open SageMaker Studio.
2. In the upper-left corner of Studio, choose **Amazon SageMaker Studio** to open Studio Launcher.
3. On the **Launcher** page, choose **Notebooks and compute resources**.
4. For **Select a SageMaker image**, choose the **SparkMagic** image.
5. Choose **Notebook** to create a Studio notebook in the SparkMagic image.

6. Run the following code in a notebook cell to create the configuration files used to connect to the EMR cluster. `%local` ensures that the code runs in the local image instead of on Spark.

   - If the EMR cluster is not configured for Kerberos authentication, run the following command:

     ```
     %local
     ! sm-sparkmagic connect --cluster-id "cluster-id"
     ```

     The output should be similar to the following:

     ```
     Successfully read emr cluster(cluster-id) details
     SparkMagic config file location: /etc/sparkmagic/config.json
     ```

   - If the EMR cluster is configured for Kerberos authentication, run the following command:

     ```
     ! sm-sparkmagic connect --cluster-id "cluster-id" --user-name "user-name"
     ```

     The output should be similar to the following:

     ```
     Successfully read emr cluster(cluster-id) details
     SparkMagic config file location: /etc/sparkmagic/config.json
     Kerberos configuration file location: /etc/krb5.conf
     ```

7. To complete the setup, do one of the following:

   - For EMR clusters that are not configured for Kerberos authentication, go to step 8.
   - For EMR clusters that are configured for Kerberos authentication, do the following:
In the notebook toolbar, choose the **Launch terminal** icon to open a terminal in the same SparkMagic image as the notebook.

2. Run the following command in the terminal to get the Kerberos ticket:

```
kinit user-name
```

3. Enter your password when prompted.

8. In the notebook toolbar, choose the **Restart kernel** icon to complete the setup.

9. To verify that the connection was set up correctly, run the following command in a notebook cell:

```
%%info
```

The output should be similar to the following:

```
Current session configs: {'driverMemory': '1000M', 'executorCores': 2, 'kind': 'pyspark'}
No active sessions.
```

---

### Perform Common Tasks in Amazon SageMaker Studio

The following sections describe how to perform common tasks in Amazon SageMaker Studio. For an overview of the Studio interface, see Amazon SageMaker Studio UI Overview (p. 79).

**Topics**

- Upload files to SageMaker Studio (p. 119)
- Clone a Git Repository in SageMaker Studio (p. 120)
- Stop a Training Job (p. 120)
- Manage Your EFS Storage Volume (p. 120)
- Provide Feedback on SageMaker Studio (p. 121)
- Update SageMaker Studio and Studio Apps (p. 121)

### Upload files to SageMaker Studio

When you onboard to Amazon SageMaker Studio, a home directory is created for you in the Amazon Elastic File System (Amazon EFS) volume that was created for your team. Studio can open only files that have been uploaded to your directory. The Studio file browser maps to your home directory.

**To upload files to your home directory**

1. In the left sidebar, choose the **File Browser** icon.
2. In the file browser, choose the **Upload Files** icon.
3. Select the files you want to upload and then choose **Open**.
4. Double-click a file to open the file in a new tab in Studio.
Clone a Git Repository in SageMaker Studio

Amazon SageMaker Studio can connect only to a local repository. In this example, you clone the `aws/amazon-sagemaker-examples` repository (repo).

**To clone the repo**

1. In the left sidebar, choose the **File Browser** icon (agog).
2. Choose the root folder or the folder you want to clone the repo into.
3. In the left sidebar, choose the **Git** icon (agog).
4. Choose **Clone a Repository**.
5. Enter the URI for the SageMaker examples repo: `https://github.com/aws/amazon-sagemaker-examples.git`.
6. Choose **CLONE**.
7. If the repo requires credentials, you are prompted to enter your username and password.
8. Wait for the download to finish. After the repo has been cloned, the **File Browser** opens to display the cloned repo.
9. Double click the repo to open it.
10. Choose the **Git** icon to view the Git user interface which now tracks the examples repo.
11. To track a different repo, open the repo in the file browser and then choose the **Git** icon.

Stop a Training Job

You can stop a training job with the Amazon SageMaker Studio UI. When you stop a training job, its status changes to **Stopping** at which time billing ceases. An algorithm can delay termination in order to save model artifacts after which the job status changes to **Stopped**. For more information, see the `stop_training_job` method in the AWS SDK for Python (Boto3).

**To stop a training job**

1. Follow the View and Compare Experiments, Trials, and Trial Components (p. 850) procedure on this page until you open the **Describe Trial Component** tab.
2. At the upper-right side of the tab, choose **Stop training job**. The **Status** at the top left of the tab changes to **Stopped**.
3. To view the training time and billing time, choose **AWS Settings**.

Manage Your EFS Storage Volume

The first time a user on your team onboards to Amazon SageMaker Studio, Amazon SageMaker creates an Amazon Elastic File System (Amazon EFS) volume for the team. A home directory is created in the volume for each user who onboards to Studio as part of your team. Notebook files and data files are stored in these directories. Users don't have access to other team member's home directories.

**Important**

Don't delete the Amazon EFS volume. If you delete it, the domain will no longer function and all of your users will lose their work.

**To find your Amazon EFS volume**

1. From the SageMaker Studio Control Panel, under **Studio Summary**, find the **Studio ID**. The ID will be in the following format: `d-xxxxxxxxxxxxx`. 


2. Pass the Studio ID, as DomainId, to the `describe_domain` method.
3. In the response from `describe_domain`, note the value for the `HomeEfsFileSystemId` key. This is the Amazon EFS file system ID.
4. Open the Amazon EFS console. Make sure the AWS Region is the same Region that's used by Studio.
5. Under File systems, choose the file system ID from the previous step.
6. To verify that you've chosen the correct file system, select the Tags heading. The value corresponding to the `ManagedByAmazonSageMakerResource` key should match the Studio ID.

For information on how to access the Amazon EFS volume, see Using file systems in Amazon EFS.
To delete the Amazon EFS volume, see Deleting an Amazon EFS file system.

**Provide Feedback on SageMaker Studio**

Amazon SageMaker takes your feedback seriously. We encourage you to provide feedback.

To provide feedback
1. At the upper-right of SageMaker Studio, choose Feedback.
2. Choose a smiley emoji to let us know how satisfied you are with SageMaker Studio and add any feedback you'd care to share with us.
3. Decide whether to share your identity with us, then choose Submit.

**Update SageMaker Studio and Studio Apps**

The following procedures show you how to update Amazon SageMaker Studio and SageMaker Studio apps to their latest versions.

Topics
- Update SageMaker Studio (p. 121)
- Update Studio Apps (p. 122)

**Update SageMaker Studio**

To update Amazon SageMaker Studio to the latest release, you must shut down the JupyterServer app. Any unsaved notebook information is lost in the process. The user data in the Amazon EFS volume isn't touched.

After the JupyterServer app is shut down, you must reopen Studio through the SageMaker Studio Control Panel which creates a new version of the JupyterServer app.

You can shut down the JupyterServer app from the Studio Control Panel or from within Studio.

**Note**
A JupyterServer app is associated with a single Studio user. When you update the app for one user it doesn't effect other users.

To shut down the JupyterServer app from the Studio Control Panel
1. Choose your user name.
2. Under Apps, in the row displaying JupyterServer, choose Delete app.
3. Choose Yes, delete app.
4. Type delete in the confirmation box.
5. Choose **Delete**.

**To shut down the JupyterServer app from inside Studio**

1. (Optional) View the current Studio version number.
   a. Open the Studio Launcher. Choose **Amazon SageMaker Studio** in the top-left of Studio or use the keyboard shortcut Ctrl + Shift + L.
   b. Open **Utilities and files**.
   c. Choose **System terminal**.
   d. Run the following command:

   
   ```
   jupyter labextension list
   ```

   The version is specified similar to @amzn/sagemaker-ui v2.13.1.

2. On the top menu, choose **File** then **Shut Down**.

3. Choose one of the following options:
   - **Shutdown Server** – Shuts down the JupyterServer app. Terminal sessions, kernel sessions, SageMaker images, and instances aren’t shut down. These resources continue to accrue charges.
   - **Shutdown All** – Shuts down all apps, terminal sessions, kernel sessions, SageMaker images, and instances. These resources no longer accrue charges.

4. Close the window.

**Update Studio Apps**

To update an Amazon SageMaker Studio app to the latest release, you must shut down the corresponding KernelGateway app from the SageMaker Studio Control Panel.

**Note**

A KernelGateway app is associated with a single Studio user. When you update the app for one user it doesn’t effect other users.

**To shut down the KernelGateway app**

1. Choose your user name.
2. Under **Apps**, in the row displaying the **App name**, choose **Delete app**.
3. Choose **Yes, delete app**.
4. Type `delete` in the confirmation box.
5. Choose **Delete**.

---

**Amazon SageMaker Studio Pricing**

When the first member of your team onboards to Amazon SageMaker Studio, Amazon SageMaker creates an Amazon Elastic File System (Amazon EFS) volume for the team. In the SageMaker Studio Control Panel, when the Studio **Status** displays as **Ready** the Amazon EFS volume has been created.

When this member, or any member of the team, opens Studio, a home directory is created in the volume for the member. A storage charge is incurred for this directory. Subsequently, additional storage charges are incurred for the notebooks and data files stored in the member’s home directory. For pricing information on Amazon EFS, see Amazon EFS Pricing.

Additional costs are incurred when other operations are run inside Studio, for example, creating an Amazon SageMaker Autopilot job, running a notebook, running training jobs, and hosting a model.
For information on the costs associated with using Studio notebooks, see Usage Metering (p. 99).

For information about billing along with pricing examples, see Amazon SageMaker Pricing.
Use Amazon SageMaker Notebook Instances

An Amazon SageMaker notebook instance is a machine learning (ML) compute instance running the Jupyter Notebook App. SageMaker manages creating the instance and related resources. Use Jupyter notebooks in your notebook instance to prepare and process data, write code to train models, deploy models to SageMaker hosting, and test or validate your models.

SageMaker also provides sample notebooks that contain complete code walkthroughs. These walkthroughs show how to use SageMaker to perform common machine learning tasks. For more information, see Example Notebooks (p. 134).

This video shows you how to setup and use SageMaker notebook instances. (Length: 26:04)

This video is a deep dive on how to use SageMaker notebook instances. (Length: 16:44)

Topics

• Create a Notebook Instance (p. 124)
• Access Notebook Instances (p. 126)
• Customize a Notebook Instance Using a Lifecycle Configuration Script (p. 127)
• Example Notebooks (p. 134)
• Set the Notebook Kernel (p. 136)
• Associate Git Repositories with SageMaker Notebook Instances (p. 137)
• Notebook Instance Metadata (p. 144)
• Monitor Jupyter Logs in Amazon CloudWatch Logs (p. 144)

Create a Notebook Instance

An Amazon SageMaker notebook instance is a ML compute instance running the Jupyter Notebook App. SageMaker manages creating the instance and related resources. Use Jupyter notebooks in your notebook instance to prepare and process data, write code to train models, deploy models to SageMaker hosting, and test or validate your models.

To create a notebook instance, use either the SageMaker console or the CreateNotebookInstance API.

The notebook instance type you choose depends on how you use your notebook instance. You want to ensure that your notebook instance is not bound by memory, CPU, or IO. If you plan to load a dataset into memory on the notebook instance for exploration or preprocessing, we recommend that you choose an instance type with enough RAM memory for your dataset. This would require an instance with at least 16 GB of memory (xlarge or larger). If you plan to use the notebook for compute intensive preprocessing, we recommend you choose a compute-optimized instance such as a c4 or c5.

A best practice when using a SageMaker notebook is to use the notebook instance to orchestrate other AWS services. For example, you can use the notebook instance to manage large dataset processing by making calls to AWS Glue for ETL (extract, transform, and load) services or Amazon EMR for mapping and data reduction using Hadoop. You can use AWS services as temporary forms of computation or storage for your data.

You can store and retrieve your training and test data using an Amazon S3 bucket. You can then use SageMaker to train and build your model, so the instance type of your notebook would have no bearing on the speed of your model training and testing.
After receiving the request, SageMaker does the following:

- **Creates a network interface**—If you choose the optional VPC configuration, SageMaker creates the network interface in your VPC. It uses the subnet ID that you provide in the request to determine which Availability Zone to create the subnet in. SageMaker associates the security group that you provide in the request with the subnet. For more information, see Connect a Notebook Instance to Resources in a VPC (p. 1681).

- **Launches an ML compute instance**—SageMaker launches an ML compute instance in a SageMaker VPC. SageMaker performs the configuration tasks that allow it to manage your notebook instance, and if you specified your VPC, it enables traffic between your VPC and the notebook instance.

- **Installs Anaconda packages and libraries for common deep learning platforms**—SageMaker installs all of the Anaconda packages that are included in the installer. For more information, see Anaconda package list. In addition, SageMaker installs the TensorFlow and Apache MXNet deep learning libraries.

- **Attaches an ML storage volume**—SageMaker attaches an ML storage volume to the ML compute instance. You can use the volume as a working area to clean up the training dataset or to temporarily store validation, test, or other data. Choose any size between 5 GB and 16384 GB, in 1 GB increments, for the volume. The default is 5 GB. ML storage volumes are encrypted, so SageMaker can't determine the amount of available free space on the volume. Because of this, you can increase the volume size when you update a notebook instance, but you can't decrease the volume size. If you want to decrease the size of the ML storage volume in use, create a new notebook instance with the desired size.

Only files and data saved within the /home/ec2-user/SageMaker folder persist between notebook instance sessions. Files and data that are saved outside this directory are overwritten when the notebook instance stops and restarts. Each notebook instance's /tmp directory provides a minimum of 10 GB of storage in an instant store. An instance store is temporary, block-level storage that isn't persistent. When the instance is stopped or restarted, SageMaker deletes the directory's contents. This temporary storage is part of the root volume of the notebook instance.

- **Copies example Jupyter notebooks**—These Python code examples illustrate model training and hosting exercises using various algorithms and training datasets.

**To create a SageMaker notebook instance:**

2. Choose **Notebook instances**, then choose **Create notebook instance**.
3. On the **Create notebook instance** page, provide the following information:
   
   a. For **Notebook instance name**, type a name for your notebook instance.
   b. For **Notebook instance type**, choose an instance size suitable for your use case. For a list of supported instance types and quotas, see Amazon SageMaker Service Quotas.
   c. For **Elastic Inference**, choose an inference accelerator type to associate with the notebook instance if you plan to conduct inferences from the notebook instance, or choose none. For information about elastic inference, see Use Amazon SageMaker Elastic Inference (EI) (p. 1312).
   d. (Optional) **Additional configuration** lets advanced users create a shell script that can run when you create or start the instance. This script, called a lifecycle configuration script, can be used to set the environment for the notebook or to perform other functions. For information, see Customize a Notebook Instance Using a Lifecycle Configuration Script (p. 127).
   e. (Optional) **Additional configuration** also lets you specify the size, in GB, of the ML storage volume that is attached to the notebook instance. You can choose a size between 5 GB and 16,384 GB, in 1 GB increments. You can use the volume to clean up the training dataset or to temporarily store validation or other data.
   f. For **IAM role**, choose either an existing IAM role in your account that has the necessary permissions to access SageMaker resources or choose **Create a new role**. If you choose **Create a new role**, SageMaker creates an IAM role named AmazonSageMaker-ExecutionRole-YYYYMMDDHHmmSS. The AWS managed policy
AmazonSageMakerFullAccess is attached to the role. The role provides permissions that allow the notebook instance to call SageMaker and Amazon S3.

g. For Root access, to enable root access for all notebook instance users, choose Enable. To disable root access for users, choose Disable. If you enable root access, all notebook instance users have administrator privileges and can access and edit all files on it.

h. (Optional) Encryption key lets you encrypt data on the ML storage volume attached to the notebook instance using an AWS Key Management Service (AWS KMS) key. If you plan to store sensitive information on the ML storage volume, consider encrypting the information.

i. (Optional) Network lets you put your notebook instance inside a Virtual Private Cloud (VPC). A VPC provides additional security and restricts access to resources in the VPC from sources outside the VPC. For more information on VPCs, see Amazon VPC User Guide.

To add your notebook instance to a VPC:

i. Choose the VPC and a SubnetId.

ii. For Security Group, choose your VPC’s default security group.

iii. If you need your notebook instance to have internet access, enable direct internet access. For Direct internet access, choose Enable. Internet access can make your notebook instance less secure. For more information, see Connect a Notebook Instance to Resources in a VPC (p. 1681).

j. (Optional) To associate Git repositories with the notebook instance, choose a default repository and up to three additional repositories. For more information, see Associate Git Repositories with SageMaker Notebook Instances (p. 137).

k. Choose Create notebook instance.

In a few minutes, Amazon SageMaker launches an ML compute instance—in this case, a notebook instance—and attaches an ML storage volume to it. The notebook instance has a preconfigured Jupyter notebook server and a set of Anaconda libraries. For more information, see the CreateNotebookInstance API.

4. When the status of the notebook instance is InService, in the console, the notebook instance is ready to use. Choose Open Jupyter next to the notebook name to open the classic Jupyter dashboard.

You can choose Open JupyterLab to open the JupyterLab dashboard. The dashboard provides access to your notebook instance and sample SageMaker notebooks that contain complete code walkthroughs. These walkthroughs show how to use SageMaker to perform common machine learning tasks. For more information, see Example Notebooks (p. 134). For more information, see Control root access to a SageMaker notebook instance (p. 1613).

For more information about Jupyter notebooks, see The Jupyter notebook.

Access Notebook Instances

To access your Amazon SageMaker notebook instances, choose one of the following options:

- Use the console.

Choose Notebook instances. The console displays a list of notebook instances in your account. To open a notebook instance with a standard Jupyter interface, choose Open Jupyter for that instance. To open a notebook instance with a JupyterLab interface, choose Open JupyterLab for that instance.
The console uses your sign-in credentials to send a `CreatePresignedNotebookInstanceUrl` API request to SageMaker. SageMaker returns the URL for your notebook instance, and the console opens the URL in another browser tab and displays the Jupyter notebook dashboard.

**Note**
The URL that you get from a call to `CreatePresignedNotebookInstanceUrl` is valid only for 5 minutes. If you try to use the URL after the 5-minute limit expires, you are directed to the AWS Management Console sign-in page.

- **Use the API.**

To get the URL for the notebook instance, call the `CreatePresignedNotebookInstanceUrl` API and use the URL that the API returns to open the notebook instance.

Use the Jupyter notebook dashboard to create and manage notebooks and to write code. For more information about Jupyter notebooks, see [http://jupyter.org/documentation.html](http://jupyter.org/documentation.html).

## Customize a Notebook Instance Using a Lifecycle Configuration Script

To install packages or sample notebooks on your notebook instance, configure networking and security for it, or otherwise use a shell script to customize it, use a lifecycle configuration. A lifecycle configuration provides shell scripts that run only when you create the notebook instance or whenever you start one. When you create a notebook instance, you can create a new lifecycle configuration and the scripts it uses or apply one that you already have.

You can also use a lifecycle configuration script to access AWS services from your notebook. For example, you can create a script that lets you use your notebook to control other AWS resources, such as an Amazon EMR instance.

We maintain a public repository of notebook lifecycle configuration scripts that address common use cases for customizing notebook instances at [https://github.com/aws-samples/amazon-sagemaker-notebook-instance-lifecycle-configuration-samples](https://github.com/aws-samples/amazon-sagemaker-notebook-instance-lifecycle-configuration-samples).

**Note**
Each script has a limit of 16384 characters.
The value of the `$PATH` environment variable that is available to both scripts is `/usr/local/sbin:/usr/local/bin:/usr/bin:/usr/sbin:/sbin:/bin`. The working directory, which is the value of the `$PWD` environment variable, is `/.`.

View CloudWatch Logs for notebook instance lifecycle configurations in log group `/aws/sagemaker/NotebookInstances` in log stream `[notebook-instance-name]/LifecycleConfigHook`. 

127
Scripts cannot run for longer than 5 minutes. If a script runs for longer than 5 minutes, it fails and the notebook instance is not created or started. To help decrease the run time of scripts, try the following:

- Cut down on necessary steps. For example, limit which conda environments in which to install large packages.
- Run tasks in parallel processes.
- Use the nohup command in your script.

To create a lifecycle configuration

1. For Lifecycle configuration - Optional, choose Create a new lifecycle configuration.
2. For Name, type a name using alphanumeric characters and "-", but no spaces. The name can have a maximum of 63 characters.
3. (Optional) To create a script that runs when you create the notebook and every time you start it, choose Start notebook.
4. In the Start notebook editor, type the script.
5. (Optional) To create a script that runs only once, when you create the notebook, choose Create notebook.
6. In the Create notebook editor, type the script configure networking.
7. Choose Create configuration.

You can see a list of notebook instance lifecycle configurations you previously created by choosing Lifecycle configuration in the SageMaker console. From there, you can view, edit, delete existing lifecycle configurations. You can create a new notebook instance lifecycle configuration by choosing Create configuration. These notebook instance lifecycle configurations are available when you create a new notebook instance.

Lifecycle Configuration Best Practices

The following are best practices for using lifecycle configurations:

- Lifecycle configurations run as the root user. If your script makes any changes within the /home/ec2-user/SageMaker directory, (for example, installing a package with pip), use the command sudo -u ec2-user to run as the ec2-user user. This is the same user that Amazon SageMaker runs as.
- SageMaker notebook instances use conda environments to implement different kernels for Jupyter notebooks. If you want to install packages that are available to one or more notebook kernels, enclose the commands to install the packages with conda environment commands that activate the conda environment that contains the kernel where you want to install the packages.

For example, if you want to install a package only for the python3 environment, use the following code:

```bash
#!/bin/bash
sudo -u ec2-user -i <<'EOF'
# This will affect only the Jupyter kernel called "conda_python3".
source activate python3

# Replace myPackage with the name of the package you want to install.
pip install myPackage
# You can also perform "conda install" here as well.
source deactivate
EOF
```
If you want to install a package in all conda environments in the notebook instance, use the following code:

```
#!/bin/bash
sudo -u ec2-user -i <<-EOF
# Note that "base" is special environment name, include it there as well.
for env in base /home/ec2-user/anaconda3/envs/*; do
    source /home/ec2-user/anaconda3/bin/activate $(basename "$env")

    # Installing packages in the Jupyter system environment can affect stability of your SageMaker Notebook Instance. You can remove this check if you'd like to install Jupyter extensions, etc.
    if [ $env = 'JupyterSystemEnv' ]; then
        continue
    fi

    # Replace myPackage with the name of the package you want to install.
    pip install --upgrade --quiet myPackage

    # You can also perform "conda install" here as well.
    source /home/ec2-user/anaconda3/bin/deactivate
done
EOF
```

- You must store all conda environments in the default environments folder (/home/user/anaconda3/envs).

**Important**
When you create or change a script, we recommend that you use a text editor that provides Unix-style line breaks, such as the text editor available in the console when you create a notebook. Copying text from a non-Linux operating system might introduce incompatible line breaks and result in an unexpected error.

### Install External Libraries and Kernels in Notebook Instances

Amazon SageMaker notebook instances come with multiple environments already installed. These environments contain Jupyter kernels and Python packages including: scikit, Pandas, NumPy, TensorFlow, and MXNet. These environments, along with all files in the sample-notebooks folder, are refreshed when you stop and start a notebook instance. You can also install your own environments that contain your choice of packages and kernels.

The different Jupyter kernels in Amazon SageMaker notebook instances are separate conda environments. For information about conda environments, see Managing environments in the Conda documentation.

Install custom environments and kernels on the notebook instance's Amazon EBS volume. This ensures that they persist when you stop and restart the notebook instance, and that any external libraries you install are not updated by SageMaker. To do that, use a lifecycle configuration that includes both a script that runs when you create the notebook instance (on-create) and a script that runs each time you restart the notebook instance (on-start). For more information about using notebook instance lifecycle configurations, see Customize a Notebook Instance Using a Lifecycle Configuration Script (p. 127).
There is a GitHub repository that contains sample lifecycle configuration scripts at Amazon SageMaker Notebook Instance Lifecycle Config Samples.

The examples at https://github.com/aws-samples/amazon-sagemaker-notebook-instance-lifecycle-config-samples/blob/master/scripts/persistent-conda-ebs/on-create.sh and https://github.com/aws-samples/amazon-sagemaker-notebook-instance-lifecycle-config-samples/blob/master/scripts/persistent-conda-ebs/on-start.sh show the best practice for installing environments and kernels on a notebook instance. The on-create script installs the ipykernel library so that you can use create custom environments as Jupyter kernels, and then uses pip install and conda install to install libraries. You can adapt the script to create custom environments and install libraries that you want. SageMaker does not update these libraries when you stop and restart the notebook instance, so you can ensure that your custom environment has specific versions of libraries that you want. The on-start script installs any custom environments that you create as Jupyter kernels, so that they appear in the dropdown list in the Jupyter New menu.

**Package installation tools**

SageMaker notebooks support the following package installation tools:

- conda install
- pip install

You can install packages using the following methods:

- Lifecycle configuration scripts.
  - For example scripts, see Amazon SageMaker Notebook Instance Lifecycle Config Samples. For more information on lifecycle configuration, see Customize a Notebook Instance Using a Lifecycle Configuration Script.
- Notebooks – The following commands are supported.
  - %conda install
  - %pip install
- The Jupyter terminal – You can install packages using pip and conda directly.

From within a notebook you can use the system command syntax (lines starting with !) to install packages, for example, !pip install and !conda install. More recently, new commands have been added to IPython: %pip and %conda. These commands are the recommended way to install packages from a notebook as they correctly take into account the activate environment or interpreter being used. For more information, see Add %pip and %conda magic functions.

**Conda**

Conda is an open source package management system and environment management system, which can install packages and their dependencies. SageMaker supports using Conda with either of the two main channels, the default channel, and the conda-forge channel. For more information, see Conda channels. The conda-forge channel is a community channel where contributors can upload packages.

**Note**

Due to how Conda resolves the dependency graph, installing packages from conda-forge can take significantly longer (in the worst cases, upwards of 10 minutes).

The Deep Learning AMI comes with many conda environments and many packages preinstalled. Due to the number of packages preinstalled, finding a set of packages that are guaranteed to be compatible is difficult. You may see a warning “The environment is inconsistent, please check the package plan carefully”. Despite this warning, SageMaker ensures that all the SageMaker provided environments are correct. SageMaker cannot guarantee that any user installed packages will function correctly.
Conda has two methods for activating environments: conda activate/deactivate, and source activate/deactivate. For more information, see Should I use ’conda activate’ or ’source activate’ in Linux.

SageMaker supports moving Conda environments onto the Amazon EBS volume, which is persisted when the instance is stopped. The environments aren’t persisted when the environments are installed to the root volume, which is the default behavior. For an example lifecycle script, see persistent-conda-ebs.

**Supported conda operations (see note at the bottom of this topic)**

- conda install of a package in a single environment
- conda install of a package in all environments
- conda install of a R package in the R environment
- Installing a package from the main conda repository
- Installing a package from conda-forge
- Changing the Conda install location to use EBS
- Supporting both conda activate and source activate

**Pip**

Pip is the de facto tool for installing and managing Python packages. Pip searches for packages on the Python Package Index (PyPI) by default. Unlike Conda, pip doesn’t have built in environment support, and is not as thorough as Conda when it comes to packages with native/system library dependencies. Pip can be used to install packages in Conda environments.

You can use alternative package repositories with pip instead of the PyPI. For an example lifecycle script, see on-start.sh.

**Supported pip operations (see note at the bottom of this topic)**

- Using pip to install a package without an active conda environment (install packages system wide)
- Using pip to install a package in a conda environment
- Using pip to install a package in all conda environments
- Changing the pip install location to use EBS
- Using an alternative repository to install packages with pip

**Unsupported**

SageMaker aims to support as many package installation operations as possible. However, if the packages were installed by SageMaker or DLAMI, and you use the following operations on these packages, it might make your notebook instance unstable:

- Uninstalling
- Downgrading
- Upgrading

We do not provide support for installing packages via yum install or installing R packages from CRAN.

Due to potential issues with network conditions or configurations, or the availability of Conda or PyPi, we cannot guarantee that packages will install in a fixed or deterministic amount of time.

**Note**

We cannot guarantee that a package installation will be successful. Attempting to install a package in an environment with incompatible dependencies can result in a failure. In such a case you should contact the library maintainer to see if it is possible to update the package.
dependencies. Alternatively you can attempt to modify the environment in such a way as to allow the installation. This modification however will likely mean removing or updating existing packages, which means we can no longer guarantee stability of this environment.

Notebook Instance Software Updates

Amazon SageMaker periodically tests and releases software that is installed on notebook instances. This includes:

- Kernel updates
- Security patches
- AWS SDK updates
- Amazon SageMaker Python SDK updates
- Open source software updates

SageMaker does not automatically update software on a notebook instance when it is in service. To ensure that you have the most recent software updates, stop and restart your notebook instance, either in the SageMaker console or by calling `StopNotebookInstance`.

You can also manually update software installed on your notebook instance while it is running by using update commands in a terminal or in a notebook.

**Note**

Updating kernels and some packages might depend on whether root access is enabled for the notebook instance. For more information, see Control root access to a SageMaker notebook instance (p. 1613).

Notebook instances do not notify you if you are running outdated software. You can check the Personal Health Dashboard or the security bulletin at https://aws.amazon.com/security/security-bulletins/ for updates.

Control an Amazon EMR Spark Instance Using a Notebook

You can use a notebook instance created with a custom lifecycle configuration script to access AWS services from your notebook. For example, you can create a script that lets you use your notebook with Sparkmagic to control other AWS resources, such as an Amazon EMR instance. You can then use the Amazon EMR instance to process your data instead of running the data analysis on your notebook. This allows you to create a smaller notebook instance because you won’t use the instance to process data. This is helpful when you have large datasets that would require a large notebook instance to process the data.

The process requires three procedures using the Amazon SageMaker console:

- Create the Amazon EMR Spark instance
- Create the Jupyter Notebook
- Test the notebook-to-Amazon EMR connection

**To create an Amazon EMR Spark instance that can be controlled from a notebook using Sparkmagic**

1. Open the Amazon EMR console at https://console.aws.amazon.com/elasticmapreduce/.
2. In the navigation pane, choose **Create cluster**.
3. On the **Create Cluster - Quick Options** page, under **Software configuration**, choose Spark: Spark 2.4.4 on Hadoop 2.8.5 YARN with Ganglia 3.7.2 and Zeppelin 0.8.2.
4. Set additional parameters on the page and then choose **Create cluster**.
5. On the **Cluster** page, choose the cluster name that you created. Note the **Master Public DNS**, the **EMR master's security group**, and the VPC name and subnet ID where the EMR cluster was created. You will use these values when you create a notebook.

To create a notebook that uses Sparkmagic to control an Amazon EMR Spark instance

2. In the navigation pane, under **Notebook instances**, choose **Create notebook**.
3. Enter the notebook instance name and choose the instance type.
4. Choose **Additional configuration**, then, under **Lifecycle configuration**, choose **Create a new lifecycle configuration**.
5. Add the following code to the lifecycle configuration script:

```bash
# OVERVIEW
# This script connects an Amazon EMR cluster to an Amazon SageMaker notebook instance that uses Sparkmagic.
# Note that this script will fail if the Amazon EMR cluster's master node IP address is not reachable.
# 1. Ensure that the EMR master node IP is resolvable from the notebook instance.
#    One way to accomplish this is to have the notebook instance and the Amazon EMR cluster in the same subnet.
# 2. Ensure the EMR master node security group provides inbound access from the notebook instance security group.
#    Type   - Protocol - Port - Source
#    Custom TCP - TCP    - 8998 - $NOTEBOOK_SECURITY_GROUP
# 3. Ensure the notebook instance has internet connectivity to fetch the SparkMagic example config.

# PARAMETERS
EMR_MASTER_IP=your.emr.master.ip

cd /home/ec2-user/.sparkmagic

echo "Fetching Sparkmagic example config from GitHub..."
wget https://raw.githubusercontent.com/jupyter-incubator/sparkmagic/master/sparkmagic/example_config.json

echo "Replacing EMR master node IP in Sparkmagic config..."
sed -i -- "s/localhost/$EMR_MASTER_IP/g" example_config.json
mv example_config.json config.json

echo "Sending a sample request to Livy.."
curl "$EMR_MASTER_IP:8998/sessions"
```
6. In the **PARAMETERS** section of the script, replace `your.emr.master.ip` with the Master Public DNS name for the Amazon EMR instance.
7. Choose **Create configuration**.
8. On the **Create notebook** page, choose **Network - optional**.
9. Choose the VPC and subnet where the Amazon EMR instance is located.
10. Choose the security group used by the Amazon EMR master node.
11. Choose Create notebook instance.

While the notebook instance is being created, the status is Pending. After the instance has been created and the lifecycle configuration script has successfully run, the status is InService.

Note
If the notebook instance can't connect to the Amazon EMR instance, SageMaker can't create the notebook instance. The connection can fail if the Amazon EMR instance and notebook are not in the same VPC and subnet, if the Amazon EMR master security group is not used by the notebook, or if the Master Public DNS name in the script is incorrect.

To test the connection between the Amazon EMR instance and the notebook
1. When the status of the notebook is InService, choose Open Jupyter to open the notebook.
2. Choose New, then choose Sparkmagic (PySpark).
3. In the code cell, enter %%info and then run the cell.

The output should be similar to the following

```markdown
Current session config: {'driverMemory': '1000M', 'executorCores': 2, 'kind': 'pyspark'}
No active sessions.
```

Example Notebooks

Your notebook instance contains example notebooks provided by Amazon SageMaker. The example notebooks contain code that shows how to apply machine learning solutions by using SageMaker. Notebook instances use the nbexamples Jupyter extension, which enables you to view a read-only version of an example notebook or create a copy of it so that you can modify and run it. For more information about the nbexamples extension, see https://github.com/danielballan/nbexamples. For information about example notebooks for SageMaker Studio, see Use Amazon SageMaker Studio Notebooks (p. 84).

Note
Example notebooks typically download datasets from the internet. If you disable SageMaker-provided internet access when you create you notebook instance, example notebooks might not work. For more information, see Connect a Notebook Instance to Resources in a VPC (p. 1681).

Use or View Example Notebooks in Jupyter Classic

To view or use the example notebooks in the classic Jupyter view, choose the SageMaker Examples tab.
Use or View Example Notebooks in Jupyterlab

To view or use the example notebooks in the Jupyterlab view, choose the examples icon in the left navigation panel.
Set the Notebook Kernel

Amazon SageMaker provides several kernels for Jupyter that provide support for Python 2 and 3, Apache MXNet, TensorFlow, and PySpark. To set a kernel for a new notebook in the Jupyter notebook dashboard, choose New, and then choose the kernel from the list.
You can also create a custom kernel that you can use in your notebook instance. For information, see Install External Libraries and Kernels in Notebook Instances (p. 129).

## Associate Git Repositories with SageMaker Notebook Instances

Associate Git repositories with your notebook instance to save your notebooks in a source control environment that persists even if you stop or delete your notebook instance. You can associate one default repository and up to three additional repositories with a notebook instance. The repositories can be hosted in AWS CodeCommit, GitHub, or on any other Git server. Associating Git repositories with your notebook instance can be useful for:

- **Persistence** - Notebooks in a notebook instance are stored on durable Amazon EBS volumes, but they do not persist beyond the life of your notebook instance. Storing notebooks in a Git repository enables you to store and use notebooks even if you stop or delete your notebook instance.
- **Collaboration** - Peers on a team often work on machine learning projects together. Storing your notebooks in Git repositories allows peers working in different notebook instances to share notebooks and collaborate on them in a source-control environment.
- **Learning** - Many Jupyter notebooks that demonstrate machine learning techniques are available in publicly hosted Git repositories, such as on GitHub. You can associate your notebook instance with a repository to easily load Jupyter notebooks contained in that repository.

There are two ways to associate a Git repository with a notebook instance:

- **Add a Git repository as a resource in your Amazon SageMaker account.** Then, to access the repository, you can specify an AWS Secrets Manager secret that contains credentials. That way, you can access repositories that require authentication.
- **Associate a public Git repository that is not a resource in your account.** If you do this, you cannot specify credentials to access the repository.

### Topics
- Add a Git Repository to Your Amazon SageMaker Account (p. 138)
- Create a Notebook Instance with an Associated Git Repository (p. 140)
- Associate a CodeCommit Repository in a Different AWS Account with a Notebook Instance (p. 141)
- Use Git Repositories in a Notebook Instance (p. 142)
Add a Git Repository to Your Amazon SageMaker Account

To manage your GitHub repositories, easily associate them with your notebook instances, and associate credentials for repositories that require authentication, add the repositories as resources in your Amazon SageMaker account. You can view a list of repositories that are stored in your account and details about each repository in the SageMaker console and by using the API.

You can add Git repositories to your SageMaker account in the SageMaker console or by using the AWS CLI.

**Note**
You can use the SageMaker API `CreateCodeRepository` to add Git repositories to your SageMaker account, but step-by-step instructions are not provided here.

Add a Git Repository to Your SageMaker Account (Console)

To add a Git repository as a resource in your SageMaker account

2. Choose *Git repositories*, then choose *Add repository*.
3. To add an CodeCommit repository, choose *AWS CodeCommit*. To add a GitHub or other Git-based repository, choose *GitHub/Other Git-based repo*.

To add an existing CodeCommit repository

1. Choose *Use existing repository*.
2. For *Repository*, choose a repository from the list.
3. Enter a name to use for the repository in SageMaker. The name must be 1 to 63 characters. Valid characters are a-z, A-Z, 0-9, and - (hyphen).
4. Choose *Add repository*.

To create a new CodeCommit repository

1. Choose *Create new repository*.
2. Enter a name for the repository that you can use in both CodeCommit and SageMaker. The name must be 1 to 63 characters. Valid characters are a-z, A-Z, 0-9, and - (hyphen).
3. Choose *Create repository*.

To add a Git repository hosted somewhere other than CodeCommit

1. Choose *GitHub/Other Git-based repo*.
2. Enter a name of up to 63 characters. Valid characters include alpha-numeric characters, a hyphen (-), and 0-9.
3. Enter the URL for the repository. Do not provide a user name in the URL. Add the username and password in AWS Secrets Manager as described in the next step.
4. For *Git credentials*, choose the credentials to use to authenticate to the repository. This is necessary only if the Git repository is private.

**Note**
If you have two-factor authentication enabled for your Git repository, use a personal access token generated by your Git service provider instead of a password.
a. To use an existing AWS Secrets Manager secret, choose **Use existing secret**, and then choose a secret from the list. For information about creating and storing a secret, see *Creating a Basic Secret* in the *AWS Secrets Manager User Guide*. The name of the secret you use must contain the string `sagemaker`.

**Note**
The secret must have a staging label of `AWSCURRENT` and must be in the following format:

```json
{"username": "UserName", "password": "Password"}
```
For GitHub repositories, we recommend using a personal access token instead of your account password. For information, see [https://help.github.com/articles/creating-a-personal-access-token-for-the-command-line/](https://help.github.com/articles/creating-a-personal-access-token-for-the-command-line/).

b. To create a new AWS Secrets Manager secret, choose **Create secret**, enter a name for the secret, and then enter the username and password to use to authenticate to the repository. The name for the secret must contain the string `sagemaker`.

**Note**
The IAM role you use to create the secret must have the `secretsmanager:GetSecretValue` permission in its IAM policy.
The secret must have a staging label of `AWSCURRENT` and must be in the following format:

```json
{"username": "UserName", "password": "Password"}
```
For GitHub repositories, we recommend using a personal access token instead of your account password.

c. To not use any credentials, choose **No secret**.

5. Choose **Create secret**.

---

**Add a Git Repository to Your Amazon SageMaker Account (CLI)**

Use the `create-code-repository` AWS CLI command. Specify a name for the repository as the value of the `code-repository-name` argument. The name must be 1 to 63 characters. Valid characters are a-z, A-Z, 0-9, and - (hyphen). Also specify the following:

- The default branch
- The URL of the Git repository

**Note**
Do not provide a user name in the URL. Add the username and password in AWS Secrets Manager as described in the next step.

- The Amazon Resource Name (ARN) of an AWS Secrets Manager secret that contains the credentials to use to authenticate the repository as the value of the `git-config` argument

For information about creating and storing a secret, see *Creating a Basic Secret* in the *AWS Secrets Manager User Guide*. The following command creates a new repository named `MyRepository` in your Amazon SageMaker account that points to a Git repository hosted at https://github.com/myprofile/my-repo".

For Linux, OS X, or Unix:

```
aws sagemaker create-code-repository
  --code-repository-name "MyRepository"
```
For Windows:

```
aws sagemaker create-code-repository ^
    --code-repository-name "MyRepository" ^
```

**Note**
The secret must have a staging label of AWSCURRENT and must be in the following format:

```
{"username": UserName, "password": Password}
```

For GitHub repositories, we recommend using a personal access token instead of your account password.

---

Create a Notebook Instance with an Associated Git Repository

You can associate Git repositories with a notebook instance when you create the notebook instance by using the AWS Management Console, or the AWS CLI. If you want to use a CodeCommit repository that is in a different AWS account than the notebook instance, set up cross-account access for the repository. For information, see Associate a CodeCommit Repository in a Different AWS Account with a Notebook Instance (p. 141).

**Topics**

- Create a Notebook Instance with an Associated Git Repository (Console) (p. 140)
- Create a Notebook Instance with an Associated Git Repository (CLI) (p. 141)

Create a Notebook Instance with an Associated Git Repository (Console)

To create a notebook instance and associate Git repositories in the Amazon SageMaker console

1. Follow the instructions at Step 2: Create an Amazon SageMaker Notebook Instance (p. 60).
2. For Git repositories, choose Git repositories to associate with the notebook instance.
   a. For Default repository, choose a repository that you want to use as your default repository. SageMaker clones this repository as a subdirectory in the Jupyter startup directory at /home/ec2-user/SageMaker. When you open your notebook instance, it opens in this repository. To choose a repository that is stored as a resource in your account, choose its name from the list. To add a new repository as a resource in your account, choose Add a repository to SageMaker (opens the Add repository flow in a new window) and then follow the instructions at Create a Notebook Instance with an Associated Git Repository (Console) (p. 140). To clone a public repository that is not stored in your account, choose Clone a public Git repository to this notebook instance only, and then specify the URL for that repository.
   b. For Additional repository 1, choose a repository that you want to add as an additional directory. SageMaker clones this repository as a subdirectory in the Jupyter startup directory at /home/ec2-user/SageMaker. To choose a repository that is stored as a resource in your account, choose its name from the list. To add a new repository as a resource in your account, choose Add a repository to SageMaker (opens the Add repository flow in a new window) and then follow the instructions at Create a Notebook Instance with an Associated Git Repository (Console) (p. 140). To clone a repository that is not stored in your account, choose Clone
a public Git repository to this notebook instance only, and then specify the URL for that repository.

Repeat this step up to three times to add up to three additional repositories to your notebook instance.

Create a Notebook Instance with an Associated Git Repository (CLI)

To create a notebook instance and associate Git repositories by using the AWS CLI, use the create-notebook-instance command as follows:

- Specify the repository that you want to use as your default repository as the value of the default-code-repository argument. Amazon SageMaker clones this repository as a subdirectory in the Jupyter startup directory at /home/ec2-user/SageMaker. When you open your notebook instance, it opens in this repository. To use a repository that is stored as a resource in your SageMaker account, specify the name of the repository as the value of the default-code-repository argument. To use a repository that is not stored in your account, specify the URL of the repository as the value of the default-code-repository argument.

- Specify up to three additional repositories as the value of the additional-code-repositories argument. SageMaker clones this repository as a subdirectory in the Jupyter startup directory at /home/ec2-user/SageMaker, and the repository is excluded from the default repository by adding it to the .git/info/exclude directory of the default repository. To use repositories that are stored as resources in your SageMaker account, specify the names of the repositories as the value of the additional-code-repositories argument. To use repositories that are not stored in your account, specify the URLs of the repositories as the value of the additional-code-repositories argument.

For example, the following command creates a notebook instance that has a repository named MyGitRepo, that is stored as a resource in your SageMaker account, as a default repository, and an additional repository that is hosted on GitHub:

```bash
aws sagemaker create-notebook-instance \
  --notebook-instance-name "MyNotebookInstance" \
  --instance-type "ml.t2.medium" \
  --role-arn "arn:aws:iam::012345678901:role/service-role/AmazonSageMaker-ExecutionRole-20181129T121390" \
  --default-code-repository "MyGitRepo" \
  --additional-code-repositories "https://github.com/myprofile/my-other-repo"
```

**Note**

If you use an AWS CodeCommit repository that does not contain "SageMaker" in its name, add the codecommit:GitPull and codecommit:GitPush permissions to the role that you pass as the role-arn argument to the create-notebook-instance command. For information about how to add permissions to a role, see Adding and Removing IAM Policies in the AWS Identity and Access Management User Guide.

Associate a CodeCommit Repository in a Different AWS Account with a Notebook Instance

To associate a CodeCommit repository in a different AWS account with your notebook instance, set up cross-account access for the CodeCommit repository.
To set up cross-account access for a CodeCommit repository and associate it with a notebook instance:

1. In the AWS account that contains the CodeCommit repository, create an IAM policy that allows access to the repository from users in the account that contains your notebook instance. For information, see Step 1: Create a Policy for Repository Access in AccountA in the CodeCommit User Guide.

2. In the AWS account that contains the CodeCommit repository, create an IAM role, and attach the policy that you created in the previous step to that role. For information, see Step 2: Create a Role for Repository Access in AccountA in the CodeCommit User Guide.

3. Create a profile in the notebook instance that uses the role that you created in the previous step:
   
a. Open the notebook instance.
b. Open a terminal in the notebook instance.
c. Edit a new profile by typing the following in the terminal:

```bash
vi /home/ec2-user/.aws/config
```

d. Edit the file with the following profile information:

```ini
[profile CrossAccountAccessProfile]
region = us-west-2
role_arn = arn:aws:iam::CodeCommitAccount:role/CrossAccountRepositoryContributorRole
credential_source=Ec2InstanceMetadata
output = json
```

Where `CodeCommitAccount` is the account that contains the CodeCommit repository, `CrossAccountAccessProfile` is the name of the new profile, and `CrossAccountRepositoryContributorRole` is the name of the role you created in the previous step.

4. On the notebook instance, configure git to use the profile you created in the previous step:

   a. Open the notebook instance.
   b. Open a terminal in the notebook instance.
   c. Edit the Git configuration file typing the following in the terminal:

```bash
vi /home/ec2-user/.gitconfig
```

d. Edit the file with the following profile information:

```ini
[credential]
helper = !aws codecommit credential-helper --
profile CrossAccountAccessProfile #@
UseHttpPath = true
```

Where `CrossAccountAccessProfile` is the name of the profile that you created in the previous step.

Use Git Repositories in a Notebook Instance

When you open a notebook instance that has Git repositories associated with it, it opens in the default repository, which is installed in your notebook instance directly under `/home/ec2-user/SageMaker`. 
You can open and create notebooks, and you can manually run Git commands in a notebook cell. For example:

```
!git pull origin master
```

To open any of the additional repositories, navigate up one folder. The additional repositories are also installed as directories under `/home/ec2-user/SageMaker`.

If you open the notebook instance with a JupyterLab interface, the `jupyter-git` extension is installed and available to use. For information about the `jupyter-git` extension for JupyterLab, see [https://github.com/jupyterlab/jupyterlab-git](https://github.com/jupyterlab/jupyterlab-git).

When you open a notebook instance in JupyterLab, you see the git repositories associated with it on the left menu:

You can use the `jupyter-git` extension to manage git visually, instead of using the command line:
When you create a notebook instance, Amazon SageMaker creates a JSON file on the instance at the location `/opt/ml/metadata/resource-metadata.json` that contains the `ResourceName` and `ResourceArn` of the notebook instance. You can access this metadata from anywhere within the notebook instance, including in lifecycle configurations. For information about notebook instance lifecycle configurations, see Customize a Notebook Instance Using a Lifecycle Configuration Script (p. 127).

The `resource-metadata.json` file has the following structure:

```json
{
    "ResourceArn": "NotebookInstanceArn",
    "ResourceName": "NotebookInstanceName"
}
```

You can use this metadata from within the notebook instance to get other information about the notebook instance. For example, the following commands get the tags associated with the notebook instance:

```
NOTEBOOK_ARN=$(jq '.ResourceArn' /opt/ml/metadata/resource-metadata.json --raw-output)
aws sagemaker list-tags --resource-arn $NOTEBOOK_ARN
```

The output looks like the following:

```json
{
    "Tags": [
        {
            "Key": "test",
            "Value": "true"
        }
    ]
}
```

### Monitor Jupyter Logs in Amazon CloudWatch Logs

Jupyter logs include important information such as events, metrics, and health information that provide actionable insights when running Amazon SageMaker notebooks. By importing Jupyter logs into
CloudWatch Logs, customers can use CloudWatch Logs to detect anomalous behaviors, set alarms, and discover insights to keep the SageMaker notebooks running more smoothly. You can access the logs even when the Amazon EC2 instance that hosts the notebook is unresponsive, and use the logs to troubleshoot the unresponsive notebook. Sensitive information such as AWS account IDs, secret keys, and authentication tokens in presigned URLs are removed so that customers can share logs without leaking private information.

**To view Jupyter logs for a notebook instance:**

2. Choose **Notebook instances**.
3. In the list of notebook instances, choose the notebook instance for which you want to view Jupyter logs.
4. Under **Monitor** on the notebook instance details page, choose **View logs**.
5. In the CloudWatch console, choose the log stream for your notebook instance. Its name is in the form `NotebookInstanceName/jupyter.log`.

For more information about monitoring CloudWatch logs for SageMaker, see Log Amazon SageMaker Events with Amazon CloudWatch (p. 1712).
Automate model development with Amazon SageMaker Autopilot

Amazon SageMaker Autopilot is a feature-set that automates key tasks of an automatic machine learning (AutoML) process. It explores your data, selects the algorithms relevant to your problem type, and prepares the data to facilitate model training and tuning. It simplifies your machine learning experience by automating these key tasks that constitute an AutoML process. It ranks all of the optimized models tested by their performance. It finds the best performing model that you can deploy at a fraction of the time normally required.

You get full visibility into how the data was wrangled and how the models were selected, trained and tuned for each of the candidates tested. This is provided by notebooks that Autopilot generates for each trial that contain the code used to explore the data and find the best candidates. The notebooks also provide educational tools that enable you to learn about and conduct your own ML experiments. You can learn about the impact of various inputs and trade-offs made in experiments by examining the various data exploration and candidate generation notebooks exposed by Autopilot. You can also conduct further experiments on the higher performing candidates by making your own modifications to the notebooks and rerunning them.

The following graphic outlines the principal tasks of an AutoML process managed by Autopilot.

You can use Autopilot in different ways: on autopilot (hence the name) or with various degrees of human guidance, without code through Amazon SageMaker Studio, or with code using one of the AWS SDKs. Autopilot currently supports regression and binary and multiclass classification. It also only supports tabular data formatted in files with comma-separated values.

With Amazon SageMaker, you pay only for what you use. Building, training, and deploying ML models is billed by the second, with no minimum fees and no upfront commitments. For more information about the cost of using SageMaker, see Amazon SageMaker Pricing.

Topics
- Get started with Amazon SageMaker Autopilot (p. 147)
- Create an Amazon SageMaker Autopilot experiment (p. 149)
- Amazon SageMaker Autopilot problem types and algorithm support (p. 151)
- Amazon SageMaker Autopilot notebooks generated to manage AutoML tasks (p. 153)
- Configure inference output in Autopilot-generated containers (p. 153)
- Amazon SageMaker Autopilot quotas (p. 155)
- API reference guide for Amazon SageMaker Autopilot (p. 157)
Get started with Amazon SageMaker Autopilot

Amazon SageMaker Autopilot provides samples, videos, and tutorials to get started with Amazon SageMaker Autopilot

Topics

- Samples: Explore modeling with Amazon SageMaker Autopilot (p. 147)
- Videos: Use Autopilot to automate and explore the machine learning process (p. 147)
- Tutorials: Get started with Amazon SageMaker Autopilot (p. 148)

Samples: Explore modeling with Amazon SageMaker Autopilot

Amazon SageMaker Autopilot provides the following sample notebooks.

- Direct marketing with Amazon SageMaker Autopilot: This notebook demonstrates how uses the Bank Marketing Data Set to predict whether a customer will enroll for a term deposit at a bank. You can use Autopilot on this dataset to get the most accurate ML pipeline by exploring options contained in various candidate pipelines. Autopilot generates each candidate in a two step procedure. The first step performs automated feature engineering on the dataset. The second step trains and tunes an algorithm to produce a model. The notebook contains instructions on how to train the model as well as how to deploy the model to perform batch inference using the best candidates.

- Customer Churn Prediction with Amazon SageMaker Autopilot: This notebook describes using machine learning for the automated identification of unhappy customers, also known as customer churn prediction. The sample shows how to analyze a publicly available dataset and perform feature engineering on it. Next it shows how to tune a model by selecting the best performing pipeline along with the optimal hyperparameters for the training algorithm. Finally it shows how to deploy the model to a hosted endpoint and evaluate its predictions against ground truth. ML models rarely give perfect predictions though, so this notebook is also about how to incorporate the relative costs of prediction mistakes when determining the financial outcome of using ML.

- Top Candidates Customer Churn Prediction with Amazon SageMaker Autopilot and Batch Transform (Python SDK): This notebook also describes using machine learning for the automated identification of unhappy customers, also known as customer churn prediction. This notebook demonstrate how to configure the model to obtain the inference probability, select the top N models, and make Batch Transform on a hold-out test set for evaluation.

  Note
  This notebook works with SageMaker Python SDK >= 1.65.1 released on 6/19/2020.

- Bringing your own data processing code to SageMaker Autopilot: This notebook demonstrates how to incorporate and deploy custom data processing code when using Amazon SageMaker Autopilot. It adds a custom feature selection step to remove irrelevant variables to a Autopilot job. It then shows how to deploy both the custom processing code and models generated by Autopilot on a real-time endpoint and, alternatively, for batch processing.

Videos: Use Autopilot to automate and explore the machine learning process

Here is a video series that provides a tour of Amazon SageMaker Autopilot capabilities using Studio. They show how to start an AutoML job, analyze and preprocess data, how to do feature engineering and hyperparameter optimization on candidate models, and how to visualize and compare the resulting model metrics.
Topics

- Start an AutoML job with Amazon SageMaker Autopilot (p. 148)
- Review data exploration and feature engineering automated in Autopilot. (p. 148)
- Tune models to optimize performance (p. 148)
- Choose and deploy the best model (p. 148)
- Amazon SageMaker Autopilot walkthrough (p. 148)

Start an AutoML job with Amazon SageMaker Autopilot

This video shows you how to start an AutoML job with Autopilot. (Length: 8:41)

Amazon SageMaker Studio - AutoML with Amazon SageMaker Autopilot (part 1)

Review data exploration and feature engineering automated in Autopilot.

This video shows you how to examine the data exploration and candidate definition notebooks generated by Amazon SageMaker Autopilot. (Length: 10:04)

Amazon SageMaker Studio - AutoML with Amazon SageMaker Autopilot (part 2)

Tune models to optimize performance

This video shows you how to optimize model performance during training using hyperparameter tuning. (Length: 4:59)

SageMaker Studio - AutoML with Amazon SageMaker Autopilot (part 3)

Choose and deploy the best model

This video shows you how to use job metrics to choose the best model and then how to deploy it. (Length: 5:20)

SageMaker Studio - AutoML with Amazon SageMaker Autopilot (part 4)

Amazon SageMaker Autopilot walkthrough

This video walks you through an end to end demo where we first build a binary classification model automatically with Amazon SageMaker Autopilot. We see how candidate models have been built and optimized using auto-generated notebooks. We also look at the top candidates with SageMaker Experiments. Finally, we deploy the top candidate (based on XGBoost), and configure data capture with SageMaker Model Monitor.

End to end demo with AutoML on SageMaker

Tutorials: Get started with Amazon SageMaker Autopilot

Autopilot get started tutorials demonstrate how to create a machine learning model automatically without having to write code. They show you how Autopilot simplifies the machine learning experience
by helping you explore your data and try different algorithms. Autopilot builds the best machine learning model for the problem type using AutoML capabilities while allowing full control and visibility.

- **Create a machine learning model automatically with Autopilot**: You assume the role of a developer working at a bank in this tutorial. You have been asked to develop a machine learning model to predict whether or not a customer will enroll for a certificate of deposit (CD). This is a binary classification problem. The model is trained on the marketing dataset that contains information on customer demographics, responses to marketing events, and external factors.

### Create an Amazon SageMaker Autopilot experiment

To create an Amazon SageMaker Autopilot experiment, you need to name it, provide locations for the input and output data, specify the target data to predict, and optionally the type of machine learning problem to solve. When you create an Amazon SageMaker Autopilot job as a pilot experiment, SageMaker analyzes your data and creates a notebook with candidate model definitions. If you choose to run the complete experiment option, SageMaker also trains and tunes these models on your behalf. You can view statistics while the experiment is running. Afterwards, you can compare trials and delve into the details.

1. Open Amazon SageMaker Studio and sign in.
2. Choose the **Create Autopilot experiment** option from the **Build model automatically** box.

![Create Autopilot experiment](image)

3. Enter information about the experiment in the **Job Settings** form:

   - **Experiment Name** – Must be unique to your account in the current AWS Region and contain a maximum of 63 alphanumeric characters. Can include hyphens (-), but not spaces.

   ![Job Settings](image)

   - **Input data location (S3 bucket)** – The S3 bucket that contains your input data.
Create an Autopilot experiment

Note
Must be an s3:// formatted URL where Amazon SageMaker has write permissions. The S3 bucket must be in the current AWS Region and must be in CSV format and contain at least 500 rows.

- **S3 bucket name** – The bucket name must be unique across all existing bucket names in S3.
- **S3 object key prefix** – The filename of the object in the bucket, including the path to the object within the bucket.
- **S3 bucket location** – The concatenation of the S3 bucket name and S3 object key prefix.
- **Is your S3 input a manifest file?** – A manifest file includes metadata with your input data. The metadata specifies the location of your data in Amazon S3 storage, how the data are formatted, and which attributes from the dataset to use when training your model. You can use a manifest file as an alternative to preprocessing when you have labeled data being streamed in Pipe mode.
- **Target attribute name** – The name of the data column you want the model to target for predictions.
- **Output data location (S3 bucket)** – The S3 bucket where you want to store the output data.

Note
Must be an s3:// formatted URL where Amazon SageMaker has write permissions. The S3 bucket must be in the current AWS Region.
• **S3 bucket name** – The bucket name must be unique across all existing bucket names in S3.

• **S3 object key prefix** – The filename of the object in the bucket, including the path to the object within the bucket.

• **S3 bucket location** – The concatenation of the S3 bucket name and S3 object key prefix.

• Select the machine learning problem type:

![Image of machine learning problem type selection]

  - **Auto** – SageMaker infers the problem type from the values of the attribute you want to predict. In some cases, SageMaker is unable to infer accurately, in which case you must provide the value for the job to succeed.

  - **Binary classification** – Binary classification is a type of supervised learning that assigns an individual to one of two predefined and mutually exclusive classes based on their attributes. For example, a medical diagnosis for whether an individual has a disease or not based on the results of diagnostic tests.

  - **Regression** – Regression estimates the values of a dependent target variable based on one or more other variables or attributes that are correlated with it. For example, house prices based on features such as square footage and number of bathrooms.

  - **Multiclass classification** – Multiclass classification is a type of supervised learning that assigns an individual to one of several classes based on their attributes. For example, the prediction of the topic most relevant to a text document, such as politics, finance, or philosophy.

• Do you want to run a complete experiment?

  - **Yes**

  - **No**, run a pilot to create a notebook with candidate definitions

If you choose **Yes**, SageMaker generates a model as well as statistics that you can view in real time while the experiment is running. After the experiment is complete, you can view the trials, sort by objective metric, and right-click to deploy the model for use in other environments.

If you choose **No**, instead of executing the entire workflow, SageMaker stops execution after generating a notebook with candidate definitions. A candidate is a combination of data preprocessors, algorithms, and algorithm parameter settings. You can use the notebook as a starting point to guide your own process of model training/tuning. The notebook has highlighted sections that explain what kinds of changes are typical, such as changing instance type, cluster size, and so on.

4. Choose **Create Experiment**.

## Amazon SageMaker Autopilot problem types and algorithm support

When setting a problem type, such as binary classification or regression, with the AutoML API, you have the option of specifying it or of letting Amazon SageMaker Autopilot detect it on your behalf. You set the type of problem with the `CreateAutoPilot.ProblemType` parameter. This limits the kind of preprocessing and algorithms that Autopilot tries. When the job is finished, if you had set the `CreateAutoPilot.ProblemType`, then the `ResolvedAttribute.ProblemType` will match the
ProblemType you set. If you leave it blank (or null), the ProblemType will be whatever Autopilot decides on your behalf.

**Note**  
In some cases, Autopilot is unable to infer the ProblemType with high enough confidence, in which case you must provide the value for the job to succeed.

## Autopilot algorithm support

Autopilot supports three types of machine learning algorithms to address machine learning problems:

- **Linear Learner Algorithm (p. 732)**: a supervised learning algorithms used for solving either classification or regression problems.
- **XGBoost Algorithm (p. 813)**: a supervised learning algorithm that attempts to accurately predict a target variable by combining an ensemble of estimates from a set of simpler and weaker models.
- **Deep Learning Algorithm**: multilayer perceptron (MLP), a feedforward artificial neural network that can handle data that is not linear separable.

**Note**  
You do not need to specify an algorithm to use for your machine learning problem. Autopilot automatically selects the appropriate algorithm to train.

## Autopilot problem types

Your problem type options are as follows:

**Topics**

- **Regression (p. 152)**
- **Binary classification (p. 152)**
- **Multiclass classification (p. 152)**

### Regression

Regression estimates the values of a dependent target variable based on one or more other variables or attributes that are correlated with it. An example is the prediction of house prices using features like the number of bathrooms and bedrooms, square footage of the house and garden. Regression analysis can create a model that takes one or more of these features as an input and predicts the price of a house.

### Binary classification

Binary classification is a type of supervised learning that assigns an individual to one of two predefined and mutually exclusive classes based on their attributes. It is supervised because the models are trained using examples where the attributes are provided with correctly labelled objects. A medical diagnosis for whether an individual has a disease or not based on the results of diagnostic tests is an example of binary classification.

### Multiclass classification

Multiclass classification is a type of supervised learning that assigns an individual to one of several classes based on their attributes. It is supervised because the models are trained using examples where the attributes are provided with correctly labelled objects. An example is the prediction of the topic most relevant to a text document. A document may be classified as being about, say, religion or politics or finance, or about one of several other predefined topic classes.
Amazon SageMaker Autopilot notebooks generated to manage AutoML tasks

Amazon SageMaker Autopilot manages the key tasks in an automatic machine learning (AutoML) process. They are implemented by Autopilot with an AutoML job. The AutoML job creates two notebooks that describe the plan that Autopilot follows to generate candidate models. A candidate model consists of a (pipeline, algorithm) pair. First, there's a data exploration notebook, that describes what Autopilot learned about the data that you provided. Second, there's a candidate generation notebook, which uses the information about the data to generate candidates.

You can run both notebooks in Amazon SageMaker or locally if you have installed the Amazon SageMaker Python SDK. You can share the notebooks just like any other SageMaker Studio notebook. The notebooks are created for you to conduct experiment. For example, you could edit the following items in the notebooks:

- the preprocessors used on the data
- the number of hyperparameter optimization (HPO) runs and their parallelism
- the algorithms to try
- the instance types used for the HPO jobs
- the hyperparameter ranges

Modifications to the candidate generation notebook are encouraged to be used as a learning tool. This capability allows you to learn about how the decisions made during the machine learning process impact the your results.

Note
When you run the notebooks in your default instance you incur baseline costs, but when you execute HPO jobs from the candidate notebook, these jobs use additional compute resources that incur additional costs.

Data exploration notebook

A notebook is produced during the analysis phase of the AutoML job that helps you identify problems in your dataset. It identifies specific areas for investigation in order to help you identify upstream problems with your data that may result in a suboptimal model.

Candidate generation notebook

The candidate generation notebook contains each suggested preprocessing step, algorithm, and hyperparameter ranges. If you chose just to produce the notebook and not to run the AutoML job, you can then decide which candidates are to be trained and tuned. They optimize automatically and a final, best candidate will be identified. If you ran the job directly without seeing the candidates first, then only best candidate is displayed when you open the notebook after the completion of the job.

Configure inference output in Autopilot-generated containers

Amazon SageMaker Autopilot generates an ordered `ContainerDefinition` list that can be used to build a model to deploy in a machine learning pipeline. This model can be used for online hosting and inference. Customers can access the list of inference container definitions with the
Inference container definitions for regression and classification problem types

The inference containers generated depend on the problem type of the job.

- **Regression**: generates two containers:
  1. First is the feature engineering container that transforms the original features to features that the regression algorithms can train on.
  2. Second is the algorithm container that transforms features and generates the regression score for the dataset.

- **Classification**: generates three containers:
  1. First is the feature engineering container that transforms the original features to features that the classification algorithms can train on.
  2. Second is the algorithm container that generates the winning `predicted_label` and that can also produce the various probabilities associated with the classification outcomes in the inference response.
  3. Third is a feature engineering container that performs post-processing of the algorithm prediction, for example, an inverse transform of the predicted label to original label.

Select inference response for classification models

Classification inference containers allow you to select the content of the inference responses. There are four predefined keys:

- `predicted_label`: The winning label determined by Autopilot.
- `probability`: The probability of the `True` class for binary classification. The probability of winning class for multiclass classification.
- `probabilities`: The list of probabilities for all corresponding labels.
- `labels`: List of all labels

By default, inference containers are configured to generate `predicted_label` only.

Three environment variables are used to select the optional inference content:

- `SAGEMAKER_INFERENCE_SUPPORTED`: this is set to provide hints to the users about what content each container supports.
- `SAGEMAKER_INFERENCE_INPUT`: should be set to the keys that the container expects in its input payload.
- `SAGEMAKER_INFERENCE_OUTPUT`: should be populated with the set of keys the container outputs.

In order to choose the inference response content, we need to add the `SAGEMAKER_INFERENCE_INPUT`, `SAGEMAKER_INFERENCE_OUTPUT` appropriately in the second and the third containers in the list of containers for classification problem.
The keys supported by the third classification model container are `predicted_label`, `labels`, `probability` and `probabilities`. Hence the `SAGEMAKER_INFERENCE_SUPPORTED` environment includes the names of all these keys.

The keys supported by the second container (Algorithm) are `predicted_label`, `probability` and `probabilities`. Note that the `labels` is deliberately not added to the `SAGEMAKER_INFERENCE_SUPPORTED`.

Here is how to update the definition of the inference containers to receive `predicted_label` and `probability`.

```python
containers[1]['Environment'].update({'SAGEMAKER_INFERENCE_OUTPUT': 'predicted_label, probability'})
containers[2]['Environment'].update({'SAGEMAKER_INFERENCE_INPUT': 'predicted_label, probability'})
containers[2]['Environment'].update({'SAGEMAKER_INFERENCE_OUTPUT': 'predicted_label, probability'})
```

Here is how to update the definition of the inference containers to receive `predicted_label` and `probabilities` and `labels`. Note that you do not need to pass the `labels` to the second container, the algorithm container, as it's redundant and can be generated by the third container independently. This reduces the latency.

```python
containers[1]['Environment'].update({'SAGEMAKER_INFERENCE_OUTPUT': 'predicted_label, probabilities'})
containers[2]['Environment'].update({'SAGEMAKER_INFERENCE_INPUT': 'predicted_label, probabilities'})
containers[2]['Environment'].update({'SAGEMAKER_INFERENCE_OUTPUT': 'predicted_label, probabilities, labels'})
```

You can use the Amazon SageMaker Python SDK to accomplish this as follows:

```python
from sagemaker import AutoML

aml = AutoML.attach(auto_ml_job_name='AUTOML_JOB_NAME')
aml_best_model = aml.create_model(name='SELECT_MODEL_NAME',
candidate=None,
inference_response_keys=['probabilities', 'labels'])

aml_transformer = aml_best_model.transformer(accept='text/csv',
assemble_with='Line',
instance_type='ml.m5.xlarge',
instance_count=1,)

aml_transformer.transform(test_data_s3_path,
content_type='text/csv',
split_type='Line',
job_name=<Add jobName>,
wait=True)
```

---

Amazon SageMaker Autopilot quotas

There are quotas that limit the resources available to you when using Amazon SageMaker Autopilot. Some of these limits are increasable and some are not.

Topics
• Quotas that you can increase (p. 156)
• Resource quotas (p. 156)

Quotas that you can increase

There are default limits for the size of datasets for the number of concurrent jobs you can run with Amazon SageMaker Autopilot for each AWS account, per AWS Region. You can increase these limits by contacting AWS Support.

Resource limits

<table>
<thead>
<tr>
<th>Resource</th>
<th>Region</th>
<th>Default limits</th>
<th>Can be increased up to</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size of input dataset</td>
<td>All</td>
<td>5 GB</td>
<td>Hundreds of GBs</td>
</tr>
<tr>
<td>Number of concurrent</td>
<td>us-east-1, us-east-2, us-west-2, ap-northeast-1, eu-west-1, eu-central-1</td>
<td>4</td>
<td>Hundreds</td>
</tr>
<tr>
<td>Autopilot jobs</td>
<td>ap-northeast-2, ap-southeast-2, eu-west-2, ap-southeast-1</td>
<td>2</td>
<td>Hundreds</td>
</tr>
<tr>
<td></td>
<td>All other regions</td>
<td>1</td>
<td>Tens</td>
</tr>
</tbody>
</table>

To request a quota increase

1. Open the AWS Support Center page, sign in if necessary, and then choose Create case.
2. On the Create case page, choose Account and billing support.
3. In the Case details panel make the following entries:
   - For Type choose Account.
   - For Subject enter Amazon SageMaker Autopilot service limit increase request.
   - Provide a detailed Description of your request, including:
     - The Autopilot quota that you want to increase, and the value you want it increased to, if known.
     - The reason why you are seeking the quota increase.
     - The region for which you are requesting the increase.
4. Provide your preferred Contact options and choose Submit.

Resource quotas

The following table contains the runtime resource limits for an Amazon SageMaker Autopilot job in an AWS region.

Resource limits per Autopilot job

<table>
<thead>
<tr>
<th>Resource</th>
<th>Limit per Autopilot job</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum runtime for an Autopilot job</td>
<td>30 days</td>
</tr>
</tbody>
</table>
API reference guide for Amazon SageMaker Autopilot

Amazon SageMaker provides API reference documentation that describes all of the REST operations and data types used by Autopilot and a higher level Amazon SageMaker Python SDK that you can use to create and manage AutoML jobs. It also provides a command line interface (CLI), an AWS SDK for Python (Boto) for low-level clients of SageMaker services, and SDKs for .NET, C++, Go, Java, JavaScript, PHP V3, and Ruby V3. The following sections describe these Autopilot programming interfaces.

Topics
- SageMaker API reference (p. 157)
- Amazon SageMaker Python SDK (p. 158)
- AWS Command Line Interface (CLI) (p. 158)
- AWS SDK for Python (Boto) (p. 158)
- AWS SDK for .NET (p. 158)
- AWS SDK for C++ (p. 159)
- AWS SDK for Go (p. 159)
- AWS SDK for Java (p. 159)
- AWS SDK for JavaScript (p. 159)
- AWS SDK for PHP V3 (p. 160)
- AWS SDK for Ruby V3 (p. 160)

SageMaker API reference

This API provides HTTP service APIs for creating and managing Amazon SageMaker Autopilot resources.

Actions
- CreateAutoMLJob
- DescribeAutoMLJob
- ListAutoMLJobs
- ListCandidatesForAutoMLJob
- StopAutoMLJob

Data Types
- AutoMLCandidate
- AutoMLCandidateStep
- AutoMLChannel
- AutoMLContainerDefinition
- AutoMLDataSource
- AutoMLJobArtifacts
- AutoMLJobCompletionCriteria
- AutoMLJobConfig
- AutoMLJobObjective
- AutoMLJobSummary
- AutoMLOutputDataConfig
Amazon SageMaker Python SDK

This Python library provides several high-level abstractions for working with SageMaker. The following classes can be used to manage AutoML jobs.

- AutoML
- AutoMLInput
- AutoMLJob
- CandidateEstimator
- CandidateStep

For more information how this Python SDK simplifies model training and deployment, see Using the Amazon SageMaker Python SDK.

AWS Command Line Interface (CLI)

The AWS CLI provides APIs for creating and managing SageMaker resources. Here are the AWS CLI for Amazon SageMaker Autopilot commands.

- create-auto-ml-job
- describe-auto-ml-job
- list-auto-ml-jobs
- list-candidates-for-auto-ml-job
- stop-auto-ml-job

AWS SDK for Python (Boto)

Boto is the Amazon Web Services (AWS) SDK for Python. It enables Python developers to create, configure, and manage AWS services such as SageMaker. Boto provides a low-level Client API that maps to the underlying SageMaker service API. Here is a list of the methods used to manage AutoML jobs with the Client class.

- create_auto_ml_job()
- describe_auto_ml_job()
- list_auto_ml_jobs()
- list_candidates_for_auto_ml_job()
- stop_auto_ml_job()

AWS SDK for .NET

The .NET SDK enables developers to create, configure, and manage AWS services such as SageMaker. The API maps to the underlying SageMaker service API. Here is a list of the methods used to manage AutoML jobs with the Client class.
• CreateAutoMLJob
• DescribeAutoMLJob
• ListAutoMLJobs
• ListCandidatesForAutoMLJob
• StopAutoMLJob

AWS SDK for C++

The C++ SDK enables developers to create, configure, and manage AWS services such as SageMaker. The API maps to the underlying SageMaker service API. For information on the methods used to manage AutoML jobs with the Client class, see `Aws::SageMaker::SageMakerClient Class Reference`.

AWS SDK for Go

The Go SDK enables developers to create, configure, and manage AWS services such as SageMaker. The API maps to the underlying SageMaker service API. Here is a list of the methods used to manage AutoML jobs with the Client class.

• CreateAutoMLJob
• DescribeAutoMLJob
• ListAutoMLJobs
• ListCandidatesForAutoMLJob
• StopAutoMLJob

AWS SDK for Java

The Java SDK enables developers to create, configure, and manage AWS services such as SageMaker. The API maps to the underlying SageMaker service API. Here is a list of the methods used to manage AutoML jobs with the Client class.

• createAutoMLJob
• describeAutoMLJob
• listAutoMLJobs
• listCandidatesForAutoMLJob
• stopAutoMLJob

AWS SDK for JavaScript

The JavaScript SDK enables developers to create, configure, and manage AWS services such as SageMaker. The API maps to the underlying SageMaker service API. Here is a list of the methods used to manage AutoML jobs with the Client class.

• createAutoMLJob
• describeAutoMLJob
• listAutoMLJobs
• listCandidatesForAutoMLJob
• stopAutoMLJob
AWS SDK for PHP V3

The PHP V3 SDK enables developers to create, configure, and manage AWS services such as SageMaker. The API maps to the underlying SageMaker service API. Here is a list of the methods used to manage AutoML jobs with the Client class.

- `CreateAutoMLJob`
- `DescribeAutoMLJob`
- `ListAutoMLJobs`
- `ListCandidatesForAutoMLJob`
- `StopAutoMLJob`

AWS SDK for Ruby V3

The Ruby V3 SDK enables developers to create, configure, and manage AWS services such as SageMaker. The API maps to the underlying SageMaker service API. Here is a list of the methods used to manage AutoML jobs with the Client class.

- `create_auto_ml_job()`
- `describe_auto_ml_job()`
- `list_auto_ml_jobs()`
- `list_candidates_for_auto_ml_job()`
- `stop_auto_ml_job()`
Label Data

To train a machine learning model, you need a large, high-quality, labeled dataset. You can label your data using Amazon SageMaker Ground Truth. Choose from one of the Ground Truth built-in task types or create your own custom labeling workflow. To improve the accuracy of your data labels and reduce the total cost of labeling your data, use Ground Truth enhanced data labeling features like automated data labeling and annotation consolidation.

Topics
- Use Amazon SageMaker Ground Truth to Label Data (p. 161)
- Create and Manage Workforces (p. 427)
- Crowd HTML Elements Reference (p. 448)

Use Amazon SageMaker Ground Truth to Label Data

To train a machine learning model, you need a large, high-quality, labeled dataset. Ground Truth helps you build high-quality training datasets for your machine learning models. With Ground Truth, you can use workers from either Amazon Mechanical Turk, a vendor company that you choose, or an internal, private workforce along with machine learning to enable you to create a labeled dataset. You can use the labeled dataset output from Ground Truth to train your own models. You can also use the output as a training dataset for an Amazon SageMaker model.

Depending on your ML application, you can choose from one of the Ground Truth built-in task types to have workers generate specific types of labels for your data. You can also build a custom labeling workflow to provide your own UI and tools to workers labeling your data. To learn more about the Ground Truth built-in task types, see Built-in Task Types (p. 312). To learn how to create a custom labeling workflow, see Creating Custom Labeling Workflows (p. 291).

In order to automate labeling your training dataset, you can optionally use automated data labeling, a Ground Truth process that uses machine learning to decide which data needs to be labeled by humans. Automated data labeling may reduce the labeling time and manual effort required. For more information, see Automate Data Labeling (p. 406). To create a custom labeling job, see Creating Custom Labeling Workflows (p. 291).

Use either pre-built or custom tools to assign the labeling tasks for your training dataset. A labeling UI template is a webpage that Ground Truth uses to present tasks and instructions to your workers. The SageMaker console provides built-in templates for labeling data. You can use these templates to get started, or you can build your own tasks and instructions by using our HTML 2.0 components. For more information, see Creating Custom Labeling Workflows (p. 291).

Use the workforce of your choice to label your dataset. You can choose your workforce from:
- The Amazon Mechanical Turk workforce of over 500,000 independent contractors worldwide.
- A private workforce that you create from your employees or contractors for handling data within your organization.
- A vendor company that you can find in the AWS Marketplace that specializes in data labeling services.
For more information, see Create and Manage Workforces (p. 427).

You store your datasets in Amazon S3 buckets. The buckets contain three things: The data to be labeled, an input manifest file that Ground Truth uses to read the data files, and an output manifest file. The output file contains the results of the labeling job. For more information, see Use Input and Output Data (p. 340).

Events from your labeling jobs appear in Amazon CloudWatch under the /aws/sagemaker/LabelingJobs group. CloudWatch uses the labeling job name as the name for the log stream.

Are You a First-time User of Ground Truth?

If you are a first-time user of Ground Truth, we recommend that you do the following:

1. **Read Getting started (p. 162)**—This section walks you through setting up your first Ground Truth labeling job.

2. **Explore other topics**—Depending on your needs, do the following:
   - **Explore built-in task types**— Use built-in task types to streamline the process of creating a labeling job. See Built-in Task Types (p. 312) to learn more about Ground Truth built-in task types.
   - **Manage your labeling workforce**—Create new work teams and manage your existing workforce. For more information, see Create and Manage Workforces (p. 427).
   - **Learn about streaming labeling jobs**— Create a streaming labeling job and send new dataset objects to workers in real time using a perpetually running labeling job. Workers continuously receive new data objects to label as long as the labeling job is active and new objects are being sent to it. To learn more, see Ground Truth Streaming Labeling Jobs (p. 343).

3. **See the Reference**—This section describes operations to automate Ground Truth operations.

**Getting started**

This video shows you how to setup and use Amazon SageMaker Ground Truth. (Length: 9:37)

To get started using Amazon SageMaker Ground Truth, follow the instructions in the following sections. The sections here explain how to use the console to create a labeling job, assign a public or private workforce, and send the labeling job to your workforce. You can also learn how to monitor the progress of a labeling job.

If you want to create a custom labeling job, see Creating Custom Labeling Workflows (p. 291) for instructions.

Before you create a labeling job, you must upload your dataset to an Amazon S3 bucket. For more information, see Use Input and Output Data (p. 340).

**Topics**

- **Step 1: Before You Begin (p. 162)**
- **Step 2: Create a Labeling Job (p. 163)**
- **Step 3: Select Workers (p. 164)**
- **Step 4: Configure the Bounding Box Tool (p. 165)**
- **Step 5: Monitoring Your Labeling Job (p. 165)**

**Step 1: Before You Begin**

Before you begin using the SageMaker console to create a labeling job, you must set up the dataset for use. Do this:
1. Save two images at publicly available HTTP URLs. The images are used when creating instructions for completing a labeling task. The images should have an aspect ratio of around 2:1. For this exercise, the content of the images is not important.

2. Create an Amazon S3 bucket to hold the input and output files. The bucket must be in the same Region where you are running Ground Truth. Make a note of the bucket name because you use it during step 2.

Ground Truth requires all S3 buckets that contain labeling job input image data have a CORS policy attached. To learn more about this change, see CORS Permission Requirement (p. 415).

3. Assign the following permissions policy to the user that is creating the labeling job:

```json
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Sid": "sagemakergroundtruth",
         "Effect": "Allow",
         "Action": [
            "cognito-idp:CreateGroup",
            "cognito-idp:CreateUserPool",
            "cognito-idp:CreateUserPoolDomain",
            "cognito-idp:AdminCreateUser",
            "cognito-idp:CreateUserPoolClient",
            "cognito-idp:AdminAddUserToGroup",
            "cognito-idp:DescribeUserPoolClient",
            "cognito-idp:DescribeUserPool",
            "cognito-idp:UpdateUserPool"
         ],
         "Resource": "*"
      }
   ]
}
```

Next

Step 2: Create a Labeling Job (p. 163)

Step 2: Create a Labeling Job

In this step you use the console to create a labeling job. You tell Amazon SageMaker Ground Truth the Amazon S3 bucket where the manifest file is stored and configure the parameters for the job. For more information about storing data in an Amazon S3 bucket, see Use Input and Output Data (p. 340).

To create a labeling job

2. From the left navigation, choose Labeling jobs.
3. Choose Create labeling job to start the job creation process.
4. In the Job overview section, provide the following information:
   - **Job name** – Give the labeling job a name that describes the job. This name is shown in your job list. The name must be unique in your account in an AWS Region.
   - **Label attribute name** – Leave this unchecked as the default value is the best option for this introductory job.
   - **Input data setup** – Select Automated data setup. This option allows you to automatically connect to your input data in S3.
• **S3 location for input datasets** – Enter the S3 location where you added the images in step 1.
• **S3 location for output datasets** – The location where your output data is written in S3.
• **Data type** – Use the drop down menu to select **Image**. Ground Truth will use all images found in the S3 location for input datasets as input for your labeling job.
• **IAM role** – Create or choose an IAM role with the SageMakerFullAccess IAM policy attached.

5. In the **Task type** section, for the **Task category** field, choose **Image**.
6. In the **Task selection** choose **Bounding box**.
7. Choose **Next** to move on to configuring your labeling job.

**Next**

**Step 3: Select Workers (p. 164)**

**Step 3: Select Workers**

In this step you choose a workforce for labeling your dataset. You can create your own private workforce or you can use the Amazon Mechanical Turk workforce. If you create a private workforce in this step you won't be able to import your Amazon Cognito user pool later. For more information, see Manage a Private Workforce (Amazon Cognito) (p. 433). Use the Amazon Mechanical Turk workforce for this exercise instead.

You can create a private workforce to test Amazon SageMaker Ground Truth. Use email addresses to invite the members of your workforce.

**To create a private workforce**

1. In the **Workers** section, choose **Private**.
2. If this is your first time using a private workforce, in the **Email addresses** field, enter up to 100 email addresses. The addresses must be separated by a comma. You should include your own email address so that you are part of the workforce and can see data object labeling tasks.
3. In the **Organization name** field, enter the name of your organization. This information is used to customize the email sent to invite a person to your private workforce.
4. In the **Contact email** field enter an email address that members of the workforce use to report problems with the task.

If you choose to use the Amazon Mechanical Turk workforce to label the dataset, you are charged for labeling tasks completed on the dataset.

**To use the Amazon Mechanical Turk workforce**

1. In the **Workers** section, choose **Public**.
2. Choose **The dataset does not contain PII** to acknowledge that the dataset does not contain any personally identifiable information.
3. Choose **The dataset does not contain adult content**. to acknowledge that the sample dataset has no adult content.
4. Review and accept the statement that the dataset will be viewed by the public workforce.

**Next**

**Step 4: Configure the Bounding Box Tool (p. 165)**
Step 4: Configure the Bounding Box Tool

Finally you configure the bounding box tool to give instructions to your workers. You can configure a task title that describes the task and provides high-level instructions for the workers. You can provide both quick instructions and full instructions. Quick instructions are displayed next to the image to be labeled. Full instructions contain detailed instructions for completing the task. In this example, you only provide quick instructions. You can see an example of full instructions by choosing Full instructions at the bottom of the section.

To configure the bounding box tool

1. In the Task description field type in brief instructions for the task. For example:
   
   **Draw a box around any objects in the image.**
   
   Replace objects with the name of an object that appears in your images.

2. In the Labels field, type a category name for the objects that the worker should draw a bounding box around. For example, if you are asking the worker to draw boxes around football players, you could use "FootballPlayer" in this field.

3. The Short instructions section enables you to create instructions that are displayed on the page with the image that your workers are labeling. We suggest that you include an example of a correctly drawn bounding box and an example of an incorrectly drawn box. To create your own instructions, use these steps:
   
   a. Select the text between GOOD EXAMPLE and the image placeholder. Replace it with the following text:
   
      **Draw the box around the object with a small border.**
   
   b. Select the first image placeholder and delete it.
   
   c. Choose the image button and then enter the HTTPS URL of one of the images that you created in step 1.
   
   d. Select the text between BAD EXAMPLE and the image placeholder. Replace it with the following text:
   
      **Don't make the bounding box too large or cut into the object.**
   
   e. Select the second image placeholder and delete it.
   
   f. Choose the image button and then enter the HTTPS URL of the other image that you created in step 1.

Configuration of your labeling job is complete. To start your job, choose Submit.

Next

Step 5: Monitoring Your Labeling Job (p. 165)

Step 5: Monitoring Your Labeling Job

After you create your labeling job, you see a list of all the jobs that you have created. You can use this list to monitor that status of your labeling jobs. The list has the following fields:

- **Name** – The name that you assigned the job when you created it.
- **Status** – The completion status of the job. The status can be one of Complete, Failed, In progress, or Stopped.
- **Labeled objects/total** – Shows the total number of objects in the labeling job and how many of them have been labeled.
• **Creation time** – The date and time that you created the job.

You can also clone, chain, or stop a job. Select a job and then select one of the following from the **Actions** menu:

• **Clone** – Creates a new labeling job with the configuration copied from the selected job. You can clone a job when you want to change to the job and run it again. For example, you can clone a job that was sent to a private workforce so that you can send it to the Amazon Mechanical Turk workforce. Or you can clone a job to rerun it against a new dataset stored in the same location as the original job.

• **Chain** – Creates a new labeling job that can build upon the data and models (if any) of a stopped, failed, or completed job. For more information about the use cases and how to use it, see **Chaining Labeling Jobs** *(p. 412).*

• **Stop** – Stops a running job. You cannot restart a stopped job. You can clone a job to start over or chain the job to continue from where it left off. Labels for any already labeled objects are written to the output file location. For more information, see **Output Data** *(p. 381).*

---

**Label Images**

Use Ground Truth to label images. Select one of the following built-in task types to learn more about that task type. Each page includes instructions to help you create a labeling job using that task type.

**Topics**

• Bounding Box *(p. 166)*
• Image Semantic Segmentation *(p. 171)*
• Auto-Segmentation Tool *(p. 174)*
• Image Classification (Single Label) *(p. 178)*
• Image Classification (Multi-label) *(p. 180)*
• Image Label Verification *(p. 184)*

---

**Bounding Box**

The images used to train a machine learning model often contain more than one object. To classify and localize one or more objects within images, use the Amazon SageMaker Ground Truth bounding box labeling job task type. In this context, localization means the pixel-location of the bounding box.

You create a bounding box labeling job using the **Ground Truth** section of the Amazon SageMaker console or the `CreateLabelingJob` operation.

**Important**

For this task type, if you create your own manifest file, use "source-ref" to identify the location of each image file in Amazon S3 that you want labeled. For more information, see **Input Data** *(p. 341).*

**Creating a Bounding Box Labeling Job (Console)**

You can follow the instructions **Create a Labeling Job (Console)** *(p. 314)* to learn how to create a bounding box labeling job in the **Console**. In Step 10, choose **Image** from the **Task category** drop-down menu, and choose **Bounding box** as the task type.

Ground Truth provides a worker UI similar to the following for labeling tasks. When you create the labeling job with the console, you specify instructions to help workers complete the job and up to 50 labels that workers can choose from.
Instructions

Good example

Fit each box tightly around the boundaries of the object.

Bad example

Boxes should not overlap with the boundaries of objects.
Create a Bounding Box Labeling Job (API)

To create a bounding box labeling job, use the SageMaker API operation `CreateLabelingJob`. This API defines this operation for all AWS SDKs. To see a list of language-specific SDKs supported for this operation, review the See Also section of `CreateLabelingJob`.

Follow the instructions on Create a Labeling Job (API) (p. 317) and do the following while you configure your request:

- Pre-annotation Lambda functions for this task type end with `PRE-BoundingBox`. To find the pre-annotation Lambda ARN for your Region, see `PreHumanTaskLambdaArn`.
- Annotation-consolidation Lambda functions for this task type end with `ACS-BoundingBox`. To find the annotation-consolidation Lambda ARN for your Region, see `AnnotationConsolidationLambdaArn`.

The following is an example of an AWS Python SDK (Boto3) request to create a labeling job in the US East (N. Virginia) Region. All parameters in red should be replaced with your specifications and resources.

```python
response = client.create_labeling_job(
    LabelingJobName='example-bounding-box-labeling-job',
    LabelAttributeName='label',
    InputConfig={
        'DataSource': {
            'S3DataSource': {
                'ManifestS3Uri': 's3://bucket/path/manifest-with-input-data.json'
            }
        },
        'DataAttributes': {
            'ContentClassifiers': [
                'FreeOfPersonallyIdentifiableInformation', 'FreeOfAdultContent'
            ]
        }
    },
    OutputConfig={
        'S3OutputPath': 's3://bucket/path/file-to-store-output-data',
        'KmsKeyId': 'string'
    },
    RoleArn='arn:aws:iam::*:role/*',
    LabelCategoryConfigS3Uri='s3://bucket/path/label-categories.json',
    StoppingConditions={
        'MaxHumanLabeledObjectCount': 123,
        'MaxPercentageOfInputDatasetLabeled': 123
    },
    HumanTaskConfig={
        'WorkteamArn': 'arn:aws:sagemaker:region:*:workteam/private-crowd/*',
        'UiConfig': {
            'UiTemplateS3Uri': 's3://bucket/path/worker-task-template.html'
        },
        'PreHumanTaskLambdaArn': 'arn:aws:lambda:us-east-1:432418664414:function:PRE-BoundingBox',
        'TaskKeywords': [
            'Bounding Box'
        ],
        'TaskTitle': 'Bounding Box task',
        'TaskDescription': 'Draw bounding boxes around objects in an image',
        'NumberOfHumanWorkersPerDataObject': 123,
        'TaskTimeLimitInSeconds': 123,
        'TaskAvailabilityLifetimeInSeconds': 123,
        'MaxConcurrentTaskCount': 123,
        'AnnotationConsolidationConfig': {
            'AnnotationConsolidationLambdaArn': 'arn:aws:lambda:us-east-1:432418664414:function:ACS-BoundingBox'
        }
    }
)`
Providing a Template for Bounding Box Labeling Jobs

If you create a labeling job using the API, you must supply a worker task template in UiTemplateS3Uri. Copy and modify the following template. Only modify the short-instructions, full-instructions, and header. Upload this template to S3, and provide the S3 URI for this file in UiTemplateS3Uri.

```html
<script src="https://assets.crowd.aws/crowd-html-elements.js"></script>

<crowd-form>
  <crowd-bounding-box
    name="boundingBox"
    src="{{ task.input.taskObject | grant_read_access }}"
    header="please draw box"
    labels="{{ task.input.labels | to_json | escape }}"
  >
    <full-instructions header="Bounding box instructions">
      <ol>
        <li><strong>Inspect</strong> the image</li>
        <li><strong>Determine</strong> if the specified label(s) are visible in the picture.</li>
        <li><strong>Outline</strong> each instance of the specified label(s) in the image using the provided "Box" tool.</li>
      </ol>
      <ul>
        <li>Boxes should fit tight around each object</li>
        <li>Do not include parts of the object are overlapping or that cannot be seen, even though you think you can interpolate the whole shape.</li>
        <li>Avoid including shadows.</li>
        <li>If the target is off screen, draw the box up to the edge of the image.</li>
      </ul>
    </full-instructions>
    <short-instructions>
      <h3><span style="color: rgb(0, 138, 0);">Good example</span></h3>
      Enter description of a correct bounding box label and add images
      <h3><span style="color: rgb(230, 0, 0);">Bad example</span></h3>
      Enter description of an incorrect bounding box label and add images
    </short-instructions>
  </crowd-bounding-box>
</crowd-form>
```

Bounding Box Output Data

Once you have created a bounding box labeling job, your output data will be located in the Amazon S3 bucket specified in the S3OutputPath parameter when using the API or in the Output dataset location field of the Job overview section of the console.

For example, the output manifest file of a successfully completed single-class bounding box task will contain the following:

```json
[
  {
    "boundingBox": {
      "boundingBoxes": [
        {
          "height": 2832,
```
The `boundingBoxes` parameter identifies the location of the bounding box drawn around an object identified as a “bird” relative to the top-left corner of the image which is taken to be the (0,0) pixel-coordinate. In the previous example, `left` and `top` identify the location of the pixel in the top-left corner of the bounding box relative to the top-left corner of the image. The dimensions of the bounding box are identified with `height` and `width`. The `inputImageProperties` parameter gives the pixel-dimensions of the original input image.

When you use the bounding box task type, you can create single- and multi-class bounding box labeling jobs. The output manifest file of a successfully completed multi-class bounding box will contain the following:

```json
[
  {
    "boundingBox": {
      "boundingBoxes": [
      {
        "height": 938,
        "label": "squirrel",
        "left": 316,
        "top": 218,
        "width": 785
      },
      {
        "height": 825,
        "label": "rabbit",
        "left": 1930,
        "top": 2265,
        "width": 540
      },
      {
        "height": 1174,
        "label": "bird",
        "left": 748,
        "top": 2113,
        "width": 927
      },
      {
        "height": 893,
        "label": "bird",
        "left": 1333,
        "top": 847,
        "width": 736
      }
      ],
      "inputImageProperties": {
      "height": 3726,
      "width": 2662
    }
  }
}
```
To learn more about the output manifest file that results from a bounding box labeling job, see Bounding Box Job Output (p. 386).

To learn more about the output manifest file generated by Ground Truth and the file structure the Ground Truth uses to store your output data, see Output Data (p. 381).

Image Semantic Segmentation

To identify the contents of an image at the pixel level, use an Amazon SageMaker Ground Truth semantic segmentation labeling task. When assigned a semantic segmentation labeling job, workers classify pixels in the image into a set of predefined labels or classes. Ground Truth supports single and multi-class semantic segmentation labeling jobs.

Images that contain large numbers of objects that need to be segmented require more time. To help workers (from a private or vendor workforce) label these objects in less time and with greater accuracy, Ground Truth provides an AI-assisted auto-segmentation tool. For information, see Auto-Segmentation Tool (p. 174).

You create a semantic segmentation labeling job using the Ground Truth section of the Amazon SageMaker console or the `CreateLabelingJob` operation.

**Important**

For this task type, if you create your own manifest file, use "source-ref" to identify the location of each image file in Amazon S3 that you want labeled. For more information, see Input Data (p. 341).

Creating a Semantic Segmentation Labeling Job (Console)

You can follow the instructions Create a Labeling Job (Console) (p. 314) to learn how to create a semantic segmentation labeling job in the SageMaker console. In Step 10, choose Image from the Task category drop down menu, and choose Semantic segmentation as the task type.

Ground Truth provides a worker UI similar to the following for labeling tasks. When you create the labeling job with the console, you specify instructions to help workers complete the job and labels that workers can choose from.
Instructions

View full instructions

View tool guide

How to use the Auto-segment tool

Good example

All pixels in the image that are part of an animal have been colors with the appropriate label color.

Bad example

Some animals in the image have not been colored in completely.

The color for a given animal extends beyond the boundaries of the animal.
Create a Semantic Segmentation Labeling Job (API)

To create a semantic segmentation labeling job, use the SageMaker API operation `CreateLabelingJob`. This API defines this operation for all AWS SDKs. To see a list of language-specific SDKs supported for this operation, review the See Also section of `CreateLabelingJob`.

Follow the instructions on Create a Labeling Job (API) (p. 317) and do the following while you configure your request:

- Pre-annotation Lambda functions for this task type end with `PRE-SemanticSegmentation`. To find the pre-annotation Lambda ARN for your Region, see `PreHumanTaskLambdaArn`.
- Annotation-consolidation Lambda functions for this task type end with `ACS-SemanticSegmentation`. To find the annotation-consolidation Lambda ARN for your Region, see `AnnotationConsolidationLambdaArn`.

The following is an example of an AWS Python SDK (Boto3) request to create a labeling job in the US East (N. Virginia) Region. All parameters in red should be replaced with your specifications and resources.

```python
response = client.create_labeling_job(
    LabelingJobName='example-semantic-segmentation-labeling-job',
    LabelAttributeName='label',
    InputConfig={
        'DataSource': {
            'S3DataSource': {
                'ManifestS3Uri': 's3://bucket/path/manifest-with-input-data.json'
            },
        },
        'DataAttributes': {
            'ContentClassifiers': [
                'FreeOfPersonallyIdentifiableInformation', 'FreeOfAdultContent',
            ],
        }
    },
    OutputConfig={
        'S3OutputPath': 's3://bucket/path/file-to-store-output-data',
        'KmsKeyId': 'string'
    },
    RoleArn='arn:aws:iam::*:role/*',
    LabelCategoryConfigS3Uri='s3://bucket/path/label-categories.json',
    StoppingConditions={
        'MaxHumanLabeledObjectCount': 123,
        'MaxPercentageOfInputDatasetLabeled': 123
    },
    HumanTaskConfig={
        'WorkteamArn': 'arn:aws:sagemaker:region:*:workteam/private-crowd/*',
        'UiConfig': {
            'UiTemplateS3Uri': 's3://bucket/path/worker-task-template.html'
        },
        'TaskKeywords': ['Semantic Segmentation'],
        'TaskTitle': 'Semantic segmentation task',
        'TaskDescription': 'For each category provided, segment out each relevant object using the color associated with that category',
        'NumberOfHumanWorkersPerDataObject': 123,
        'TaskTimeLimitInSeconds': 123,
        'TaskAvailabilityLifetimeInSeconds': 123,
        'MaxConcurrentTaskCount': 123,
        'AnnotationConsolidationConfig': {
```
```
Provide a Template for Semantic Segmentation Labeling Jobs

If you create a labeling job using the API, you must supply a worker task template in UiTemplateS3Uri. Copy and modify the following template. Only modify the short-instructions, full-instructions, and header.

Upload this template to S3, and provide the S3 URI for this file in UiTemplateS3Uri.

Semantic Segmentation Output Data

Once you have created a semantic segmentation labeling job, your output data will be located in the Amazon S3 bucket specified in the S3OutputPath parameter when using the API or in the Output dataset location field of the Job overview section of the console.

To learn more about the output manifest file generated by Ground Truth and the file structure the Ground Truth uses to store your output data, see Output Data (p. 381).

To see an example of an output manifest file for a semantic segmentation labeling job, see 3D Point Cloud Semantic Segmentation Output (p. 397).

Auto-Segmentation Tool

Image segmentation is the process of dividing an image into multiple segments, or sets of labeled pixels. In Amazon SageMaker Ground Truth, the process of identifying all pixels that fall under a given label involves applying a colored filler, or “mask”, over those pixels. Some labeling job tasks contain
images with a large numbers of objects that need to be segmented. To help workers label these objects in less time and with greater accuracy, Ground Truth provides an auto-segmentation tool for segmentation tasks assigned to private and vendor workforces. This tool uses a machine learning model to automatically segment individual objects in the image with minimal worker input. Workers can refine the mask generated by the auto-segmentation tool using other tools found in the worker console. This helps workers complete image segmentation tasks faster and more accurately, resulting in lower cost and higher label quality.

**Note**
The auto-segmentation tool is available for segmentation tasks that are sent to a private workforce or vendor workforce. It isn't available for tasks sent to the public workforce (Amazon Mechanical Turk).

**Tool Preview**
When workers are assigned a labeling job that provides the auto-segmentation tool, they are provided with detailed instructions on how to use the tool. For example, a worker might see the following in the worker console:
Instructions

View full instructions
View tool guide
How to use the Auto-segment tool

Good example

All pixels in the image that are part of an animal have been colored with the appropriate label color.

Bad example

Some animals in the image have not been colored in completely.

The color for a given animal extends beyond the boundaries of the animal.

For each animal in the image, label the animal with the appropriate label color.
Workers can use View full instructions to learn how to use the tool. Workers will need to place a point on four extreme-points (top-most, bottom-most, left-most, and right-most points) of the object of interest, and the tool will automatically generate a mask for the object. Workers can further-refine the mask using the other tools provided, or by using the auto-segment tool on smaller portions of the object that were missed.

**Tool Availability**

The auto-segmentation tool automatically appears in your workers' consoles if you create a semantic segmentation labeling job using the Amazon SageMaker console. While creating a semantic segmentation job in the SageMaker console, you will be able to preview the tool while creating worker instructions. To learn how to create a semantic segmentation labeling job in the SageMaker console, see Getting started (p. 162).

If you are creating a custom instance segmentation labeling job in the SageMaker console or creating an instance- or semantic-segmentation labeling job using the Ground Truth API, you need to create a custom task template to design your worker console and instructions. To include the auto-segmentation tool in your worker console, ensure that the following conditions are met in your custom task template:

- For semantic segmentation labeling jobs created using the API, the `<crowd-semantic-segmentation>` tag is present in the task template. For custom instance segmentation labeling jobs, the `<crowd-instance-segmentation>` tag is present in the task template.
- The task is assigned to a private workforce or vendor workforce.
- The images to be labeled are Amazon Simple Storage Service Amazon S3 objects that have been pre-signed for the Worker so that they can access it. This is true if the task template includes the `grant_read_access` filter. For information about the `grant_read_access` filter, see Adding automation with Liquid (p. 295).

The following is an example of a custom task template for a custom instance segmentation labeling job, which includes the `<crowd-instance-segmentation/>` tag and the `grant_read_access` Liquid filter.

```html
<script src="https://assets.crowd.aws/crowd-html-elements.js"></script>
<crowd-form>
  <crowd-instance-segmentation
    name="crowd-instance-segmentation"
    src="{{ task.input.taskObject | grant_read_access }}"
    labels="['Car','Road']"
  >
    <full-instructions header="Segmentation instructions">
      Segment each instance of each class of objects in the image.
    </full-instructions>
    <short-instructions>
      <p>Segment each instance of each class of objects in the image.</p>
      <h3 style="color: green">GOOD EXAMPLES</h3>
      <img src="path/to/image.jpg" style="width: 100%"
      >Good because A, B, C.</p>
      <h3 style="color: red">BAD EXAMPLES</h3>
      <img src="path/to/image.jpg" style="width: 100%"
      >Bad because X, Y, Z.</p>
    </short-instructions>
  </crowd-instance-segmentation>
</crowd-form>
```
Image Classification (Single Label)

Use an Amazon SageMaker Ground Truth image classification labeling task when you need workers to classify images using predefined labels that you specify. Workers are shown images and are asked to choose one label for each image.

You can create an image classification labeling job using the Ground Truth section of the Amazon SageMaker console or the CreateLabelingJob operation.

Important
For this task type, if you create your own manifest file, use "source-ref" to identify the location of each image file in Amazon S3 that you want labeled. For more information, see Input Data (p. 341).

Create an Image Classification Labeling Job (Console)

You can follow the instructions Create a Labeling Job (Console) (p. 314) to learn how to create a image classification labeling job in the SageMaker console. In Step 10, choose Image from the Task category drop down menu, and choose Image Classification (Single Label) as the task type.

Ground Truth provides a worker UI similar to the following for labeling tasks. When you create the labeling job with the console, you specify instructions to help workers complete the job and labels that workers can choose from.

Create an Image Classification Labeling Job (API)

To create an image classification labeling job, use the SageMaker API operation CreateLabelingJob. This API defines this operation for all AWS SDKs. To see a list of language-specific SDKs supported for this operation, review the See Also section of CreateLabelingJob.
Follow the instructions on Create a Labeling Job (API) (p. 317) and do the following while you configure your request:

- Pre-annotation Lambda functions for this task type end with PRE-ImageMultiClass. To find the pre-annotation Lambda ARN for your Region, see PreHumanTaskLambdaArn.
- Annotation-consolidation Lambda functions for this task type end with ACS-ImageMultiClass. To find the annotation-consolidation Lambda ARN for your Region, see AnnotationConsolidationLambdaArn.

The following is an example of an AWS Python SDK (Boto3) request to create a labeling job in the US East (N. Virginia) Region. All parameters in red should be replaced with your specifications and resources.

```python
response = client.create_labeling_job(
    LabelingJobName='example-image-classification-labeling-job',
    LabelAttributeName='label',
    InputConfig={
        'DataSource': {
            'S3DataSource': {
                'ManifestS3Uri': 's3://bucket/path/manifest-with-input-data.json'
            }
        },
        'DataAttributes': {
            'ContentClassifiers': [
                'FreeOfPersonallyIdentifiableInformation', 'FreeOfAdultContent'
            ]
        }
    },
    OutputConfig={
        'S3OutputPath': 's3://bucket/path/file-to-store-output-data',
        'KmsKeyId': 'string'
    },
    RoleArn='arn:aws:iam::*:role/*',
    LabelCategoryConfigS3Uri='s3://bucket/path/label-categories.json',
    StoppingConditions={
        'MaxHumanLabeledObjectCount': 123,
        'MaxPercentageOfInputDatasetLabeled': 123
    },
    HumanTaskConfig={
        'WorkteamArn': 'arn:aws:sagemaker:region:*:workteam/private-crowd/*',
        'UiConfig': {
            'UiTemplateS3Uri': 's3://bucket/path/worker-task-template.html'
        },
        'TaskKeywords': [
            'Image classification'
        ],
        'TaskTitle': 'Image classification task',
        'TaskDescription': 'Carefully inspect the image and classify it by selecting one label from the categories provided.',
        'NumberOfHumanWorkersPerDataObject': 123,
        'TaskTimeLimitInSeconds': 123,
        'TaskAvailabilityLifetimeInSeconds': 123,
        'MaxConcurrentTaskCount': 123,
        'AnnotationConsolidationConfig': {
            'AnnotationConsolidationLambdaArn': 'arn:aws:lambda:us-east-1:432418664414:function:ACS-ImageMultiClass'
        },
        Tags=[
            { 'Key': 'string', 'Value': 'string'
        }]
)`
Provide a Template for Image Classification Labeling Jobs

If you create a labeling job using the API, you must supply a worker task template in `UiTemplateS3Uri`. Copy and modify the following template. Only modify the `short-instructions`, `full-instructions`, and `header`.

Upload this template to S3, and provide the S3 URI for this file in `UiTemplateS3Uri`.

```html
<script src="https://assets.crowd.aws/crowd-html-elements.js"></script>
<crowd-form>
  <crowd-image-classifier
    name="crowd-image-classifier"
    src="{{ task.input.taskObject | grant_read_access }}"
    header="please classify"
    categories="{{ task.input.labels | to_json | escape }}""
  >
    <full-instructions header="Image classification instructions">
      <ol>
        <li><strong>Read</strong> the task carefully and inspect the image.</li>
        <li><strong>Read</strong> the options and review the examples provided to understand more about the labels.</li>
        <li><strong>Choose</strong> the appropriate label that best suits the image.</li>
      </ol>
    </full-instructions>
    <short-instructions>
      <h3><span style="color: rgb(0, 138, 0);">Good example</span></h3>
      <p>Enter description to explain the correct label to the workers</p>
      <h3><span style="color: rgb(230, 0, 0);">Bad example</span></h3>
      <p>Enter description of an incorrect label</p>
    </short-instructions>
  </crowd-image-classifier>
</crowd-form>
```

Image Classification Output Data

Once you have created an image classification labeling job, your output data will be located in the Amazon S3 bucket specified in the `S3OutputPath` parameter when using the API or in the `Output dataset location` field of the `Job overview` section of the console.

To learn more about the output manifest file generated by Ground Truth and the file structure the Ground Truth uses to store your output data, see Output Data (p. 381).

To see an example of an output manifest file from an image classification labeling job, see Classification Job Output (p. 385).

Image Classification (Multi-label)

Use an Amazon SageMaker Ground Truth multi-label image classification labeling task when you need workers to classify multiple objects in an image. For example, the following image features a dog and a cat. You can use multi-label image classification to associate the labels "dog" and "cat" with this image.
When working on a multi-label image classification task, workers should choose all applicable labels, but must choose at least one. When creating a job using this task type, you can provide up to 50 label-categories.

When creating a labeling job in the console, Ground Truth doesn't provide a "none" category for when none of the labels applies to an image. To provide this option to workers, include a label similar to "none" or "other" when you create a multi-label image classification job.

To restrict workers to choosing a single label for each image, use the Image Classification (Single Label) (p. 178) task type.

**Important**
For this task type, if you create your own manifest file, use "source-ref" to identify the location of each image file in Amazon S3 that you want labeled. For more information, see Input Data (p. 341).

**Create a Multi-Label Image Classification Labeling Job (Console)**

You can follow the instructions Create a Labeling Job (Console) (p. 314) to learn how to create a multi-label image classification labeling job in the SageMaker console. In Step 10, choose Image from the Task category drop down menu, and choose Image Classification (Multi-label) as the task type.

Ground Truth provides a worker UI similar to the following for labeling tasks. When you create a labeling job in the console, you specify instructions to help workers complete the job and labels that workers can choose from.
Create a Multi-Label Image Classification Labeling Job (API)

To create a multi-label image classification labeling job, use the SageMaker API operation CreateLabelingJob. This API defines this operation for all AWS SDKs. To see a list of language-specific SDKs supported for this operation, review the See Also section of CreateLabelingJob.

Follow the instructions on Create a Labeling Job (API) (p. 317) and do the following while you configure your request:

- Pre-annotation Lambda functions for this task type end with PRE-ImageMultiClassMultiLabel. To find the pre-annotation Lambda ARN for your Region, see PreHumanTaskLambdaArn.
- Annotation-consolidation Lambda functions for this task type end with ACS-ImageMultiClassMultiLabel. To find the annotation-consolidation Lambda ARN for your Region, see AnnotationConsolidationLambdaArn.

The following is an example of an AWS Python SDK (Boto3) request to create a labeling job in the US East (N. Virginia) Region. All parameters in red should be replaced with your specifications and resources.

```python
response = client.create_labeling_job(
    LabelingJobName='example-multi-label-image-classification-labeling-job',
    LabelAttributeName='label',
    InputConfig={
        'DataSource': {
            'S3DataSource': {
                'ManifestS3Uri': 's3://bucket/path/manifest-with-input-data.json'
            }
        },
        'DataAttributes': {
            'ContentClassifiers': [
                'FreeOfPersonallyIdentifiableInformation', 'FreeOfAdultContent'
            ]
        }
    }
)
```
OutputConfig={
    'S3OutputPath': 's3://bucket/path/file-to-store-output-data',
    'KmsKeyId': 'string'
},
RoleArn='arn:aws:iam::*:role/*,
LabelCategoryConfigS3Uri='s3://bucket/path/label-categories.json',
StoppingConditions={
    'MaxHumanLabeledObjectCount': 123,
    'MaxPercentageOfInputDatasetLabeled': 123
},
HumanTaskConfig={
    'WorkteamArn': 'arn:aws:sagemaker:region::*:workteam/private-crowd/*',
    'UiConfig': {
        'UiTemplateS3Uri': 's3://bucket/path/worker-task-template.html'
    },
    'TaskKeywords': [
        'Image Classification',
    ],
    'TaskTitle': 'Multi-label image classification task',
    'TaskDescription': 'Select all labels that apply to the images shown',
    'NumberOfHumanWorkersPerDataObject': 123,
    'TaskTimeLimitInSeconds': 123,
    'TaskAvailabilityLifetimeInSeconds': 123,
    'MaxConcurrentTaskCount': 123,
    'AnnotationConsolidationConfig': {
        'AnnotationConsolidationLambdaArn': 'arn:aws:lambda:us-east-1:432418664414:function:ACS-ImageMultiClassMultiLabel'
    },
    Tags=[
        {'Key': 'string',
         'Value': 'string'
        }
    ]
}

Provide a Template for Multi-label Image Classification

If you create a labeling job using the API, you must supply a worker task template in UiTemplateS3Uri. Copy and modify the following template. Only modify the short-instructions, full-instructions, and header.

Upload this template to S3, and provide the S3 URI for this file in UiTemplateS3Uri.

<script src="https://assets.crowd.aws/crowd-html-elements.js"></script>
<crowd-form>
  <crowd-image-classifier-multi-select
    name="crowd-image-classifier-multi-select"
    src="{{ task.input.taskObject | grant_read_access }}"
    header="Please identify all classes in image"
    categories="{{ task.input.labels | to_json | escape }}"
  >
    <full-instructions header="Multi Label Image classification instructions">
      <ol>
        <li><strong>Read</strong> the task carefully and inspect the image.</li>
        <li><strong>Read</strong> the options and review the examples provided to understand more about the labels.</li>
        <li><strong>Choose</strong> the appropriate labels that best suit the image.</li>
      </ol>
    </full-instructions>
  </crowd-image-classifier-multi-select>
</crowd-form>
Multi-label Image Classification Output Data

Once you have created a multi-label image classification labeling job, your output data will be located in the Amazon S3 bucket specified in the S3OutputPath parameter when using the API or in the Output dataset location field of the Job overview section of the console.

To learn more about the output manifest file generated by Ground Truth and the file structure the Ground Truth uses to store your output data, see Output Data (p. 381).

To see an example of output manifest files for multi-label image classification labeling job, see Multi-label Classification Job Output (p. 385).

Image Label Verification

Building a highly accurate training dataset for your machine learning (ML) algorithm is an iterative process. Typically, you review and continuously adjust your labels until you are satisfied that they accurately represent the ground truth, or what is directly observable in the real world.

You can use an Amazon SageMaker Ground Truth image label verification task to direct workers to review a dataset’s labels and improve label accuracy. Workers can indicate if the existing labels are correct or rate label quality. They can also add comments to explain their reasoning. Amazon SageMaker Ground Truth supports label verification for Bounding Box (p. 166) and Image Semantic Segmentation (p. 171) labels.

You create an image label verification labeling job using the Ground Truth section of the Amazon SageMaker console or the CreateLabelingJob operation.

Ground Truth provides a worker console similar to the following for labeling tasks. When you create the labeling job with the console, you can modify the images and content that are shown. To learn how to create a labeling job using the Ground Truth console, see Create a Labeling Job (Console) (p. 314).
You can create a label verification labeling job using the SageMaker console or API. To learn how to create a labeling job using the Ground Truth API operation `CreateLabelingJob`, see [Create a Labeling Job (API) (p. 317)](#).

### Use Ground Truth to Label Text

Use Ground Truth to text. Select one of the following built-in task types to learn more about that task type. Each page includes instructions to help you create a labeling job using that task type.

#### Topics
- Named Entity Recognition (p. 185)
- Text Classification (Single Label) (p. 188)
- Text Classification (Multi-label) (p. 191)

### Named Entity Recognition

To extract information from unstructured text and classify it into predefined categories, use an Amazon SageMaker Ground Truth named entity recognition (NER) labeling task. Traditionally, NER involves sifting through text data to locate noun phrases, called named entities, and categorizing each with a label, such as "person," "organization," or "brand." You can broaden this task to label longer spans of text and categorize those sequences with predefined labels that you specify.

When tasked with a named entity recognition labeling job, workers apply your labels to specific words or phrases within a larger text block. They choose a label, then apply it by using the cursor to highlight the part of the text to which the label applies. Workers can't apply multiple labels to the same text, and labels can't overlap.

You can create a named entity recognition labeling job using the Ground Truth section of the Amazon SageMaker console or the `CreateLabelingJob` operation.
Important
If you manually create an input manifest file, use "source" to identify the text that you want labeled. For more information, see Input Data (p. 341).

Create a Named Entity Recognition Labeling Job (Console)

You can follow the instructions Create a Labeling Job (Console) (p. 314) to learn how to create a named entity recognition labeling job in the SageMaker console. In Step 10, choose Text from the Task category drop down menu, and choose Named entity recognition as the task type.

Ground Truth provides a worker UI similar to the following for labeling tasks. When you create the labeling job with the console, you specify instructions to help workers complete the job and labels that workers can choose from.

Create a Named Entity Recognition Labeling Job (API)

To create a named entity recognition labeling job, using the SageMaker API operation CreateLabelingJob. This API defines this operation for all AWS SDKs. To see a list of language-specific SDKs supported for this operation, review the See Also section of CreateLabelingJob.

Follow the instructions on Create a Labeling Job (API) (p. 317) and do the following while you configure your request:

- Pre-annotation Lambda functions for this task type end with PRE-NamedEntityRecognition. To find the pre-annotation Lambda ARN for your Region, see PreHumanTaskLambdaArn.
- Annotation-consolidation Lambda functions for this task type end with ACS-NamedEntityRecognition. To find the annotation-consolidation Lambda ARN for your Region, see AnnotationConsolidationLambdaArn.

The following is an example of an AWS Python SDK (Boto3) request to create a labeling job in the US East (N. Virginia) Region. All parameters in red should be replaced with your specifications and resources.
response = client.create_labeling_job(
    LabelingJobName='example-ner-labeling-job',
    LabelAttributeName='label',
    InputConfig={
        'DataSource': {
            'S3DataSource': {
                'ManifestS3Uri': 's3://bucket/path/manifest-with-input-data.json'
            }
        },
        'DataAttributes': {
            'ContentClassifiers': [
                'FreeOfPersonallyIdentifiableInformation', 'FreeOfAdultContent'
            ]
        }
    },
    OutputConfig={
        'S3OutputPath': 's3://bucket/path/file-to-store-output-data',
        'KmsKeyId': 'string'
    },
    RoleArn='arn:aws:iam::*:role/*',
    LabelCategoryConfigS3Uri='s3://bucket/path/label-categories.json',
    StoppingConditions={
        'MaxHumanLabeledObjectCount': 123,
        'MaxPercentageOfInputDatasetLabeled': 123
    },
    HumanTaskConfig={
        'WorkteamArn': 'arn:aws:sagemaker:region::*:workteam/private-crowd/*',
        'UiConfig': {
            'UiTemplateS3Uri': 's3://bucket/path/worker-task-template.html'
        },
        'TaskKeywords': ['Named entity Recognition'],
        'TaskTitle': 'Named entity Recognition task',
        'TaskDescription': 'Apply the labels provided to specific words or phrases within the larger text block.',
        'NumberOfHumanWorkersPerDataObject': 123,
        'TaskTimeLimitInSeconds': 123,
        'TaskAvailabilityLifetimeInSeconds': 123,
        'MaxConcurrentTaskCount': 123,
        'AnnotationConsolidationConfig': {
        },
        Tags=[
            {
                'Key': 'string',
                'Value': 'string'
            }
        ]
    })

Provide a Template for Named Entity Recognition Labeling Jobs

If you create a labeling job using the API, you must supply a worker task template in UiTemplateS3Uri. Copy and modify the following template. Only modify the short-instructions, full-instructions, and header.

Upload this template to S3, and provide the S3 URI for this file in UiTemplateS3Uri.
Named Entity Recognition Output Data

Once you have created a named entity recognition labeling job, your output data will be located in the Amazon S3 bucket specified in the S3OutputPath parameter when using the API or in the Output dataset location field of the Job overview section of the console.

To learn more about the output manifest file generated by Ground Truth and the file structure the Ground Truth uses to store your output data, see Output Data (p. 381).

Text Classification (Single Label)

To categorize articles and text into predefined categories, use text classification. For example, you can use text classification to identify the sentiment conveyed in a review or the emotion underlying a section of text. Use Amazon SageMaker Ground Truth text classification to have workers sort text into categories that you define.

You create a text classification labeling job using the Ground Truth section of the Amazon SageMaker console or the CreateLabelingJob operation.

**Important**

If you manually create an input manifest file, use "source" to identify the text that you want labeled. For more information, see Input Data (p. 341).

Create a Text Classification Labeling Job (Console)

You can follow the instructions Create a Labeling Job (Console) (p. 314) to learn how to create a text classification labeling job in the SageMaker console. In Step 10, choose Text from the Task category drop down menu, and choose Text Classification (Single Label) as the task type.

Ground Truth provides a worker UI similar to the following for labeling tasks. When you create the labeling job with the console, you specify instructions to help workers complete the job and labels that workers can choose from.
Create a Text Classification Labeling Job (API)

To create a text classification labeling job, use the SageMaker API operation CreateLabelingJob. This API defines this operation for all AWS SDKs. To see a list of language-specific SDKs supported for this operation, review the See Also section of CreateLabelingJob.

Follow the instructions on Create a Labeling Job (API) (p. 317) and do the following while you configure your request:

- Pre-annotation Lambda functions for this task type end with `PRE-TextMultiClass`. To find the pre-annotation Lambda ARN for your Region, see PreHumanTaskLambdaArn.
- Annotation-consolidation Lambda functions for this task type end with `ACS-TextMultiClass`. To find the annotation-consolidation Lambda ARN for your Region, see AnnotationConsolidationLambdaArn.

The following is an example of an AWS Python SDK (Boto3) request to create a labeling job in the US East (N. Virginia) Region. All parameters in red should be replaced with your specifications and resources.

```python
response = client.create_labeling_job(
    LabelingJobName='example-text-classification-labeling-job,
    LabelAttributeName='label',
    InputConfig={
        'DataSource': {
            'S3DataSource': {
                'ManifestS3Uri': 's3://bucket/path/manifest-with-input-data.json'
            }
        },
        'DataAttributes': {
            'ContentClassifiers': [
                'FreeOfPersonallyIdentifiableInformation'|'FreeOfAdultContent',
            ]
        }
    }
)```
Provide a Template for Text Classification Labeling Jobs

If you create a labeling job using the API, you must supply a worker task template in `UiTemplateS3Uri`. Copy and modify the following template. Only modify the `short-instructions`, `full-instructions`, and `header`.

Upload this template to S3, and provide the S3 URI for this file in `UiTemplateS3Uri`.

```html
<script src="https://assets.crowd.aws/crowd-html-elements.js"></script>

<crowd-form>
  <crowd-classifier
    name="crowd-classifier"
    categories="{{ task.input.labels | to_json | escape }}"
    header="classify text"
  >
    <classification-target style="white-space: pre-wrap">
      {{ task.input.taskObject }}
    </classification-target>
    <full-instructions header="Classifier instructions">
      <ol>
        <li><strong>Read</strong> the text carefully.</li>
        <li><strong>Read</strong> the examples to understand more about the options.</li>
        <li><strong>Choose</strong> the appropriate labels that best suit the text.</li>
      </ol>
    </full-instructions>
  </crowd-classifier>
</crowd-form>
```
Text Classification Output Data

Once you have created a text classification labeling job, your output data will be located in the Amazon S3 bucket specified in the S3OutputPath parameter when using the API or in the Output dataset location field of the Job overview section of the console.

To learn more about the output manifest file generated by Ground Truth and the file structure the Ground Truth uses to store your output data, see Output Data (p. 381).

To see an example of an output manifest files from a text classification labeling job, see Classification Job Output (p. 385).

Text Classification (Multi-label)

To categorize articles and text into multiple predefined categories, use the multi-label text classification task type. For example, you can use this task type to identify more than one emotion conveyed in text.

When working on a multi-label text classification task, workers should choose all applicable labels, but must choose at least one. When creating a job using this task type, you can provide up to 50 label categories.

Amazon SageMaker Ground Truth doesn't provide a "none" category for when none of the labels applies. To provide this option to workers, include a label similar to "none" or "other" when you create a multi-label text classification job.

To restrict workers to choosing a single label for each document or text selection, use the Text Classification (Single Label) (p. 188) task type.

Important
If you manually create an input manifest file, use "source" to identify the text that you want labeled. For more information, see Input Data (p. 341).

Create a Multi-Label Text Classification Labeling Job (Console)

You can follow the instructions Create a Labeling Job (Console) (p. 314) to learn how to create a multi-label text classification labeling job in the Amazon SageMaker console. In Step 10, choose Text from the Task category drop down menu, and choose Text Classification (Multi-label) as the task type.

Ground Truth provides a worker UI similar to the following for labeling tasks. When you create the labeling job with the console, you specify instructions to help workers complete the job and labels that workers can choose from.
Create a Multi-Label Text Classification Labeling Job (API)

To create a multi-label text classification labeling job, use the SageMaker API operation `CreateLabelingJob`. This API defines this operation for all AWS SDKs. To see a list of language-specific SDKs supported for this operation, review the `See Also` section of `CreateLabelingJob`.

Follow the instructions on [Create a Labeling Job (API) (p. 317)](#) and do the following while you configure your request:

- Pre-annotation Lambda functions for this task type end with `PRE-TextMultiClassMultiLabel`. To find the pre-annotation Lambda ARN for your Region, see `PreHumanTaskLambdaArn`.
- Annotation-consolidation Lambda functions for this task type end with `ACS-TextMultiClassMultiLabel`. To find the annotation-consolidation Lambda ARN for your Region, see `AnnotationConsolidationLambdaArn`.

The following is an example of an [AWS Python SDK (Boto3) request](#) to create a labeling job in the US East (N. Virginia) Region. All parameters in red should be replaced with your specifications and resources.

```python
response = client.create_labeling_job(
    LabelingJobName='example-multi-label-text-classification-labeling-job,
    LabelAttributeName='label',
    InputConfig={
        'DataSource': {
            'S3DataSource': {
                'ManifestS3Uri': 's3://bucket/path/manifest-with-input-data.json'
            }
        }
    }
)
```
Amazon SageMaker Developer Guide
Label Text
},
'DataAttributes': {
'ContentClassifiers': [
'FreeOfPersonallyIdentifiableInformation'|'FreeOfAdultContent',
]
}

},
OutputConfig={
'S3OutputPath': 's3://bucket/path/file-to-store-output-data',
'KmsKeyId': 'string'
},
RoleArn='arn:aws:iam::*:role/*,
LabelCategoryConfigS3Uri='s3://bucket/path/label-categories.json',
StoppingConditions={
'MaxHumanLabeledObjectCount': 123,
'MaxPercentageOfInputDatasetLabeled': 123
},
HumanTaskConfig={
'WorkteamArn': 'arn:aws:sagemaker:region:*:workteam/private-crowd/*',
'UiConfig': {
'UiTemplateS3Uri': 's3://bucket/path/custom-worker-task-template.html'
},
'PreHumanTaskLambdaArn': 'arn:aws:lambda::function:PRE-TextMultiClassMultiLabel,
'TaskKeywords': [
'Text Classification',
],
'TaskTitle': 'Multi-label text classification task',
'TaskDescription': 'Select all labels that apply to the text shown',
'NumberOfHumanWorkersPerDataObject': 123,
'TaskTimeLimitInSeconds': 123,
'TaskAvailabilityLifetimeInSeconds': 123,
'MaxConcurrentTaskCount': 123,
'AnnotationConsolidationConfig': {
'AnnotationConsolidationLambdaArn': 'arn:aws:lambda:useast-1:432418664414:function:ACS-TextMultiClassMultiLabel'
},
Tags=[
{
'Key': 'string',
'Value': 'string'
},
]
)

Create a Template for Multi-label Text Classiﬁcation
If you create a labeling job using the API, you must supply a worker task template in UiTemplateS3Uri.
Copy and modify the following template. Only modify the short-instructions, fullinstructions, and header.
Upload this template to S3, and provide the S3 URI for this ﬁle in UiTemplateS3Uri.
<script src="https://assets.crowd.aws/crowd-html-elements.js"></script>
<crowd-form>
<crowd-classifier-multi-select
name="crowd-classifier-multi-select"
categories="{{ task.input.labels | to_json | escape }}"
header="Please identify all classes in the below text"
>
<classification-target style="white-space: pre-wrap">
{{ task.input.taskObject }}
</classification-target>
<full-instructions header="Classifier instructions">
<ol><li><strong>Read</strong> the text carefully.</li>

193


To learn how to create a custom template, see Creating Custom Labeling Workflows (p. 291).

**Multi-label Text Classification Output Data**

Once you have created a multi-label text classification labeling job, your output data will be located in the Amazon S3 bucket specified in the `S3OutputPath` parameter when using the API or in the `Output dataset location` field of the `Job overview` section of the console.

To learn more about the output manifest file generated by Ground Truth and the file structure the Ground Truth uses to store your output data, see Output Data (p. 381).

To see an example of output manifest files for multi-label text classification labeling job, see Multi-label Classification Job Output (p. 385).

**Label Videos and Video Frames**

You can use Ground Truth to classify videos and annotate video frames (still images extracted from videos) using one of the three built-in video task types. These task types streamline the process of creating video and video frame labeling jobs using the Amazon SageMaker console, API, and language-specific SDKs.

- **Video clip classification** – Enable workers to classify videos into categories you specify. For example, you can use this task type to have workers categorize videos into topics like sports, comedy, music, and education. To learn more, see Video Classification (p. 195).
- **Video frame labeling jobs** – Enable workers to annotate video frames extracted from a video using bounding boxes, polylines, polygons or keypoint annotation tools. Ground Truth offers two built-in task types to label video frames:
  - Enable workers to identify and locate objects in video frames using *video frame object detection*.
  - Enable workers to track the movement of objects across video frames using *video frame object tracking*.

If you have video files, you can use the Ground Truth automatic frame extraction tool to extract video frames from your videos.

To learn more, see Label Video Frames (p. 199).

**Topics**

- Video Classification (p. 195)
- Label Video Frames (p. 199)
- Worker Instructions (p. 212)
Video Classification

Use an Amazon SageMaker Ground Truth video classification labeling task when you need workers to classify videos using predefined labels that you specify. Workers are shown videos and are asked to choose one label for each video.

You create a video classification labeling job using the Ground Truth section of the Amazon SageMaker console or the CreateLabelingJob operation.

Important
For this task type, if you create your own manifest file, use "source-ref" to identify the location of each video file in Amazon S3 that you want labeled. For more information, see Input Data (p. 341).

Create a Video Classification Labeling Job (Console)

You can follow the instructions in Create a Labeling Job (Console) (p. 314) to learn how to create a video classification labeling job in the SageMaker console. In step 10, choose Video from the Task category dropdown list, and choose Video Classification as the task type.

Ground Truth provides a worker UI similar to the following for labeling tasks. When you create a labeling job in the console, you specify instructions to help workers complete the job and labels from which workers can choose.
Instructions

View full instructions

View tool guide

Select a single label that best describes this video clip. Select none of the above if none of the other labels apply.

Select Submit when you are done.
Create a Video Classification Labeling Job (API)

This section covers details you need to know when you create a labeling job using the SageMaker API operation `CreateLabelingJob`. This API defines this operation for all AWS SDKs. To see a list of language-specific SDKs supported for this operation, review the `See Also` section of `CreateLabelingJob`.

Follow the instructions on Create a Labeling Job (API) (p. 317) and do the following while you configure your request:

- Use a pre-annotation Lambda function that ends with `PRE-VideoClassification`. To find the pre-annotation Lambda ARN for your Region, see `PreHumanTaskLambdaArn`.
- Use an annotation-consolidation Lambda function that ends with `ACS-VideoClassification`. To find the annotation-consolidation Lambda ARN for your Region, see `AnnotationConsolidationLambdaArn`.

The following is an example of an AWS Python SDK (Boto3) request to create a labeling job in the US East (N. Virginia) Region.

```python
response = client.create_labeling_job(
    LabelingJobName='example-video-classification-labeling-job',
    LabelAttributeName='label',
    InputConfig={
        'DataSource': {
            'S3DataSource': {
                'ManifestS3Uri': 's3://bucket/path/manifest-with-input-data.json'
            }
        },
        'DataAttributes': {
            'ContentClassifiers': ['FreeOfPersonallyIdentifiableInformation','FreeOfAdultContent']
        }
    },
    OutputConfig={
        'S3OutputPath': 's3://bucket/path/file-to-store-output-data',
        'KmsKeyId': 'string'
    },
    RoleArn='arn:aws:iam::*:role/*',
    LabelCategoryConfigS3Uri='s3://bucket/path/label-categories.json',
    StoppingConditions={
        'MaxHumanLabeledObjectCount': 123,
        'MaxPercentageOfInputDatasetLabeled': 123
    },
    HumanTaskConfig={
        'WorkteamArn': 'arn:aws:sagemaker:region:::workteam/private-crowd/*',
        'UiConfig': {
            'UiTemplateS3Uri': 's3://bucket/path/worker-task-template.html'
        },
        'PreHumanTaskLambdaArn': 'arn:aws:lambda:us-east-1:43241866414:function:PRE-VideoClassification',
        'TaskKeywords': ['Video Classification'],
        'TaskTitle': 'Video classification task',
        'TaskDescription': 'Select a label to classify this video',
        'NumberOfHumanWorkersPerDataObject': 123,
        'TaskTimeLimitInSeconds': 123,
        'TaskAvailabilityLifetimeInSeconds': 123,
        'MaxConcurrentTaskCount': 123,
        'AnnotationConsolidationConfig': {
```
Provide a Template for Video Classification

If you create a labeling job using the API, you must supply a worker task template in `UiTemplateS3Uri`. Copy and modify the following template by modifying the `short-instructions`, `full-instructions`, and `header`. Upload this template to Amazon S3, and provide the Amazon S3 URI to this file in `UiTemplateS3Uri`.

```html
<crowd-form>
  <crowd-classifier
    name="crowd-classifier"
    categories="{{ task.input.labels | to_json | escape }}"
    header="Please classify video"
  >
    <classification-target>
      <video width="100%" controls/>
      <source src="{{ task.input.taskObject | grant_read_access }}" type="video/mp4"/>
      <source src="{{ task.input.taskObject | grant_read_access }}" type="video/webm"/>
      <source src="{{ task.input.taskObject | grant_read_access }}" type="video/ogg"/>
      Your browser does not support the video tag.
    </video>
    <full-instructions header="Video classification instructions">
      <ol><li><strong>Read</strong> the task carefully and inspect the video.</li>
      <li><strong>Read</strong> the options and review the examples provided to understand more about the labels.</li>
      <li><strong>Choose</strong> the appropriate label that best suits the video.</li></ol>
    </full-instructions>
    <short-instructions>
      <h3><span style="color: rgb(0, 138, 0);">Good example</span></h3>
      <p>Enter description to explain the correct label to the workers</p>
      <p><img src="https://d7evko5405gb7.cloudfront.net/fe4fed9b-660c-4477-9294-2c66a15d6bbe/src/images/quick-instructions-example-placeholder.png" style="max-width:100%"></p>
      <h3><span style="color: rgb(230, 0, 0);">Bad example</span></h3>
      <p>Enter description of an incorrect label</p>
      <p><img src="https://d7evko5405gb7.cloudfront.net/fe4fed9b-660c-4477-9294-2c66a15d6bbe/src/images/quick-instructions-example-placeholder.png" style="max-width:100%"></p>
    </short-instructions>
  </classification-target>
</crowd-classifier>
</crowd-form>
```
Video Classification Output Data

Once you have created a video classification labeling job, your output data is located in the Amazon S3 bucket specified in the `S3OutputPath` parameter when using the API or in the **Output dataset location** field of the **Job overview** section of the console.

To learn more about the output manifest file generated by Ground Truth and the file structure the Ground Truth uses to store your output data, see **Output Data** (p. 381).

To see an example of output manifest files for video classification labeling jobs, see **Classification Job Output** (p. 385).

Label Video Frames

You can use Ground Truth built-in video frame task types to have workers annotate video frames using bounding boxes, polylines, polygons or keypoints. A **video frame** is a sequence of images that have been extracted from a video.

If you do not have video frames, you can provide video files (MP4 files) and use the Ground Truth automated frame extraction tool to extract video frames. To learn more, see **Provide Video Files** (p. 377).

You can use the following built-in video task types to create video frame labeling jobs using the Amazon SageMaker console, API, and language-specific SDKs.

- **Video frame object detection** – Use this task type when you want workers to identify and locate objects in sequences of video frames. You provide a list of categories, and workers can select one category at a time and annotate objects which the category applies to in all frames. For example, you can use this task to ask workers to identify and localize various objects in a scene, such as cars, bikes, and pedestrians.

- **Video frame object tracking** – Use this task type when you want workers to track the movement of instances of objects across sequences of video frames. When a worker adds an annotation to a single frame, that annotation is associated with a unique instance ID. The worker adds annotations associated with the same ID in all other frames to identify the same object or person. For example, a worker can track the movement of a vehicle across a sequences of video frames by drawing bounding boxes associated with the same ID around the vehicle in each frame that it appears.

Use the following topics to learn more about these built-in task types and how to create a labeling job using each task type. See **Task Types** (p. 211) to learn more about the annotations tools (bounding boxes, polylines, polygons and keypoints) available for these task types.

Before you create a labeling job, we recommend that you review **Video Frame Labeling Job Overview** (p. 209).

Topics

- **Video Frame Object Detection** (p. 199)
- **Video Frame Object Tracking** (p. 204)
- **Video Frame Labeling Job Overview** (p. 209)

Video Frame Object Detection

You can use the video frame object detection task type to have workers identify and locate objects in a sequence of video frames (images extracted from a video) using bounding boxes, polylines, polygons
or keypoint annotation tools. The tool you choose defines the video frame task type you create. For example, you can use a bounding box video frame object detection task type workers to identify and localize various objects in a series of video frames, such as cars, bikes, and pedestrians.

You can create a video frame object detection labeling job using the Amazon SageMaker Ground Truth console, the SageMaker API, and language-specific AWS SDKs. To learn more, see Create a Video Frame Object Detection Labeling Job (p. 200) and select your preferred method. See Task Types (p. 211) to learn more about the annotations tools you can choose from when you create a labeling job.

Ground Truth provides a worker UI and tools to complete your labeling job tasks: Preview the Worker UI (p. 200).

You can create a job to adjust annotations created in a video object detection labeling job using the video object detection adjustment task type. To learn more, see Create Video Frame Object Detection Adjustment or Verification Labeling Job (p. 203).

**Preview the Worker UI**

Ground Truth provides workers with a web user interface (UI) to complete your video frame object detection annotation tasks. You can preview and interact with the worker UI when you create a labeling job in the console. If you are a new user, we recommend that you create a labeling job through the console using a small input dataset to preview the worker UI and ensure your video frames, labels, and label attributes appear as expected.

The UI provides workers with the following assistive labeling tools to complete your object detection tasks:

- For all tasks, workers can use the **Copy to next** and **Copy to all** features to copy an annotation to the next frame or to all subsequent frames respectively.
- For tasks that include the bounding box tools, workers can use a **Predict next** feature to draw a bounding box in a single frame, and then have Ground Truth predict the location of boxes with the same label in all other frames. Workers can then make adjustments to correct predicted box locations.

**Create a Video Frame Object Detection Labeling Job**

You can create a video frame object detection labeling job using the SageMaker console or the CreateLabelingJob API operation.

This section assumes that you have reviewed the Video Frame Labeling Job Overview (p. 209) and have chosen the type of input data and the input dataset connection you are using.

**Create a Labeling Job (Console)**

You can follow the instructions in Create a Labeling Job (Console) (p. 314) to learn how to create a video frame object tracking job in the SageMaker console. In step 10, choose **Video - Object detection** from the Task category dropdown list. Select the task type you want by selecting one of the cards in Task selection.
**Task type**

**Info**

**Task category**

Select the type of data being labeled to view available task templates for it or select 'Custom'.

- **Video - Object detection**

**Task selection**

Select the task that a human worker will perform to label objects in your dataset.

- **Bounding box**
  
  Get workers to draw bounding boxes around specified objects in your video.  
  
  ![Bounding box example](image)

- **Polyline**
  
  Get workers to draw polyline around specified objects in your video.  
  
  ![Polyline example](image)

- **Keyframes**
  
  Get workers to select keyframes to label video events.

  ![Keyframes example](image)
If you set your take time limit to be greater than 8 hours, you must set `MaxSessionDuration` for your IAM execution role to at least 8 hours. To see how to update this value for your IAM role in the IAM console, see Modifying a Role Maximum Session Duration (Console) in the IAM User Guide.

Create a Labeling Job (API)

You create an object detection labeling job using the SageMaker API operation `CreateLabelingJob`. This API defines this operation for all AWS SDKs. To see a list of language-specific SDKs supported for this operation, review the See Also section of `CreateLabelingJob`.

Create a Labeling Job (API) (p. 317) provides an overview of the `CreateLabelingJob` operation. Follow these instructions and do the following while you configure your request:

- You must enter an ARN for `HumanTaskUiArn`. Use `arn:aws:sagemaker:<region>:394669845002:human-task-ui/VideoObjectDetection`. Replace `<region>` with the AWS Region in which you are creating the labeling job. Do not include an entry for the `UiTemplateS3Uri` parameter.
- Your `LabelAttributeName` must end in `-ref`. For example, `video-od-labels-ref`.
- Your input manifest file must be a video frame sequence manifest file. You can create this manifest file using the SageMaker console, or create it manually and upload it to Amazon S3. For more information, see Input Data Setup (p. 377).
- You can only use private or vendor work teams to create video frame object detection labeling jobs.
- You specify your labels, label category and frame attributes, the task type, and worker instructions in a label category configuration file. Specify the task type (bounding boxes, polylines, polygons or keypoint) using `annotationType` in your label category configuration file. For more information, see Create a Labeling Category Configuration File with Label Category and Frame Attributes (p. 326) to learn how to create this file.
- You need to provide pre-defined ARNs for the pre-annotation and post-annotation (ACS) Lambda functions. These ARNs are specific to the AWS Region you use to create your labeling job.
- To find the pre-annotation Lambda ARN, refer to `PreHumanTaskLambdaArn`. Use the Region in which you are creating your labeling job to find the correct ARN that ends with `PRE-VideoObjectDetection`.
- To find the post-annotation Lambda ARN, refer to `AnnotationConsolidationLambdaArn`. Use the Region in which you are creating your labeling job to find the correct ARN that ends with `ACS-VideoObjectDetection`.
- The number of workers specified in `NumberOfHumanWorkersPerDataObject` must be 1.
- Automated data labeling is not supported for 3D point cloud labeling jobs. Do not specify values for parameters in `LabelingJobAlgorithmsConfig`.
- 3D point cloud object tracking labeling jobs can take multiple hours to complete. You can specify a longer time limit for these labeling jobs in `TaskTimeLimitInSeconds` (up to 7 days, or 604,800 seconds).

Important

If you set your take time limit to be greater than 8 hours, you must set `MaxSessionDuration` for your IAM execution role to at least 8 hours. To see how to update this value for your IAM role, see Modifying a Role in the IAM User Guide, choose your preferred method to modify the role, and then follow the steps in Modifying a Role Maximum Session Duration.

The following is an example of an AWS Python SDK (Boto3) request to create a labeling job in the US East (N. Virginia) Region.

```python
response = client.create_labeling_job(
    LabelingJobName='example-video-od-labeling-job',
    LabelAttributeName='label',
)```

202
Create Video Frame Object Detection Adjustment or Verification Labeling Job

You can create an adjustment labeling job using the Ground Truth console or CreateLabelingJob API. You must use the Ground Truth API to create a video frame object detection verification labeling job. To learn more about adjustment and verification labeling jobs, and to learn how to create one, see Verify and Adjust Labels (p. 287).

Output Data Format

When you create a video frame object detection labeling job, tasks are sent to workers. When these workers complete their tasks, labels are written to the Amazon S3 output location you specified when
you created the labeling job. To learn about the video frame object detection output data format, see Video Frame Object Detection Output (p. 393). If you are a new user of Ground Truth, see Output Data (p. 381) to learn more about the Ground Truth output data format.

**Video Frame Object Tracking**

You can use the video frame object tracking task type to have workers track the movement of objects in a sequence of video frames (images extracted from a video) using bounding boxes, polylines, polygons or keypoint annotation tools. The tool you choose defines the video frame task type you create. For example, you can use a bounding box video frame object tracking task type to ask workers to track the movement of objects, such as cars, bikes, and pedestrians by drawing boxes around them.

You provide a list of categories, and each annotation that a worker adds to a video frame is identified as an instance of that category using an instance ID. For example, if you provide the label category car, the first car that a worker annotates will have the instance ID car:1. The second car the worker annotates will have the instance ID car:2. To track an object's movement, the worker adds annotations associated with the same instance ID around to object in all frames.

You can create a video frame object tracking labeling job using the Amazon SageMaker Ground Truth console, the SageMaker API, and language-specific AWS SDKs. To learn more, see Create a Video Frame Object Detection Labeling Job (p. 200) and select your preferred method. See Task Types (p. 211) to learn more about the annotations tools you can choose from when you create a labeling job.

Ground Truth provides a worker UI and tools to complete your labeling job tasks: Preview the Worker UI (p. 200).

You can create a job to adjust annotations created in a video object detection labeling job using the video object detection adjustment task type. To learn more, see Create Video Frame Object Detection Adjustment or Verification Labeling Job (p. 203).

**Preview the Worker UI**

Ground Truth provides workers with a web user interface (UI) to complete your video frame object tracking annotation tasks. You can preview and interact with the worker UI when you create a labeling job in the console. If you are a new user, we recommend that you create a labeling job through the console using a small input dataset to preview the worker UI and ensure your video frames, labels, and label attributes appear as expected.

The UI provides workers with the following assistive labeling tools to complete your object tracking tasks:

- For all tasks, workers can use the **Copy to next** and **Copy to all** features to copy an annotation with the same unique ID to the next frame or to all subsequent frames respectively.
- For tasks that include the bounding box tools, workers can use a **Predict next** feature to draw a bounding box in a single frame, and then have Ground Truth predict the location of boxes with the same unique ID in all other frames. Workers can then make adjustments to correct predicted box locations.

**Create a Video Frame Object Tracking Labeling Job**

You can create a video frame object tracking labeling job using the SageMaker console or the CreateLabelingJob API operation.

This section assumes that you have reviewed the Video Frame Labeling Job Overview (p. 209) and have chosen the type of input data and the input dataset connection you are using.

**Create a Labeling Job (Console)**

You can follow the instructions in Create a Labeling Job (Console) (p. 314) to learn how to create a video frame object tracking job in the SageMaker console. In step 10, choose Video - Object tracking.
from the Task category dropdown list. Select the task type you want by selecting one of the cards in Task selection.
Task type  Info

Task category
Select the type of data being labeled to view available task templates for it or select 'Custom' to create your own.

Video - Object tracking

Task selection
Select the task that a human worker will perform to label objects in your dataset.

• Bounding box
Get workers to track specific instances of objects in your video across multiple frames in your bounding boxes.  Info

• Polyline
Get workers to track specific instances of objects in your video across multiple frames in your polylines.  Info

• Keyframe
Get workers to indicate important frames in your video.  Info
If you set your take time limit to be greater than 8 hours, you must set **MaxSessionDuration** for your IAM execution role to at least 8 hours. To see how to update this value for your IAM role in the IAM console, see **Modifying a Role Maximum Session Duration (Console)** in the IAM User Guide.

**Create a Labeling Job (API)**

You create an object tracking labeling job using the SageMaker API operation **CreateLabelingJob**. This API defines this operation for all AWS SDKs. To see a list of language-specific SDKs supported for this operation, review the **See Also** section of **CreateLabelingJob**.

**Create a Labeling Job (API)** (p. 317) provides an overview of the **CreateLabelingJob** operation. Follow these instructions and do the following while you configure your request:

- You must enter an ARN for **HumanTaskUiArn**. Use
  
  
  Replace `<region>` with the AWS Region in which you are creating the labeling job.

  Do not include an entry for the **UiTemplateS3Uri** parameter.

- Your **LabelAttributeName** must end in -ref. For example, **ot-labels-ref**.

- Your input manifest file must be a video frame sequence manifest file. You can create this manifest file using the SageMaker console, or create it manually and upload it to Amazon S3. For more information, see **Input Data Setup** (p. 377). If you create a streaming labeling job, the input manifest file is optional.

- You can only use private or vendor work teams to create video frame object detection labeling jobs.

- You specify your labels, label category and frame attributes, the task type, and worker instructions in a label category configuration file. Specify the task type (bounding boxes, polylines, polygons or keypoint) using **annotationType** in your label category configuration file. For more information, see **Create a Labeling Category Configuration File with Label Category and Frame Attributes** (p. 326) to learn how to create this file.

- You need to provide pre-defined ARNs for the pre-annotation and post-annotation (ACS) Lambda functions. These ARNs are specific to the AWS Region you use to create your labeling job.

  To find the pre-annotation Lambda ARN, refer to **PreHumanTaskLambdaArn**. Use the Region in which you are creating your labeling job to find the correct ARN that ends with **PRE-VideoObjectTracking**.

  To find the post-annotation Lambda ARN, refer to **AnnotationConsolidationLambdaArn**. Use the Region in which you are creating your labeling job to find the correct ARN that ends with **ACS-VideoObjectTracking**.

- The number of workers specified in **NumberOfHumanWorkersPerDataObject** must be 1.

- Automated data labeling is not supported for 3D point cloud labeling jobs. Do not specify values for parameters in **LabelingJobAlgorithmsConfig**.

- 3D point cloud object tracking labeling jobs can take multiple hours to complete. You can specify a longer time limit for these labeling jobs in **TaskTimeLimitInSeconds** (up to 7 days, or 604,800 seconds).

  **Important**

  If you set your take time limit to be greater than 8 hours, you must set **MaxSessionDuration** for your IAM execution role to at least 8 hours. To see how to update this value for your IAM role, see **Modifying a Role** in the IAM User Guide, choose your preferred method to modify the role, and then follow the steps in **Modifying a Role Maximum Session Duration**.

The following is an example of an **AWS Python SDK (Boto3)** request to create a labeling job in the US East (N. Virginia) Region.

```python
response = client.create_labeling_job(
```
Create a Video Frame Object Tracking Adjustment or Verification Labeling Job

You can create an adjustment labeling job using the Ground Truth console or CreateLabelingJob API. You must use the Ground Truth API to create a video frame object tracking verification labeling job. To learn more about adjustment and verification labeling jobs, and to learn how create one, see Verify and Adjust Labels (p. 287).
Output Data Format

When you create a video frame object tracking labeling job, tasks are sent to workers. When these workers complete their tasks, labels are written to the Amazon S3 output location you specified when you created the labeling job. To learn about the video frame object tracking output data format, see Video Frame Object Tracking Output (p. 395). If you are a new user of Ground Truth, see Output Data (p. 381) to learn more about the Ground Truth output data format.

Video Frame Labeling Job Overview

Use this page to learn about the object detection and object tracking video frame labeling jobs. The information on this page applies to both of these built-in task types.

The video frame labeling job is unique because of the following:

• You can either provide data objects that are ready to be annotated (video frames), or you can provide video files and have Ground Truth automatically extract video frames.

• Workers have the ability to save work as they go.

• You cannot use the Amazon Mechanical Turk workforce to complete your labeling tasks.

• Ground Truth provides a worker UI, as well as assistive and basic labeling tools, to help workers complete your tasks. You do not need to provide a worker task template.

Use the following topics to learn more.

Topics

• Input Data (p. 209)
• Job Completion Times (p. 209)
• Workforces (p. 210)
• Worker User Interface (UI) (p. 210)

Input Data

The video frame labeling job uses sequences of video frames. A single sequence is a series of images that have been extracted from a single video. You can either provide your own sequences of video frames, or have Ground Truth automatically extract video frame sequences from your video files. To learn more, see Provide Video Files (p. 377).

Ground Truth uses sequence files to identify all images in a single sequence. All of the sequences that you want to include in a single labeling job are identified in an input manifest file. Each sequence is used to create a single worker task. You can automatically create sequence files and an input manifest file using Ground Truth automatic data setup. To learn more, see Automated Video Frame Input Data Setup (p. 378).

To learn how to manually create sequence files and an input manifest file, see Create a Video Frame Input Manifest File (p. 379).

Job Completion Times

Video and video frame labeling jobs can take workers hours to complete. You can set the total amount of time that workers can work on each task when you create a labeling job. The maximum time you can set for workers to work on tasks is 7 days. The default value is 3 days.

We strongly recommend that you create tasks that workers can complete within 12 hours. Workers must keep the worker UI open while working on a task. They can save work as they go and Ground Truth saves their work every 15 minutes.
When using the SageMaker CreateLabelingJob API operation, set the total time a task is available to workers in the TaskTimeLimitInSeconds parameter of HumanTaskConfig.

When you create a labeling job in the console, you can specify this time limit when you select your workforce type and your work team.

**Important**
If you set your task time limit to be greater than 8 hours, you must set MaxSessionDuration for your IAM execution role to at least 8 hours.

**Workforces**

When you create a video frame labeling job, you need to specify a work team to complete your annotation tasks. You can choose a work team from a private workforce of your own workers, or from a vendor workforce that you select in the AWS Marketplace. You cannot use the Amazon Mechanical Turk workforce for video frame labeling jobs.

To learn more about vendor workforces, see Managing Vendor Workforces (p. 429).
To learn how to create and manage a private workforce, see Use a Private Workforce (p. 430).

**Worker User Interface (UI)**

Ground Truth provides a worker user interface (UI), tools, and assistive labeling features to help workers complete your video labeling tasks.

You can preview the worker UI when you create a labeling job in the console.

When you create a labeling job using the API operation CreateLabelingJob, you must provide an ARN provided by Ground Truth in the parameter HumanTaskUiArn to specify the worker UI for your task type. You can use HumanTaskUiArn with the SageMaker RenderUiTemplate API operation to preview the worker UI.

You provide worker instructions, labels, and optionally, attributes that workers can use to provide more information about labels (label category attributes). They are all displayed in the worker UI.

**Label Category and Frame Attributes**

When you create a video object tracking or video object detection labeling job, you can add one or more label category attributes and frame attributes:

- **Label category attribute** – A list of options (strings), a free form text box, or a numeric field associated with one or more labels. It is used by workers to provide metadata about a label.
- **Frame attribute** – A list of options (strings), a free form text box, or a numeric field that appears on each video frame a worker is sent to annotate. It is used by workers to provide metadata about video frames.

Use the following sections to learn more about these attributes. To learn how to add label category attributes to a labeling job, use the Create Labeling Job sections on the task type page (p. 199) of your choice.

**Label Category Attributes**

Add label category attributes to labels to give workers the ability to provide more information about the annotations they create.

For example, if you add the label category car, you might also want to capture additional data about your labeled cars, such as if they are occluded or the size of the car. You can capture this metadata using label category attributes. In this example, if you added the attribute occluded to the car label category, you can assign partial, completely, no to the occluded attribute and enable workers to select one of these options.
A label category attribute is added to an individual label, or to all labels. When a label category attribute is applied to all labels it is referred to as a *global label category attribute*.

**Label Frame Attributes**

Add frame attributes to give workers the ability to provide more information about individual video frames.

For example, you can add a number-frame attribute to have workers identify the number of objects they see in a particular frame.

In another example, you may want to provide a free-form text box to give workers the ability to provide an answer to a question.

Each frame attribute you add appears on all frames.

**Task Types**

When you create a video object tracking or video object detection labeling job, you specify the type of annotation that you want workers to create while working on your labeling task. The annotation type determines the type of output data Ground Truth returns and defines the *task type* for your labeling job.

If you are creating a labeling job using the API operation `CreateLabelingJob`, you specify the task type using the label category configuration file parameter `annotationType`. To learn more, see Create a Labeling Category Configuration File with Label Category and Frame Attributes (p. 326).

The following task types are available for both video object tracking or video object detection labeling jobs:

- **Bounding box** – Workers are provided with tools to create bounding box annotations. A bounding box is a box that a worker draws around an objects to identify the pixel-location and label of that object in the frame.

- **Polyline** – Workers are provided with tools to create polyline annotations. A polyline is defined by the series of ordered x, y coordinates. Each point added to the polyline is connected to the previous point by a line. The polyline does not have to be closed (the start point and end point do not have to be the same) and there are no restrictions on the angles formed between lines.

- **Polygon** – Workers are provided with tools to create polygon annotations. A polygon is a closed shape defined by a series of ordered x, y coordinates. Each point added to the polygon is connected to the previous point by a line and there are no restrictions on the angles formed between lines. Two lines (sides) of the polygon cannot cross. The start and end point of a polygon must be the same.

- **Keypoint** – Workers are provided with tools to create keypoint annotations. A keypoint is a single point associated with an x, y coordinate in the video frame.

**Worker Instructions**

You can provide worker instructions to help your workers complete your video frame labeling tasks. You might want to cover the following topics when writing your instructions:

- Best practices and things to avoid when annotating objects.
- The label category attributes provided (for object detection and object tracking tasks) and how to use them.
- How to save time while labeling by using keyboard shortcuts.

You can add your worker instructions using the SageMaker console while creating a labeling job. If you create a labeling job using the API operation `CreateLabelingJob`, you specify worker instructions in your label category configuration file.
In addition to your instructions, Ground Truth provides a link to help workers navigate and use the worker portal. View these instructions by selecting the task type on Worker Instructions (p. 212).

**Worker Instructions**

This topic provides an overview of the Ground Truth worker portal and the tools available to complete your video frame labeling task. First, select the type of task you are working on from Topics.

**Important**  
It is recommended that you complete your task using a Google Chrome web browser.

For adjustment jobs, select the original labeling job task type that produced the labels you are adjusting. Review and adjust the labels in your task as needed.

**Topics**
- Work on Video Frame Object Tracking Tasks (p. 212)
- Work on Video Frame Object Detection Tasks (p. 218)

**Work on Video Frame Object Tracking Tasks**

Video frame object tracking tasks require you to track the movement of objects across video frames. A video frame is a still image from a video scene.

You can use the worker UI to navigate between video frames and use the tools provided to identify unique objects and track their movement from one frame to the next. Use this page to learn how to navigate your worker UI, use the tools provided, and complete your task.

It is recommended that you complete your task using a Google Chrome web browser.

**Important**  
If you see annotations have already been added to one or more video frames when you open your task, adjust those annotations and add additional annotations as needed.

**Topics**
- Your Task (p. 212)
- Navigate the UI (p. 214)
- Tool Guide (p. 214)
- Icons Guide (p. 217)
- Shortcuts (p. 218)
- Saving Your Work and Submitting (p. 218)

**Your Task**

When you work on a video frame object tracking task, you need to select a category from the **Label category** menu on the right side of your worker portal to start annotating. After you've chosen a category, use the tools provided to annotate the objects that the category applies to. This annotation will be associated with a unique label ID that should only be used for that object. Use this same label ID to create additional annotations for the same object in all of the video frames that it appears in. Refer to Tool Guide (p. 214) to learn more about the tools provided.

After you've added a label, you may see a downward pointing arrow next to the label in the **Labels** menu. Select this arrow and then select one option for each label attribute you see to provide more information about that label.

You may see frame attributes under the **Labels** menu. These attributes will appear on each frame in your task. Use these attribute prompts to enter additional information about each frame.
After you've added a label, you can quickly add and edit a label category attribute value by using the downward pointing arrow next to the label in the **Labels** menu. If you select the pencil icon next to the label in the **Labels** menu, the **Edit instance** menu will appear. You can edit the label ID, label category, and label category attributes using this menu.

To edit an annotation, select the label of the annotation that you want to edit in the **Labels** menu or select the annotation in the frame. When you edit or delete an annotation, the action will only modify the annotation in a single frame.

If you are working on a task that includes a bounding box tool, use the predict next icon to predict the location of all bounding boxes that you have drawn in a frame in the next frame. If you select a single box and then select the predict next icon, only that box will be predicted in the next frame. If you have not added any boxes to the current frame, you will receive an error. You must add at least one box to the frame before using this feature.
After you've used the predict next icon, review the location of each box in the next frame and make adjustments to the box location and size if necessary.

For all other tools, you can use the **Copy to next** and **Copy to all** tools to copy your annotations to the next or all frames respectively.

**Navigate the UI**

You can navigate between video frames using the navigation bar in the bottom-left corner of your UI.

Use the play button to automatically move through the entire sequence of frames.

Use the next frame and previous frame buttons to move forward or back one frame at a time. You can also input a frame number to navigate to that frame.

You can zoom in to and out of all video frames. Once you have zoomed into a video frame, you can move around in that frame using the move icon. When you set a new view in a single video frame by zooming and moving within that frame, all video frames are set to the same view. You can reset all video frames to their original view using the fit screen icon. For additional view options, see Icons Guide (p. 217).

When you are in the worker UI, you see the following menus:

- **Instructions** – Review these instructions before starting your task. Additionally, select More instructions and review these instructions.
- **Shortcuts** – Use this menu to view keyboard shortcuts that you can use to navigate video frames and use the tools provided.
- **Help** – Use this option to refer back to this documentation.

**Tool Guide**

Your task will include one or more tools. The tool provided dictates the type of annotations you will create to identify and track objects. Use the following table to learn more about each tool provided.

<table>
<thead>
<tr>
<th>Tool</th>
<th>Icon</th>
<th>Action</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bounding box</td>
<td><img src="#" alt="Icon" /></td>
<td>Add a bounding box annotation.</td>
<td>Choose this icon to add a bounding box. Each bounding box you add is associated with the category you choose from the Label category drop down menu. Select the bounding box or its associated label to adjust it.</td>
</tr>
<tr>
<td>Bounding box</td>
<td><img src="#" alt="Icon" /></td>
<td>Predict bounding boxes in the next frame.</td>
<td>Select a bounding box, and then choose this icon to predict the location of that box in the next frame. You can select the icon multiple times in a row to automatically detect the location of box in multiple frames. For example, select</td>
</tr>
<tr>
<td>Tool</td>
<td>Icon</td>
<td>Action</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>------</td>
<td>-------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Keypoint</td>
<td><img src="image" alt="Icon" /></td>
<td>Add a keypoint annotation.</td>
<td>Choose this icon to add a keypoint. Click on an object the image to place the keypoint at that location. Each keypoint you add is associated with the category you choose from the Label category drop down menu. Select a keypoint or its associated label to adjust it.</td>
</tr>
<tr>
<td>Polyline</td>
<td><img src="image" alt="Icon" /></td>
<td>Add a polyline annotation.</td>
<td>Choose this icon to add a polyline. To add a polyline, continuously click around the object of interest to add new points. To stop drawing a polyline, select the last point that you placed a second time (this point will be green), or press Enter on your keyboard. Each point added to the polyline is connected to the previous point by a line. The polyline does not have to be closed (the start point and end point do not have to be the same) and there are no restrictions on the angles formed between lines. Each polyline you add is associated with the category you choose from the Label category drop down menu. Select the polyline or its associated label to adjust it.</td>
</tr>
<tr>
<td>Tool</td>
<td>Icon</td>
<td>Action</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>------</td>
<td>-------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| Polygon      | ![Polygon Icon](image) | Add a polygon annotation. | Choose this icon to add a polygon. To add a polygon, continuously click around the object of interest to add new points. To stop drawing the polygon, select the start point (this point will be green).  
A polygon is a closed shape defined by a series of points that you place. Each point added to the polygon is connected to the previous point by a line and there are no restrictions on the angles formed between lines. The start and end point must be the same.  
Each polyline you add is associated with the category you choose from the Label category drop down menu. Select the polyline or its associated label to adjust it. |
| Copy to Next | ![Copy to Next Icon](image) | Copy annotations to the next frame. | If one or more annotations are selected in the current frame, those annotations are copied to the next frame. If no annotations are selected, all annotations in the current frame will be copied to the next frame. |
## Copy to All

<table>
<thead>
<tr>
<th>Icon</th>
<th>Action</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Copy" /></td>
<td>Copy annotations to all subsequent frames.</td>
<td>If one or more annotations are selected in the current frame, those annotations are copied to all subsequent frames. If no annotations are selected, all annotations in the current frame will be copied to all subsequent frames.</td>
</tr>
</tbody>
</table>

## Icons Guide

Use this table to learn about the icons you see in your UI. You can automatically select some of these icons using the keyboard shortcuts found in the Shortcuts menu.

<table>
<thead>
<tr>
<th>Icon</th>
<th>Action</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Brightness" /></td>
<td>brightness</td>
<td>Choose this icon to adjust the brightness of all video frames.</td>
</tr>
<tr>
<td><img src="image" alt="Contrast" /></td>
<td>contrast</td>
<td>Choose this icon to adjust the contrast of all video frames.</td>
</tr>
<tr>
<td><img src="image" alt="Zoom In" /></td>
<td>zoom in</td>
<td>Choose this icon to zoom into all of the video frames.</td>
</tr>
<tr>
<td><img src="image" alt="Zoom Out" /></td>
<td>zoom out</td>
<td>Choose this icon to zoom out of all of the video frames.</td>
</tr>
<tr>
<td><img src="image" alt="Move Screen" /></td>
<td>move screen</td>
<td>After you've zoomed into a video frame, choose this icon to move around in that video frame. You can move around the video frame using your mouse by clicking and dragging the frame in the direction you want it to move. This will change the view in all view frames.</td>
</tr>
<tr>
<td><img src="image" alt="Fit Screen" /></td>
<td>fit screen</td>
<td>Reset all video frames to their original position.</td>
</tr>
</tbody>
</table>
## Label Videos and Video Frames

<table>
<thead>
<tr>
<th>Icon</th>
<th>Action</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Undo Icon" /></td>
<td>undo</td>
<td>Undo an action. You can use this icon to remove a bounding box that you just added, or to undo an adjustment you made to a bounding box.</td>
</tr>
<tr>
<td><img src="image" alt="Redo Icon" /></td>
<td>redo</td>
<td>Redo an action that was undone using the undo icon.</td>
</tr>
<tr>
<td><img src="image" alt="Delete Label Icon" /></td>
<td>delete label</td>
<td>Delete a label. This will delete the bounding box associated with the label in a single frame.</td>
</tr>
<tr>
<td><img src="image" alt="Eye Icon" /></td>
<td>show or hide label</td>
<td>Select this icon to show a label that has been hidden. If this icon has a slash through it, select it to hide the label.</td>
</tr>
<tr>
<td><img src="image" alt="Pencil Icon" /></td>
<td>edit label</td>
<td>Select this icon to open the <strong>Edit instance</strong> menu. Use this menu to edit a label category, ID, and to add or edit label attributes.</td>
</tr>
</tbody>
</table>

### Shortcuts

The keyboard shortcuts listed in the **Shortcuts** menu can help you quickly select icons, undo and redo annotations, and use tools to add and edit annotations. For example, once you add a bounding box, you can use P to quickly predict the location of that box in subsequent frames.

Before you start your task, it is recommended that you review the **Shortcuts** menu and become acquainted with these commands.

### Saving Your Work and Submitting

You should periodically save your work using the **Save** button. Ground Truth will automatically save your work every 15 minutes.

When you open a task, you must complete your work on it before pressing **Submit**. If you select **Stop Working** you will lose that task, and other workers will be able to start working on it.

### Work on Video Frame Object Detection Tasks

Video frame object detection tasks require you to classify and identify the location of objects in video frames using annotations. A video frame is a still image from a video scene.

You can use the worker UI to navigate between video frames and create annotations to identify objects of interest. Use the sections on this page to learn how to navigate your worker UI, use the tools provided, and complete your task.

It is recommended that you complete your task using a Google Chrome web browser.
**Important**
If you see annotations have already been added to one or more video frames when you open your task, adjust those annotations and add additional annotations as needed.

**Topics**
- Your Task (p. 219)
- Navigate the UI (p. 221)
- Tool Guide (p. 221)
- UI Icon Guide (p. 224)
- Shortcuts (p. 225)
- Saving Your Work and Submitting (p. 225)

**Your Task**

When you work on a video frame object detection task, you need to select a category from the **Label category** menu on the right side of your worker portal to start annotating. After you've chosen a category, draw annotations around objects that this category applies to. To learn more about the tools you see in your worker UI, refer to the Tool Guide (p. 221).

After you've added a label, you may see a downward pointing arrow next to the label in the **Labels** menu. Select this arrow and then select one option for each label attribute you see to provide more information about that label.

You may see frame attributes under the **Labels** menu. These attributes will appear on each frame in your task. Use these attribute prompts to enter additional information about each frame.
To edit an annotation, select the label of the annotation that you want to edit in the **Labels** menu or select the annotation in the frame. When you edit or delete an annotation, the action will only modify the annotation in a single frame.

If you are working on a task that includes a bounding box tool, use the predict next icon to predict the location of all bounding boxes that you have drawn in a frame in the next frame. If you select a single box and then select the predict next icon, only that box will be predicted in the next frame. If you have not added any boxes to the current frame, you will receive an error. You must add at least one box to the frame before using this feature.

**Note**

The predict next feature will not overwrite manually created annotations. It will only add annotations. If you use predict next and as a result have more than one bounding box around a single object, delete all but one box. Each object should only be identified with a single box.
After you've used the predict next icon, review the location of each box in the next frame and make adjustments to the box location and size if necessary.

For all other tools, you can use the **Copy to next** and **Copy to all** tools to copy your annotations to the next or all frames respectively.

**Navigate the UI**

You can navigate between video frames using the navigation bar in the bottom-left corner of your UI.

Use the play button to automatically play through multiple frames.

Use the next frame and previous frame buttons to move forward or back one frame at a time. You can also input a frame number to navigate to that frame.

You can zoom in to and out of all video frames. Once you have zoomed into a video frame, you can move around in that frame using the move icon. When you navigate to a new view in a single video frame by zooming and moving within that frame, all video frames are set to the same view. You can reset all video frames to their original view using the fit screen icon. To learn more, see UI Icon Guide (p. 224).

When you are in the worker UI, you see the following menus:

- **Instructions** – Review these instructions before starting your task. Additionally, select More instructions and review these instructions.
- **Shortcuts** – Use this menu to view keyboard shortcuts that you can use to navigate video frames and use the annotation tools provided.
- **Help** – Use this option to refer back to this documentation.

**If you**

**Tool Guide**

Your task will include one or more tools. The tool provided dictates the type of annotations you will create to identify and label objects. Use the following table to learn more about the tool or tools you may see in your worker UI.

<table>
<thead>
<tr>
<th>Tool</th>
<th>Icon</th>
<th>Action</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bounding box</td>
<td><img src="image" alt="Icon" /></td>
<td>Add a bounding box annotation.</td>
<td>Choose this icon to add a bounding box. Each bounding box you add is associated with the category you choose from the Label category drop down menu. Select the bounding box or its associated label to adjust it.</td>
</tr>
<tr>
<td>Predict next</td>
<td><img src="image" alt="Icon" /></td>
<td>Predict bounding boxes in the next frame.</td>
<td>Select a bounding box, and then choose this icon to predict the location of that box in the next frame. You can select the icon multiple times in a row to automatically detect the location of...</td>
</tr>
<tr>
<td>Tool</td>
<td>Icon</td>
<td>Action</td>
<td>Description</td>
</tr>
<tr>
<td>---------------</td>
<td>------</td>
<td>--------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>box in multiple frames. For example, select this icon 5 times to predict the location of a bounding box in the next 5 frames.</td>
</tr>
<tr>
<td>Keypoint</td>
<td><img src="image" alt="Keypoint Icon" /></td>
<td>Add a keypoint annotation.</td>
<td>Choose this icon to add a keypoint. Click on an object the image to place the keypoint at that location.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Each keypoint you add is associated with the category you choose from the Label category drop down menu. Select a keypoint or its associated label to adjust it.</td>
</tr>
<tr>
<td>Polyline</td>
<td><img src="image" alt="Polyline Icon" /></td>
<td>Add a polyline annotation.</td>
<td>Choose this icon to add a polyline. To add a polyline, continuously click around the object of interest to add new points. To stop drawing a polyline, select the last point that you placed a second time (this point will be green), or press <strong>Enter</strong> on your keyboard.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Each point added to the polyline is connected to the previous point by a line. The polyline does not have to be closed (the start point and end point do not have to be the same) and there are no restrictions on the angles formed between lines.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Each polyline you add is associated with the category you choose from the Label category drop down menu. Select the polyline or its associated label to adjust it.</td>
</tr>
<tr>
<td>Tool</td>
<td>Icon</td>
<td>Action</td>
<td>Description</td>
</tr>
<tr>
<td>---------------</td>
<td>------</td>
<td>-------------------------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Polygon</td>
<td><img src="image" alt="Polygon Icon" /></td>
<td>Add a polygon annotation.</td>
<td>Choose this icon to add a polygon. To add a polygon, continuously click around the object of interest to add new points. To stop drawing the polygon, select the start point (this point will be green). A polygon is a closed shape defined by a series of points that you place. Each point added to the polygon is connected to the previous point by a line and there are no restrictions on the angles formed between lines. Two lines (sides) of the polygon cannot cross. A line will become red if it violates this condition. The start and end point must be the same. Each polyline you add is associated with the category you choose from the Label category drop down menu. Select the poly</td>
</tr>
<tr>
<td>Copy to Next</td>
<td><img src="image" alt="Copy to Next Icon" /></td>
<td>Copy annotations to the next frame.</td>
<td>If one or more annotations are selected in the current frame, those annotations are copied to the next frame. If no annotations are selected, all annotations in the current frame will be copied to the next frame.</td>
</tr>
</tbody>
</table>
### Tool

<table>
<thead>
<tr>
<th>Tool</th>
<th>Icon</th>
<th>Action</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Copy to All</td>
<td><img src="image" alt="Copy Icon" /></td>
<td>Copy annotations to all subsequent frames.</td>
<td>If one or more annotations are selected in the current frame, those annotations are copied to all subsequent frames. If no annotations are selected, all annotations in the current frame will be copied to all subsequent frames.</td>
</tr>
</tbody>
</table>

### UI Icon Guide

Use this table to learn about the icons you see in your worker task portal. You can automatically select these icons using the keyboard shortcuts found in the **Shortcuts** menu.

<table>
<thead>
<tr>
<th>Icon</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Brightness Icon" /></td>
<td>Choose this icon to adjust the brightness of all video frames.</td>
</tr>
<tr>
<td><img src="image" alt="Contrast Icon" /></td>
<td>Choose this icon to adjust the contrast of all video frames.</td>
</tr>
<tr>
<td><img src="image" alt="Zoom In Icon" /></td>
<td>Choose this icon to zoom into all of the video frames.</td>
</tr>
<tr>
<td><img src="image" alt="Zoom Out Icon" /></td>
<td>Choose this icon to zoom out of all of the video frames.</td>
</tr>
<tr>
<td><img src="image" alt="Move Screen Icon" /></td>
<td>After you've zoomed into a video frame, choose this icon to move around in that video frame. You can move around in the video frame using your mouse by clicking and dragging the frame in the direction you want it to move. This will change the view in all view frames.</td>
</tr>
<tr>
<td><img src="image" alt="Fit Screen Icon" /></td>
<td>Reset all video frames to their original position.</td>
</tr>
</tbody>
</table>
Undo an action. You can use this icon to remove a bounding box that you just added, or to undo an adjustment you made to a bounding box.

Redo an action that was undone using the undo icon.

Delete a label. This will delete the bounding box associated with the label in a single frame.

Select this icon to show a label that has been hidden. If this icon has a slash through it, select it to hide the label.

The keyboard shortcuts listed in the **Shortcuts** menu can help you quickly select icons, undo and redo annotations, and use tools to add and edit annotations. For example, once you add a bounding box, you can use `P` to quickly predict the location of that box in subsequent frames.

Before you start your task, it is recommended that you review the **Shortcuts** menu and become acquainted with these commands.

Saving Your Work and Submitting

You should periodically save your work. Ground Truth automatically saves your work every 15 minutes.

When you open a task, you must complete your work before pressing **Submit**. If you select **Stop Working**, you lose that task, and other workers can start working on it.

Use Ground Truth to Label 3D Point Clouds

Create a 3D point cloud labeling job to have workers label objects in 3D point clouds generated from 3D sensors like Light Detection and Ranging (LiDAR) sensors and depth cameras, or generated from 3D reconstruction by stitching images captured by an agent like a drone.

3D Point Clouds

Point clouds are made up of three-dimensional (3D) visual data that consists of points. Each point is described using three coordinates, typically $x$, $y$, and $z$. To add color or variations in point intensity to the point cloud, points may be described with additional attributes, such as $i$ for intensity or values for the red ($r$), green ($g$), and blue ($b$) 8-bit color channels. When you create a Ground Truth 3D point cloud labeling job, you can provide point cloud data and, optionally, sensor fusion data.

The following image shows a single, 3D point cloud scene rendered by Ground Truth and displayed in the semantic segmentation worker UI.
LiDAR

A Light Detection and Ranging (LiDAR) sensor is a common type of sensor used to collect measurements that are used to generate point cloud data. LiDAR is a remote sensing method that uses light in the form of a pulsed laser to measure the distances of objects from the sensor. You can provide 3D point cloud data generated from a LiDAR sensor for a Ground Truth 3D point cloud labeling job using the raw data formats described in Accepted Raw 3D Data Formats (p. 352).

Sensor Fusion

Ground Truth 3D point cloud labeling jobs include a sensor fusion feature that supports video camera sensor fusion for all task types. Some sensors come with multiple LiDAR devices and video cameras that capture images and associate them with a LiDAR frame. To help annotators visually complete your tasks with high confidence, you can use the Ground Truth sensor fusion feature to project annotations (labels) from a 3D point cloud to 2D camera images and vice versa using 3D scanner (such as LiDAR) extrinsic matrix and camera extrinsic and intrinsic matrices. To learn more, see Sensor Fusion (p. 368).

Label 3D Point Clouds

Ground Truth provides a user interface (UI) and tools that workers use to label or annotate 3D point clouds. When you use the object detection or semantic segmentation task types, workers can annotate a single point cloud frame. When you use object tracking, workers annotate a sequence of frames. You can use object tracking to track object movement across all frames in a sequence.

The following demonstrates how a worker would use the Ground Truth worker portal and tools to annotate a 3D point cloud for an object detection task. For similar visual examples of other task types, see 3D Point Cloud Task types (p. 229).
Assistive Labeling Tools for Point Cloud Annotation

Ground Truth offers assistive labeling tools to help workers complete your point cloud annotation tasks faster and with more accuracy. For details about assistive labeling tools that are included in the worker UI for each task type, select a task type (p. 229) and refer to the View the Worker Task Interface section of that page.

Next Steps

You can create six types of tasks when you use Ground Truth 3D point cloud labeling jobs. Use the topics in 3D Point Cloud Task types (p. 229) to learn more about these task types and to learn how to create a labeling job using the task type of your choice.

The 3D point cloud labeling job is different from other Ground Truth labeling modalities. Before creating a labeling job, we recommend that you read 3D Point Cloud Labeling Jobs Overview (p. 254).

For an end-to-end demo using the SageMaker API and AWS Python SDK (boto 3) to create a 3D point cloud labeling job, see create-3D-pointcloud-labeling-job.ipynb in the SageMaker Examples notebook tab.

Important
If you use a notebook instance created before June 5th, 2020 to run this notebook, you must stop and restart that notebook instance for the notebook to work.

Topics
- 3D Point Cloud Task types (p. 229)
- 3D Point Cloud Labeling Jobs Overview (p. 254)
- Worker Instructions (p. 257)

3D Point Cloud Task types

You can use Ground Truth 3D point cloud labeling modality for a variety of use cases. The following list briefly describes each 3D point cloud task type. For additional details and instructions on how to create a labeling job using a specific task type, select the task type name to see its task type page.

- 3D point cloud object detection – Use this task type when you want workers to locate and classify objects in a 3D point cloud by adding and fitting 3D cuboids around objects.
- 3D point cloud object tracking – Use this task type when you want workers to add and fit 3D cuboids around objects to track their movement across a sequence of 3D point cloud frames. For example, you can use this task type to ask workers to track the movement of vehicles across multiple point cloud frames.
- 3D point cloud semantic segmentation – Use this task type when you want workers to create a point-level semantic segmentation mask by painting objects in a 3D point cloud using different colors where each color is assigned to one of the classes you specify.
- 3D point cloud adjustment task types – Each of the task types above has an associated adjustment task type that you can use to audit and adjust annotations generated from a 3D point cloud labeling job. Refer to the task type page of the associated type to learn how to create an adjustment labeling job for that task.

3D Point Cloud Object Detection

Use this task type when you want workers to classify objects in a 3D point cloud by drawing 3D cuboids around objects. For example, you can use this task type to ask workers to identify different types of objects in a point cloud, such as cars, bikes, and pedestrians.
For this task type, the data object that workers label is a single point cloud frame. Ground Truth renders a 3D point cloud using point cloud data you provide. You can also provide camera data to give workers more visual information about scenes in the frame, and to help workers draw 3D cuboids around objects.

Ground Truth provides workers with tools to annotate objects with 9 degrees of freedom \((x,y,z,rx,ry,rz,l,w,h)\) in three dimensions in both 3D scene and projected side views (top, side, and back). If you provide sensor fusion information (like camera data), when a worker adds a cuboid to identify an object in the 3D point cloud, the cuboid shows up and can be modified in the 2D images. After a cuboid has been added, all edits made to that cuboid in the 2D or 3D scene are projected into the other view.

You can create a job to adjust annotations created in a 3D point cloud object detection labeling job using the 3D point cloud object detection adjustment task type.

If you are a new user of the Ground Truth 3D point cloud labeling modality, we recommend you review 3D Point Cloud Labeling Jobs Overview (p. 254). This labeling modality is different from other Ground Truth task types, and this page provides an overview of important details you should be aware of when creating a 3D point cloud labeling job.

Topics

- View the Worker Task Interface (p. 230)
- Create a 3D Point Cloud Object Detection Labeling Job (p. 234)
- Create a 3D Point Cloud Object Detection Adjustment or Verification Labeling Job (p. 236)
- Output Data Format (p. 236)

View the Worker Task Interface

Ground Truth provides workers with a web portal and tools to complete your 3D point cloud object detection annotation tasks. When you create the labeling job, you provide the Amazon Resource Name (ARN) for a pre-built Ground Truth worker UI in the HumanTaskUiArn parameter. When you create a labeling job using this task type in the console, this worker UI is automatically used. You can preview and interact with the worker UI when you create a labeling job in the console. If you are a new user, it is recommended that you create a labeling job using the console to ensure your label attributes, point cloud frames, and if applicable, images, appear as expected.

The following is a GIF of the 3D point cloud object detection worker task interface. If you provide camera data for sensor fusion in the world coordinate system, images are matched up with scenes in the point cloud frame. These images appear in the worker portal as shown in the following GIF.
Worker can navigate in the 3D scene using their keyboard and mouse. They can:

- Double click on specific objects in the point cloud to zoom into them.
- Use a mouse-scroller or trackpad to zoom in and out of the point cloud.
- Use both keyboard arrow keys and Q, E, A, and D keys to move Up, Down, Left, Right. Use keyboard keys W and S to zoom in and out.

Once a worker places a cuboid in the 3D scene, a side-view will appear with the three projected side views: top, side, and back. These side-views show points in and around the placed cuboid and help workers refine cuboid boundaries in that area. Workers can zoom in and out of each of those side-views using their mouse.

The following video demonstrates movements around the 3D point cloud and in the side-view.
<table>
<thead>
<tr>
<th>Instructions</th>
<th>Shortcuts</th>
<th>Label</th>
<th>View</th>
<th>3D Point Cloud</th>
<th>Help</th>
</tr>
</thead>
</table>

Point cloud View X

![Point cloud View](image)
Assistive Labeling Tools

Ground Truth helps workers annotate 3D point clouds faster and more accurately using machine learning and computer vision powered assistive labeling tools for 3D point cloud object tracking tasks. The following assistive labeling tools are available for this task type:

- **Snapping** – Workers can add a cuboid around an object and use a keyboard shortcut or menu option to have Ground Truth's autofit tool snap the cuboid tightly around the object.
- **Set to ground** – After a worker adds a cuboid to the 3D scene, the worker can automatically snap the cuboid to the ground. For example, the worker can use this feature to snap a cuboid to the road or sidewalk in the scene.
- **Multi-view labeling** – After a worker adds a 3D cuboid to the 3D scene, a side panel displays front, side, and top perspectives to help the worker adjust the cuboid tightly around the object. In all of these views, the cuboid includes an arrow that indicates the orientation, or heading of the object. When the worker adjusts the cuboid, the adjustment will appear in real time on all of the views (that is, 3D, top, side, and front).
- **Sensor fusion** – If you provide data for sensor fusion, workers can adjust annotations in the 3D scenes and in 2D images, and the annotations will be projected into the other view in real time. Additionally, workers will have the option to view the direction the camera is facing and the camera frustum.
- **View options** – Enables workers to easily hide or view cuboids, label text, a ground mesh, and additional point attributes like color or intensity. Workers can also choose between perspective and orthogonal projections.

Create a 3D Point Cloud Object Detection Labeling Job

You can create a 3D point cloud labeling job using the SageMaker console or API operation, `CreateLabelingJob`. To create a labeling job for this task type you need the following:

- A single-frame input manifest file. To learn how to create this type of manifest file, see Create a Point Cloud Frame Input Manifest File (p. 354). If you are a new user of Ground Truth 3D point cloud labeling modalities, you may also want to review Accepted Raw 3D Data Formats (p. 352).
- A work team from a private or vendor workforce. You cannot use Amazon Mechanical Turk for video frame labeling jobs. To learn how to create workforces and work teams, see Create and Manage Workforces (p. 427).
- A label category configuration file. For more information, see Create a Labeling Category Configuration File with Label Category and Frame Attributes (p. 326).

Additionally, make sure that you have reviewed and satisfied the Assign IAM Permissions to Use Ground Truth (p. 416).

Use one of the following sections to learn how to create a labeling job using the console or an API.

Create a Labeling Job (Console)

You can follow the instructions Create a Labeling Job (Console) (p. 314) in order to learn how to create a 3D point cloud object detection labeling job in the SageMaker console. While you are creating your labeling job, be aware of the following:

- Your input manifest file must be a single-frame manifest file. For more information, see Create a Point Cloud Frame Input Manifest File (p. 354).
- Optionally, you can provide label category attributes. Workers can assign one or more of these attributes to annotations to provide more information about that object. For example, you might want to use the attribute `occluded` to have workers identify when an object is partially obstructed.
• Automated data labeling and annotation consolidation are not supported for 3D point cloud labeling tasks.

• 3D point cloud object detection labeling jobs can take multiple hours to complete. You can specify a longer time limit for these labeling jobs when you select your work team (up to 7 days, or 604800 seconds).

  **Important**  
  If you set your take time limit to be greater than 8 hours, you must set MaxSessionDuration for your IAM execution role to at least 8 hours. To see how to update this value for your IAM role, see Modifying a Role in the IAM User Guide, choose your preferred method to modify the role, and then follow the steps in Modifying a Role Maximum Session Duration.

Create a Labeling Job (API)

This section covers details you need to know when you create a labeling job using the SageMaker API operation CreateLabelingJob. This API defines this operation for all AWS SDKs. To see a list of language-specific SDKs supported for this operation, review the **See Also** section of CreateLabelingJob.

Create a Labeling Job (API) (p. 317), provides an overview of the CreateLabelingJob operation. Follow these instructions and do the following while you configure your request:

• You must enter an ARN for HumanTaskUiArn. Use arn:aws:sagemaker:<region>:394669845002:human-task-ui/PointCloudObjectDetection. Replace <region> with the AWS Region you are creating the labeling job in. There should not be an entry for the UiTemplateS3Uri parameter.

• Your input manifest file must be a single-frame manifest file. For more information, see Create a Point Cloud Frame Input Manifest File (p. 354).

• You specify your labels, label category and frame attributes, and worker instructions in a label category configuration file. To learn how to create this file, see Create a Labeling Category Configuration File with Label Category and Frame Attributes (p. 326).

• You need to provide pre-defined ARNs for the pre-annotation and post-annotation (ACS) Lambda functions. These ARNs are specific to the AWS Region you use to create your labeling job.

  • To find the pre-annotation Lambda ARN, refer to PreHumanTaskLambdaArn. Use the Region you are creating your labeling job in to find the correct ARN. For example, if you are creating your labeling job in us-east-1, the ARN will be arn:aws:lambda:us-east-1:432418664414:function:PRE-3DPointCloudObjectDetection.

  • To find the post-annotation Lambda ARN, refer to AnnotationConsolidationLambdaArn. Use the Region you are creating your labeling job in to find the correct ARN. For example, if you are creating your labeling job in us-east-1, the ARN will be arn:aws:lambda:us-east-1:432418664414:function:ACS-3DPointCloudObjectDetection.

• The number of workers specified in NumberOfHumanWorkersPerDataObject must be 1.

• Automated data labeling is not supported for 3D point cloud labeling jobs. You should not specify values for parameters in LabelingJobAlgorithmsConfig.

• 3D point cloud object detection labeling jobs can take multiple hours to complete. You can specify a longer time limit for these labeling jobs in TaskTimeLimitInSeconds (up to 7 days, or 604,800 seconds).

  **Important**  
  If you set your take time limit to be greater than 8 hours, you must set MaxSessionDuration for your IAM execution role to at least 8 hours. To see how to update this value for your IAM role, see Modifying a Role in the IAM User Guide, choose your preferred method to modify the role, and then follow the steps in Modifying a Role Maximum Session Duration.
preferred method to modify the role, and then follow the steps in Modifying a Role Maximum Session Duration.

Create a 3D Point Cloud Object Detection Adjustment or Verification Labeling Job

You can create an adjustment labeling job using the Ground Truth console or CreateLabelingJob API. You must use the Ground Truth API to create a 3D point cloud object detection verification labeling job. To learn more about adjustment and verification labeling jobs, and to learn how create one, see Verify and Adjust Labels (p. 287).

When you create an adjustment labeling job, your input data to the labeling job can include labels, and yaw, pitch, and roll measurements from a previous labeling job or external source. In the adjustment job, pitch, and roll will be visualized in the worker UI, but cannot be modified. Yaw is adjustable.

Ground Truth uses Tait-Bryan angles with the following intrinsic rotations to visualize yaw, pitch and roll in the worker UI. First, rotation is applied to the vehicle according to the z-axis (yaw). Next, the rotated vehicle is rotated according to the intrinsic y’-axis (pitch). Finally, the vehicle is rotated according to the intrinsic x’’-axis (roll).

Output Data Format

When you create a 3D point cloud object detection labeling job, tasks are sent to workers. When these workers complete their tasks, labels are written to the Amazon S3 bucket you specified when you created the labeling job. The output data format determines what you see in your Amazon S3 bucket when your labeling job status (LabelingJobStatus) is Completed.

If you are a new user of Ground Truth, see Output Data (p. 381) to learn more about the Ground Truth output data format. To learn about the 3D point cloud object detection output data format, see 3D Point Cloud Object Detection Output (p. 398).

3D Point Cloud Object Tracking

Use this task type when you want workers to add and fit 3D cuboids around objects to track their movement across 3D point cloud frames. For example, you can use this task type to ask workers to track the movement of vehicles across multiple point cloud frames.

For this task type, the data object that workers label is a sequence of point cloud frames. A sequence is defined as a temporal series of point cloud frames. Ground Truth renders a series of 3D point cloud visualizations using a sequence you provide and workers can switch between these 3D point cloud frames in the worker task interface.

Ground Truth providers workers with tools to annotate objects with 9 degrees of freedom: \((x,y,z,rx,ry,rz,l,w,h)\) in three dimensions in both 3D scene and projected side views (top, side, and back). When a worker draws a cuboid around an object, that cuboid is given a unique ID, for example Car:1 for one car in the sequence and Car:2 for another. Workers use that ID to label the same object in multiple frames.

You can also provide camera data to give workers more visual information about scenes in the frame, and to help workers draw 3D cuboids around objects. When a worker adds a 3D cuboid to identify an object in either the 2D image or the 3D point cloud, and the cuboid shows up in the other view.

You can adjust annotations created in a 3D point cloud object detection labeling job using the 3D point cloud object tracking adjustment task type.

If you are a new user of the Ground Truth 3D point cloud labeling modality, we recommend you review 3D Point Cloud Labeling Jobs Overview (p. 254). This labeling modality is different from other Ground
Truth task types, and this page provides an overview of important details you should be aware of when creating a 3D point cloud labeling job.

Topics
- View the Worker Task Interface (p. 237)
- Create a 3D Point Cloud Object Tracking Labeling Job (p. 244)
- Create a 3D Point Cloud Object Tracking Adjustment or Verification Labeling Job (p. 245)
- Output Data Format (p. 245)

View the Worker Task Interface

Ground Truth provides workers with a web portal and tools to complete your 3D point cloud object tracking annotation tasks. When you create the labeling job, you provide the Amazon Resource Name (ARN) for a pre-built Ground Truth UI in the `HumanTaskUiArn` parameter. When you create a labeling job using this task type in the console, this UI is automatically used. You can preview and interact with the worker UI when you create a labeling job in the console. If you are a new user, it is recommended that you create a labeling job using the console to ensure your label attributes, point cloud frames, and if applicable, images, appear as expected.

The following is a GIF of the 3D point cloud object tracking worker task interface and demonstrates how the worker can navigate the point cloud frames in the sequence.
Once workers add a single cuboid, that cuboid is replicated in all frames of the sequence with the same ID. Once workers adjust the cuboid in another frame, Ground Truth will interpolate the movement of that object and adjust all cuboids between the manually adjusted frames. The following GIF demonstrates this interpolation feature. In the navigation bar on the bottom-left, red-areas indicate manually adjusted frames.
If you provide camera data for sensor fusion, images are matched up with scenes in point cloud frames. These images appear in the worker portal as shown in the following GIF.

Worker can navigate in the 3D scene using their keyboard and mouse. They can:

- Double click on specific objects in the point cloud to zoom into them.
- Use a mouse-scroller or trackpad to zoom in and out of the point cloud.
- Use both keyboard arrow keys and Q, E, A, and D keys to move Up, Down, Left, Right. Use keyboard keys W and S to zoom in and out.

Once a worker places a cuboids in the 3D scene, a side-view will appear with the three projected side views: top, side, and back. These side-views show points in and around the placed cuboid and help workers refine cuboid boundaries in that area. Workers can zoom in and out of each of those side-views using their mouse.

The following video demonstrates movements around the 3D point cloud and in the side-view.
Additional view options and features are available. See the worker instruction page for a comprehensive overview of the Worker UI.

**Worker Tools**

Workers can navigate through the 3D point cloud by zooming in and out, and moving in all directions around the cloud using the mouse and keyboard shortcuts. If workers click on a point in the point cloud, the UI will automatically zoom into that area. Workers can use various tools to draw 3D cuboid around objects. For more information, see **Assistive Labeling Tools**.

After workers have placed a 3D cuboid in the point cloud, they can adjust these cuboids to fit tightly around cars using a variety of views: directly in the 3D cuboid, in a side-view featuring three zoomed-in perspectives of the point cloud around the box, and if you include images for sensor fusion, directly in the 2D image.

View options that enable workers to easily hide or view label text, a ground mesh, and additional point attributes. Workers can also choose between perspective and orthogonal projections.

**Assistive Labeling Tools**

Ground Truth helps workers annotate 3D point clouds faster and more accurately using UX, machine learning and computer vision powered assistive labeling tools for 3D point cloud object tracking tasks. The following assistive labeling tools are available for this task type:

- **Label autofill** – When a worker adds a cuboid to a frame, a cuboid with the same dimensions and orientation is automatically added to all frames in the sequence.

- **Label interpolation** – After a worker has labeled a single object in two frames, Ground Truth uses those annotations to interpolate the movement of that object between those two frames.

- **Bulk label management** – Workers can add, delete, and rename annotations in bulk.
  - Workers can manually delete annotations for a given object before or after a frame. For example, a worker can delete all labels for an object after frame 10 if that object is no longer located in the scene after that frame.
  - If a worker accidentally bulk deletes all annotations for an object, they can add them back. For example, if a worker deletes all annotations for an object before frame 100, they can bulk add them to those frames.
  - Workers can rename a label in one frame and all 3D cuboids assigned that label are updated with the new name across all frames.

- **Snapping** – Workers can add a cuboid around an object and use a keyboard shortcut or menu option to have Ground Truth's autofit tool snap the cuboid tightly around the object's boundaries.

- **Fit to ground** – After a worker adds a cuboid to the 3D scene, the worker can automatically snap the cuboid to the ground. For example, the worker can use this feature to snap a cuboid to the road or sidewalk in the scene.

- **Multi-view labeling** – After a worker adds a 3D cuboid to the 3D scene, a side-panel displays front and two side perspectives to help the worker adjust the cuboid tightly around the object. Workers can annotation the 3D point cloud, the side panel and the adjustments appear in the other views in real time.

- **Sensor fusion** – If you provide data for sensor fusion, workers can adjust annotations in the 3D scenes and in 2D images, and the annotations will be projected into the other view in real time.

- **Auto-merge cuboids** – Workers can automatically merge two cuboids across all frames if they determine that cuboids with different labels actually represent a single object.

- **View options** – Enables workers to easily hide or view label text, a ground mesh, and additional point attributes like color or intensity. Workers can also choose between perspective and orthogonal projections.
Create a 3D Point Cloud Object Tracking Labeling Job

You can create a 3D point cloud labeling job using the SageMaker console or API operation, `CreateLabelingJob`. To create a labeling job for this task type you need the following:

- A sequence input manifest file. To learn how to create this type of manifest file, see Create a Point Cloud Sequence Input Manifest (p. 360). If you are a new user of Ground Truth 3D point cloud labeling modalities, we recommend that you review Accepted Raw 3D Data Formats (p. 352).
- A work team from a private or vendor workforce. You cannot use Amazon Mechanical Turk for 3D point cloud labeling jobs. To learn how to create workforces and work teams, see Create and Manage Workforces (p. 427).
- A label category configuration file. For more information, see Create a Labeling Category Configuration File with Label Category and Frame Attributes (p. 326).

Additionally, make sure that you have reviewed and satisfied the Assign IAM Permissions to Use Ground Truth (p. 416).

To learn how to create a labeling job using the console or an API, see the following sections.

Create a Labeling Job (API)

This section covers details you need to know when you create a labeling job using the SageMaker API operation `CreateLabelingJob`. This API defines this operation for all AWS SDKs. To see a list of language-specific SDKs supported for this operation, review the See Also section of `CreateLabelingJob`.

`Create a Labeling Job (API)` (p. 317) provides an overview of the `CreateLabelingJob` operation. Follow these instructions and do the following while you configure your request:

- You must enter an ARN for `HumanTaskUiArn`. Use `arn:aws:sagemaker:<region>:394669845002:human-task-ui/PointCloudObjectTracking`. Replace `<region>` with the AWS Region you are creating the labeling job in.
  
  There should not be an entry for the `UiTemplateS3Uri` parameter.
- Your `LabelAttributeName` must end in `-ref`. For example, `ot-labels-ref`.
- Your input manifest file must be a point cloud frame sequence manifest file. For more information, see Create a Point Cloud Sequence Input Manifest (p. 360).
- You specify your labels, label category and frame attributes, and worker instructions in a label category configuration file. For more information, see Create a Labeling Category Configuration File with Label Category and Frame Attributes (p. 326) to learn how to create this file.
- You need to provide pre-defined ARNs for the pre-annotation and post-annotation (ACS) Lambda functions. These ARNs are specific to the AWS Region you use to create your labeling job.
  
  To find the pre-annotation Lambda ARN, refer to `PreHumanTaskLambdaArn`. Use the Region you are creating your labeling job in to find the correct ARN that ends with `PRE-3DPointCloudObjectTracking`.
  
  To find the post-annotation Lambda ARN, refer to `AnnotationConsolidationLambdaArn`. Use the Region you are creating your labeling job in to find the correct ARN that ends with `ACS-3DPointCloudObjectTracking`.
- The number of workers specified in `NumberOfHumanWorkersPerDataObject` should be 1.
- Automated data labeling is not supported for 3D point cloud labeling jobs. You should not specify values for parameters in `LabelingJobAlgorithmsConfig`.
- 3D point cloud object tracking labeling jobs can take multiple hours to complete. You can specify a longer time limit for these labeling jobs in `TaskTimeLimitInSeconds` (up to 7 days, or 604,800 seconds).
Important
If you set your take time limit to be greater than 8 hours, you must set `MaxSessionDuration` for your IAM execution role to at least 8 hours. To see how to update this value for your IAM role, see Modifying a Role in the IAM User Guide, choose your preferred method to modify the role, and then follow the steps in Modifying a Role Maximum Session Duration.

Create a Labeling Job (Console)
You can follow the instructions Create a Labeling Job (Console) (p. 314) in order to learn how to create a 3D point cloud object tracking labeling job in the SageMaker console. While you are creating your labeling job, be aware of the following:

- Your input manifest file must be a sequence manifest file. For more information, see Create a Point Cloud Sequence Input Manifest (p. 360).
- Optionally, you can provide label category attributes. Workers can assign one or more of these attributes to annotations to provide more information about that object. For example, you might want to use the attribute `occluded` to have workers identify when an object is partially obstructed.
- Automated data labeling and annotation consolidation are not supported for 3D point cloud labeling tasks.
- 3D point cloud object tracking labeling jobs can take multiple hours to complete. You can specify a longer time limit for these labeling jobs when you select your work team (up to 7 days, or 604800 seconds).

Important
If you set your take time limit to be greater than 8 hours, you must set `MaxSessionDuration` for your IAM execution role to at least 8 hours. To see how to update this value for your IAM role, see Modifying a Role in the IAM User Guide, choose your preferred method to modify the role, and then follow the steps in Modifying a Role Maximum Session Duration.

Create a 3D Point Cloud Object Tracking Adjustment or Verification Labeling Job
You can create an adjustment labeling job using the Ground Truth console or `CreateLabelingJob` API. You must use the Ground Truth API to create a 3D point cloud object tracking verification labeling job. To learn more about adjustment and verification labeling jobs, and to learn how create one, see Verify and Adjust Labels (p. 287).

When you create an adjustment labeling job, your input data to the labeling job can include labels, and yaw, pitch, and roll measurements from a previous labeling job or external source. In the adjustment job, pitch, and roll will be visualized in the worker UI, but cannot be modified. Yaw is adjustable.

Ground Truth uses Tait-Bryan angles with the following intrinsic rotations to visualize yaw, pitch and roll in the worker UI. First, rotation is applied to the vehicle according to the z-axis (yaw). Next, the rotated vehicle is rotated according to the intrinsic y’-axis (pitch). Finally, the vehicle is rotated according to the intrinsic x”-axis (roll).

Output Data Format
When you create a 3D point cloud object tracking labeling job, tasks are sent to workers. When these workers complete their tasks, their annotations are written to the Amazon S3 bucket you specified when you created the labeling job. The output data format determines what you see in your Amazon S3 bucket when your labeling job status (`LabelingJobStatus`) is Completed.

If you are a new user of Ground Truth, see Output Data (p. 381) to learn more about the Ground Truth output data format. To learn about the 3D point cloud object tracking output data format, see 3D Point Cloud Object Tracking Output (p. 401).
3D Point Cloud Semantic Segmentation

Semantic segmentation involves classifying individual points of a 3D point cloud into pre-specified categories. Use this task type when you want workers to create a point-level semantic segmentation mask for 3D point clouds. For example, if you specify the classes car, pedestrian, and bike, workers select one class at a time, and color all of the points that this class applies to the same color in the point cloud.

For this task type, the data object that workers label is a single point cloud frame. Ground Truth generates a 3D point cloud visualization using point cloud data you provide. You can also provide camera data to give workers more visual information about scenes in the frame, and to help workers paint objects. When a worker paints an object in either the 2D image or the 3D point cloud, the paint shows up in the other view.

You can adjust annotations created in a 3D point cloud object detection labeling job using the 3D point cloud semantic segmentation adjustment task type.

If you are a new user of the Ground Truth 3D point cloud labeling modality, we recommend you review 3D Point Cloud Labeling Jobs Overview (p. 254). This labeling modality is different from other Ground Truth task types, and this topic provides an overview of important details you should be aware of when creating a 3D point cloud labeling job.

Topics

- View the Worker Task Interface (p. 246)
- Create a 3D Point Cloud Semantic Segmentation Labeling Job (p. 252)
- Create a 3D Point Cloud Semantic Segmentation Adjustment or Verification Labeling Job (p. 253)
- Output Data Format (p. 254)

View the Worker Task Interface

Ground Truth provides workers with a web portal and tools to complete your 3D point cloud semantic segmentation annotation tasks. When you create the labeling job, you provide the Amazon Resource Name (ARN) for a pre-built Ground Truth UI in the HumanTaskUiArn parameter. When you create a labeling job using this task type in the console, this UI is automatically used. You can preview and interact with the worker UI when you create a labeling job in the console. If you are a new use, it is recommended that you create a labeling job using the console to ensure your label attributes, point cloud frames, and if applicable, images, appear as expected.

The following is a GIF of the 3D point cloud semantic segmentation worker task interface. If you provide camera data for sensor fusion, images are matched with scenes in the point cloud frame. Workers can paint objects in either the 3D point cloud or the 2D image, and the paint appears in the corresponding location in the other medium. These images appear in the worker portal as shown in the following GIF.
Worker can navigate in the 3D scene using their keyboard and mouse. They can:

- Double click on specific objects in the point cloud to zoom into them.
- Use a mouse-scroller or trackpad to zoom in and out of the point cloud.
- Use both keyboard arrow keys and Q, E, A, and D keys to move Up, Down, Left, Right. Use keyboard keys W and S to zoom in and out.

The following video demonstrates movements around the 3D point cloud. Workers can hide and re-expand all side views and menus. In this GIF, the side-views and menus have been collapsed.
The following GIF demonstrates how a worker can label multiple objects quickly, refine painted objects using the Unpaint option and then view only points that have been painted.
Additional view options and features are available. See the worker instruction page for a comprehensive overview of the Worker UI.

**Worker Tools**

Workers can navigate through the 3D point cloud by zooming in and out, and moving in all directions around the cloud using the mouse and keyboard shortcuts. When you create a semantic segmentation job, workers have the following tools available to them:

- A paint brush to paint and unpaint objects. Workers paint objects by selecting a label category and then painting in the 3D point cloud. Workers unpaint objects by selecting the Unpaint option from the label category menu and using the paint brush to erase paint.
- A polygon tool that workers can use to select and paint an area in the point cloud.
- A background paint tool, which enables workers to paint behind objects they have already annotated without altering the original annotations. For example, workers might use this tool to paint the road after painting all of the cars on the road.
- View options that enable workers to easily hide or view label text, a ground mesh, and additional point attributes like color or intensity. Workers can also choose between perspective and orthogonal projections.

**Create a 3D Point Cloud Semantic Segmentation Labeling Job**

You can create a 3D point cloud labeling job using the SageMaker console or API operation, CreateLabelingJob. To create a labeling job for this task type you need the following:

- A single-frame input manifest file. To learn how to create this type of manifest file, see Create a Point Cloud Frame Input Manifest File (p. 354). If you are a new user of Ground Truth 3D point cloud labeling modalities, we recommend that you review Accepted Raw 3D Data Formats (p. 352).
- A work team from a private or vendor workforce. You cannot use Amazon Mechanical Turk workers for 3D point cloud labeling jobs. To learn how to create workforces and work teams, see Create and Manage Workforces (p. 427).
- A label category configuration file. For more information, see Create a Labeling Category Configuration File with Label Category and Frame Attributes (p. 326).

Additionally, make sure that you have reviewed and satisfied the Assign IAM Permissions to Use Ground Truth (p. 416).

Use one of the following sections to learn how to create a labeling job using the console or an API.

**Create a Labeling Job (Console)**

You can follow the instructions Create a Labeling Job (Console) (p. 314) in order to learn how to create a 3D point cloud semantic segmentation labeling job in the SageMaker console. While you are creating your labeling job, be aware of the following:

- Your input manifest file must be a single-frame manifest file. For more information, see Create a Point Cloud Frame Input Manifest File (p. 354).
- Automated data labeling and annotation consolidation are not supported for 3D point cloud labeling tasks.
- 3D point cloud semantic segmentation labeling jobs can take multiple hours to complete. You can specify a longer time limit for these labeling jobs when you select your work team (up to 7 days, or 604800 seconds).

**Important**

If you set your take time limit to be greater than 8 hours, you must set MaxSessionDuration for your IAM execution role to at least 8 hours. To see how to
update this value for your IAM role, see Modifying a Role in the IAM User Guide, choose your preferred method to modify the role, and then follow the steps in Modifying a Role Maximum Session Duration.

Create a Labeling Job (API)

This section covers details you need to know when you create a labeling job using the SageMaker API operation CreateLabelingJob. This API defines this operation for all AWS SDKs. To see a list of language-specific SDKs supported for this operation, review the See Also section of CreateLabelingJob.

The page, Create a Labeling Job (API) (p. 317), provides an overview of the CreateLabelingJob operation. Follow these instructions and do the following while you configure your request:

• You must enter an ARN for HumanTaskUiArn. Use
  arn:aws:sagemaker:<region>:39469845002:human-task-ui/PointCloudSemanticSegmentation. Replace <region> with the AWS Region you are creating the labeling job in.

  There should not be an entry for the UiTemplateS3Uri parameter.

• Your LabelAttributeName must end in -ref. For example, ss-labels-ref.

• Your input manifest file must be a single-frame manifest file. For more information, see Create a Point Cloud Frame Input Manifest File (p. 354).

• You specify your labels and worker instructions in a label category configuration file. See Create a Labeling Category Configuration File with Label Category and Frame Attributes (p. 326) to learn how to create this file.

• You need to provide a pre-defined ARNs for the pre-annotation and post-annotation (ACS) Lambda functions. These ARNs are specific to the AWS Region you use to create your labeling job.

  To find the pre-annotation Lambda ARN, refer to PreHumanTaskLambdaArn. Use the Region you are creating your labeling job in to find the correct ARN. For example, if you are creating your labeling job in us-east-1, the ARN will be arn:aws:lambda:us-east-1:432418664414:function:PRE-3DPointCloudSemanticSegmentation.

  To find the post-annotation Lambda ARN, refer to AnnotationConsolidationLambdaArn. Use the Region you are creating your labeling job in to find the correct ARN. For example, if you are creating your labeling job in us-east-1, the ARN will be arn:aws:lambda:us-east-1:432418664414:function:ACS-3DPointCloudSemanticSegmentation.

• The number of workers specified in NumberOfHumanWorkersPerDataObject should be 1.

• Automated data labeling is not supported for 3D point cloud labeling jobs. You should not specify values for parameters in LabelingJobAlgorithmsConfig.

• 3D point cloud semantic segmentation labeling jobs can take multiple hours to complete. You can specify a longer time limit for these labeling jobs in TaskTimeLimitInSeconds (up to 7 days, or 604800 seconds).

  Important
  If you set your take time limit to be greater than 8 hours, you must set MaxSessionDuration for your IAM execution role to at least 8 hours. To see how to update this value for your IAM role, see Modifying a Role in the IAM User Guide, choose your preferred method to modify the role, and then follow the steps in Modifying a Role Maximum Session Duration.

Create a 3D Point Cloud Semantic Segmentation Adjustment or Verification Labeling Job

You can create an adjustment labeling job using the Ground Truth console or CreateLabelingJob API. You must use the Ground Truth API to create a 3D point cloud semantic segmentation verification
labeling job. To learn more about adjustment and verification labeling jobs, and to learn how create one, see Verify and Adjust Labels (p. 287).

Output Data Format

When you create a 3D point cloud semantic segmentation labeling job, tasks are sent to workers. When these workers complete their tasks, their annotations are written to the Amazon S3 bucket you specified when you created the labeling job. The output data format determines what you see in your Amazon S3 bucket when your labeling job status (LabelingJobStatus) is Completed.

If you are a new user of Ground Truth, see Output Data (p. 381) to learn more about the Ground Truth output data format. To learn about the 3D point cloud object detection output data format, see 3D Point Cloud Semantic Segmentation Output (p. 397).

3D Point Cloud Labeling Jobs Overview

This topic provides an overview of the unique features of a Ground Truth 3D point cloud labeling job. You can use the 3D point cloud labeling jobs to have workers label objects in a 3D point cloud generated from a 3D sensors like LiDAR and depth cameras or generated from 3D reconstruction by stitching images captured by an agent like a drone.

Job Pre-processing Time

When you create a 3D point cloud labeling job, you need to provide an input manifest file (p. 351). The input manifest file can be:

- A **frame input manifest file** that has a single point cloud frame on each line.
- A **sequence input manifest file** that has a single sequence on each line. A sequence is defined as a temporal series of point cloud frames.

For both types of manifest files, job pre-processing time (that is, the time before Ground Truth starts sending tasks to your workers) depends on the total number and size of point cloud frames you provide in your input manifest file. For frame input manifest files, this is the number of lines in your manifest file. For sequence manifest files, this is the number of frames in each sequence multiplied by the total number of sequences, or lines, in your manifest file.

Additionally, the number of points per point cloud and the number of fused sensor data objects (like images) factor into job pre-processing times. On average, Ground Truth can pre-process 200 point cloud frames in approximately 5 minutes. If you create a 3D point cloud labeling job with a large number of point cloud frames, you might experience longer job pre-processing times. For example, if you create a sequence input manifest file with 4 point cloud sequences, and each sequence contains 200 point clouds, Ground Truth pre-processes 800 point clouds and so your job pre-processing time might be around 20 minutes. During this time, your labeling job status is InProgress.

While your 3D point cloud labeling job is pre-processing, you receive CloudWatch messages notifying you of the status of your job. To identify these messages, search for 3D_POINT_CLOUD_PROCESSING_STATUS in your labeling job logs.

For **frame input manifest files**, your CloudWatch logs will have a message similar to the following:

```json
{
   "labeling-job-name": "example-point-cloud-labeling-job",
   "event-name": "3D_POINT_CLOUD_PROCESSING_STATUS",
   "event-log-message": "datasetObjectId from: 0 to 10, status: IN_PROGRESS"
}
```

The event log message, datasetObjectId from: 0 to 10, status: IN_PROGRESS identifies the number of frames from your input manifest that have been processed. You receive a new message every...
time a frame has been processed. For example, after a single frame has processed, you receive another message that says datasetObjectId from: 1 to 10, status: IN_PROGRESS.

For **sequence input manifest files**, your CloudWatch logs will have a message similar to the following:

```
{
    "labeling-job-name": "example-point-cloud-labeling-job",
    "event-name": "3D_POINT_CLOUD_PROCESSING_STATUS",
    "event-log-message": "datasetObjectId: 0, status: IN_PROGRESS"
}
```

The event log message, datasetObjectId from: 0, status: IN_PROGRESS identifies the number of sequences from your input manifest that have been processed. You receive a new message every time a sequence has been processed. For example, after a single sequence has processed, you receive a message that says datasetObjectId from: 1, status: IN_PROGRESS as the next sequence begins processing.

**Job Completion Times**

3D point cloud labeling jobs can take workers hours to complete. You can set the total amount of time that workers can work on each task when you create a labeling job. The maximum time you can set for workers to work on tasks is 7 days. The default value is 3 days.

It is strongly recommended that you create tasks that workers can complete within 12 hours. Workers must keep the worker UI open while working on a task. They can save work as they go and Ground Truth will save their work every 15 minutes.

When using the SageMaker CreateLabelingJob API operation, set the total time a task is available to workers in the TaskTimeLimitInSeconds parameter of HumanTaskConfig.

When you create a labeling job in the console, you can specify this time limit when you select your workforce type and your work team.

**Important**

If you set your task time limit to be greater than 8 hours, you must set MaxSessionDuration for your IAM execution role to at least 8 hours. See 3D Point Cloud Labeling Job Permission Requirements (p. 257) for more information.

**Workforces**

When you create a 3D point cloud labeling job, you need to specify a work team that will complete your point cloud annotation tasks. You can choose a work team from a private workforce of your own workers, or from a vendor workforce that you select in the AWS Marketplace. You cannot use the Amazon Mechanical Turk workforce for 3D point cloud labeling jobs.

To learn more about vendor workforce, see Managing Vendor Workforces (p. 429).

To learn how to create and manage a private workforce, see Use a Private Workforce (p. 430).

**Worker User Interface (UI)**

Ground Truth provides a worker user interface (UI), tools, and assistive labeling features to help workers complete your 3D point cloud labeling tasks.

You can preview the worker UI when you create a labeling job in the console.

When you create a labeling job using the API operation CreateLabelingJob, you must provide an ARN provided by Ground Truth in the parameter HumanTaskUiArn to specify the worker UI for your task.
You can use `HumanTaskUiArn` with the SageMaker `RenderUiTemplate` API operation to preview the worker UI.

You provide worker instructions, labels, and optionally, label category attributes that are displayed in the worker UI.

**Label Category Attributes**

When you create a 3D point cloud object tracking or object detection labeling job, you can add one or more label category attributes. You can add frame attributes to all 3D point cloud task types:

- **Label category attribute** – A list of options (strings), a free form text box, or a numeric field associated with one or more labels. It is used by workers to provide metadata about a label.
- **Frame attribute** – A list of options (strings), a free form text box, or a numeric field that appears on each point cloud frame a worker is sent to annotate. It is used by workers to provide metadata about frames.

Use the following sections to learn more about these attributes. To learn how to add label category attributes to a labeling job, use the Create Labeling Job section on the task type page of your choice.

**Label Category Attributes**

Add label category attributes to labels to give workers the ability to provide more information about the annotations they create.

For example, if you add the label category `car`, you might also want to capture additional data about your labeled cars, such as if they are occluded or the size of the car. You can capture this metadata using label category attributes. In this example, if you added the attribute `occluded` to the car label category, you can assign `partial`, `completely`, `no` to the `occluded` attribute and enable workers to select one of these options.

A label category attribute is added to an individual label, or to all labels. When a label category attribute is applied to all labels it is referred to as a global label category attribute.

**Label Frame Attributes**

Add frame attributes to give workers the ability to provide more information about individual point cloud frames.

For example, you can add a frame attribute that allows workers to enter a number. You may want to use this attribute to have workers identify the number of objects they see in a particular frame.

In another example, you may want to provide a free-form text box to give workers the ability to provide a free form answer to a question.

You can specify up to 10 frame attributes, and these attributes will appear on all frames.

**Worker Instructions**

You can provide worker instructions to help your workers complete your point cloud labeling tasks. You might want to use these instructions to do the following:

- Best practices and things to avoid when annotating objects.
- Explanation of the label category attributes provided (for object detection and object tracking tasks), and how to use them.
- Advice on how to save time while labeling by using keyboard shortcuts.
You can add your worker instructions using the SageMaker console while creating a labeling job. If you create a labeling job using the API operation CreateLabelingJob, you specify worker instructions in your label category configuration file.

In addition to your instructions, Ground Truth provides a link to help workers navigate and use the worker portal. View these instructions by selecting the task type on Worker Instructions (p. 257).

### 3D Point Cloud Labeling Job Permission Requirements

When you create a 3D point cloud labeling job, in addition to the permission requirements found in Assign IAM Permissions to Use Ground Truth (p. 416), you must add a CORS policy to your S3 bucket that contains your input manifest file.

Additionally if you choose to allow workers to work on tasks for more than 8 hours, you must increase the MaxSessionDuration of the IAM execution role you use to create the labeling job.

#### Add a CORS Permission Policy to S3 Bucket

When you create a 3D point cloud labeling job, you specify buckets in S3 where your input data and manifest file are located and where your output data will be stored. These buckets may be the same. You must attach the following Cross-origin resource sharing (CORS) policy to your input and output buckets:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<CORSConfiguration xmlns="http://s3.amazonaws.com/doc/2006-03-01/">
  <CORSRule>
    <AllowedOrigin>*</AllowedOrigin>
    <AllowedMethod>GET</AllowedMethod>
    <AllowedMethod>HEAD</AllowedMethod>
    <AllowedMethod>PUT</AllowedMethod>
    <MaxAgeSeconds>3000</MaxAgeSeconds>
    <ExposeHeader>Access-Control-Allow-Origin</ExposeHeader>
    <AllowedHeader>*</AllowedHeader>
  </CORSRule>
</CORSConfiguration>
```

To learn how to add a CORS policy to an S3 bucket, see How do I add cross-domain resource sharing with CORS? in the Amazon Simple Storage Service Console User Guide.

#### Increase MaxSessionDuration for Execution Role

3D point cloud labeling tasks can take workers more time to complete than other task types. You can set the total amount of time that workers can work on each task by doing one of the following:

- When you create a labeling job in the console, set Task Timeout when you select your work team.
- Using the TaskTimeLimitInSeconds parameter when creating a labeling job using the SageMaker API.

The maximum time you can set for workers to work on tasks is 7 days. The default value is 3 days. If you set your task time limit to be greater than 8 hours, you must set MaxSessionDuration for your IAM execution role to at least 8 hours.

To see how to update this value for your IAM role, see Modifying a Role in the IAM User Guide, choose your preferred method to modify the role, and then follow the steps in Modifying a Role Maximum Session Duration. To learn more about permission requirements for the Ground Truth execution role, see Create an Execution Role to Start a Labeling Job (p. 420).

### Worker Instructions

This topic provides an overview of the Ground Truth worker portal and the tools available to complete your 3D Point Cloud labeling task. First, select the type of task you are working on from Topics.
For adjustment jobs, select the original labeling job task type that produced the labels you are adjusting. Review and adjust the labels in your task as needed.

**Topics**
- 3D Point Cloud Semantic Segmentation (p. 258)
- 3D Point Cloud Object Detection (p. 266)
- 3D Point Cloud Object Tracking (p. 276)

### 3D Point Cloud Semantic Segmentation

Use this page to become familiar with the user interface and tools available to complete your 3D point cloud semantic segmentation task.

**Topics**
- Your Task (p. 258)
- Navigate the UI (p. 263)
- Icon Guide (p. 265)
- Shortcuts (p. 266)
- Saving Your Work and Submitting (p. 266)

**Your Task**

When you work on a 3D point cloud semantic segmentation task, you need to select a category from the **Annotations** menu on the right side of your worker portal using the drop down menu **Label Categories**. After you've selected a category, use the paint brush and polygon tools to paint each object in the 3D point cloud that this category applies to. For example, if you select the category **Car**, you would use these tools to paint all of the cars in the point cloud. The following video demonstrates how to use the paint brush tool to paint an object.

If you see one or more images in your worker portal, you can paint in the images or paint in the 3D point cloud and the paint will show up in the other medium.

You may see frame attributes under the **Labels** menu. Use these attribute prompts to enter additional information about the point cloud.
If you see that objects have already been painted when you open the task, adjust those annotations.

The following video includes an image that can be annotated. You may not see an image in your task.
After you've painted one or more objects using a label category, you can select that category from the Label Category menu on the right to only view points painted for that category.
Navigate the UI

You can navigate in the 3D scene using their keyboard and mouse. You can:

- Double click on specific objects in the point cloud to zoom into them.
- Use a mouse-scroller or trackpad to zoom in and out of the point cloud.
- Use both keyboard arrow keys and Q, E, A, and D keys to move Up, Down, Left, Right. Use keyboard keys W and S to zoom in and out.

The following video demonstrates movements around the 3D point cloud and in the side-view. You can hide and re-expand all side views using the full screen icon. In this GIF, the side-views and menus have been collapsed.
When you are in the worker UI, you see the following menus:

- **Instructions** – Review these instructions before starting your task.
- **Shortcuts** – Use this menu to view keyboard shortcuts that you can use to navigate the point cloud and use the annotation tools provided.
- **View** – Use this menu to toggle different view options on and off. For example, you can use this menu to add a ground mesh to the point cloud, and to choose the projection of the point cloud.
- **3D Point Cloud** – Use this menu to add additional attributes to the points in the point cloud, such as color, and pixel intensity. Note that some or all of these options may not be available.
- **Paint** – Use this menu to modify the functionality of the paint brush.

When you open a task, the move scene icon is on, and you can move around the point cloud using your mouse and the navigation buttons in the point cloud area of the screen. To return to the original view you see when you first opened the task, choose the reset scene icon.

After you select the paint icon, you can add paint to the point cloud and images (if included). You must select the move scene icon again to move to another area in the 3D point cloud or image.

To collapse all panels on the right and make the 3D point cloud full screen, select the full screen icon.

For the camera images and side-panels, you have the following view options:

- **C** – View the camera angle on point cloud view.
- **F** – View the frustum, or field of view, of the camera used to capture that image on point cloud view.
- **P** – View the point cloud overlaid on the image.

**Icon Guide**

Use this table to learn about the icons available in your worker task portal.

<table>
<thead>
<tr>
<th>Icon</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Brush" /></td>
<td>brush</td>
<td>Choose this icon to turn on the brush tool. To use with this tool, choose and move over the objects that you want to paint with your mouse. After you choose it, everything you paint be associated with the category you chose.</td>
</tr>
<tr>
<td><img src="image" alt="Polygon" /></td>
<td>polygon</td>
<td>Choose this icon to use the polygon paint tool. Use this tool to draw polygons around objects that you want to paint. After you choose it, everything you draw a polygon around will be associated with the category you have chosen.</td>
</tr>
<tr>
<td><img src="image" alt="Reset Scene" /></td>
<td>reset scene</td>
<td>Choose this icon to reset the view of the point cloud, side panels, and if applicable, all images to their original position when the task was first opened.</td>
</tr>
<tr>
<td><img src="image" alt="Move Scene" /></td>
<td>move scene</td>
<td>Choose this icon to move the scene. By default, this icon will be selected when you first start a task.</td>
</tr>
</tbody>
</table>
Icon | Name | Description
--- | --- | ---
full screen | Choose this icon to make the 3D point cloud visualization full screen, and to collapse all side panels.

Shortcuts

The shortcuts listed in the Shortcuts menu can help you navigate the 3D point cloud and use the paint tool.

Before you start your task, it is recommended that you review the Shortcuts menu and become acquainted with these commands.

Saving Your Work and Submitting

You should periodically save your work. Ground Truth will automatically save your work ever 15 minutes.

When you open a task, you must complete your work on it before pressing Submit. If you select Stop Working you will loose that task, and other workers will be able to start working on it.

3D Point Cloud Object Detection

Use this page to become familiarize with the user interface and tools available to complete your 3D point cloud object detection task.

Topics

- Your Task (p. 266)
- Navigate the UI (p. 268)
- Icon Guide (p. 275)
- Shortcuts (p. 275)
- Saving Your Work and Submitting (p. 276)

Your Task

When you work on a 3D point cloud object detection task, you need to select a category from the Annotations menu on the right side of your worker portal using the Label Categories menu. After you've chosen a category, use the add cuboid and fit cuboid tools to fit a cuboid around objects in the 3D point cloud that this category applies to. After you place a cuboid, you can modify its dimensions, location, and orientation directly in the point cloud, and the three panels shown on the right.

If you see one or more images in your worker portal, you can also modify cuboids in the images or in the 3D point cloud and the edits will show up in the other medium.

If you see cuboids have already been added to the 3D point cloud when you open your task, adjust those cuboids and add additional cuboids as needed.

To edit a cuboid, including moving, re-orienting, and changing cuboid dimensions, you must use shortcut keys. You can see a full list of shortcut keys in the Shortcuts menu in your UI. The following are important key-combinations that you should become familiar with before starting your labeling task.

<table>
<thead>
<tr>
<th>Mac Command</th>
<th>Windows Command</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cmd + Drag</td>
<td>Ctrl + Drag</td>
<td>Modify the dimensions of the cuboid.</td>
</tr>
</tbody>
</table>
### Mac Command | Windows Command | Action
--- | --- | ---
Option + Drag | Alt + Drag | Move the cuboid.
Shift + Drag | Shift + Drag | Rotate the cuboid.
Option + O | Alt + O | Fit the cuboid tightly around the points it has been drawn around. Before using the option, make sure the cuboid fully-surrounds the object of interest.
Option + G | Alt + G | Set the cuboid to the ground.

Individual labels may have one or more label attributes. If a label has a label attribute associated with it, it will appear when you select the downward pointing arrow next to the label from the **Label Id** menu. Fill in required values for all label attributes.

You may see frame attributes under the **Labels** menu. Use these attribute prompts to enter additional information about each frame.
Navigate the UI

You can navigate in the 3D scene using your keyboard and mouse. You can:

- Double click on specific objects in the point cloud to zoom into them.
- You can use the [ and ] keys on your keyboard to zoom into and move from one label to the next. If no label is selected, when you select [ or ], the UI will zoom into the first label in the **Label Id** list.
- Use a mouse-scroller or trackpad to zoom in and out of the point cloud.
- Use both keyboard arrow keys and Q, E, A, and D keys to move Up, Down, Left, Right. Use keyboard keys W and S to zoom in and out.
Once you place a cuboids in the 3D scene, a side-view will appear with three projected views: top, side, and back. These side-views show points in and around the placed cuboid and help workers refine cuboid boundaries in that area. Workers can zoom in and out of each of those side-views using their mouse.

The following video demonstrates movements around the 3D point cloud and in the side-view.
When you are in the worker UI, you see the following menus:

- **Instructions** – Review these instructions before starting your task.
- **Shortcuts** – Use this menu to view keyboard shortcuts that you can use to navigate the point cloud and use the annotation tools provided.
- **Label** – Use this menu to modify a cuboid. First, select a cuboid, and then choose an option from this menu. This menu includes assistive labeling tools like setting a cuboid to the ground and automatically fitting the cuboid to the object's boundaries.
- **View** – Use this menu to toggle different view options on and off. For example, you can use this menu to add a ground mesh to the point cloud, and to choose the projection of the point cloud.
- **3D Point Cloud** – Use this menu to add additional attributes to the points in the point cloud, such as color, and pixel intensity. Note that these options may not be available.

When you open a task, the move scene icon is on, and you can move around the point cloud using your mouse and the navigation buttons in the point cloud area of the screen. To return to the original view you see when you first opened the task, choose the reset scene icon. Resetting the view will not modify your annotations.

After you select the add cuboid icon, you can add cuboids to the 3D point cloud visualization. Once you've added a cuboid, you can adjust it in the three views (top, side, and front) and in the images (if included).
Point cloud View
You must choose the move scene icon again to move to another area in the 3D point cloud or image.

To collapse all panels on the right and make the 3D point cloud full-screen, choose the full screen icon.

If camera images are included, you may have the following view options:

- C – View the camera angle on point cloud view.
- F – View the frustum, or field of view, of the camera used to capture that image on point cloud view.
- P – View the point cloud overlaid on the image.
- B – View cuboids in the image.

The following video demonstrates how to use these view options. The F option is used to view the field of view of the camera (the gray area), the C options shows the direction the camera is facing and angle of the camera (blue lines), and the B option is used to view the cuboid.
Icon Guide

Use this table to learn about the icons you see in your worker task portal.

<table>
<thead>
<tr>
<th>Icon</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="add cuboid icon" /></td>
<td>add cuboid</td>
</tr>
<tr>
<td><img src="image" alt="edit cuboid icon" /></td>
<td>edit cuboid</td>
</tr>
<tr>
<td><img src="image" alt="reset scene icon" /></td>
<td>reset scene</td>
</tr>
<tr>
<td><img src="image" alt="move scene icon" /></td>
<td>move scene</td>
</tr>
<tr>
<td><img src="image" alt="full screen icon" /></td>
<td>full screen</td>
</tr>
<tr>
<td><img src="image" alt="show labels icon" /></td>
<td>show labels</td>
</tr>
<tr>
<td><img src="image" alt="hide labels icon" /></td>
<td>hide labels</td>
</tr>
<tr>
<td><img src="image" alt="hide labels icon" /></td>
<td>hide labels</td>
</tr>
</tbody>
</table>

Shortcuts

The shortcuts listed in the **Shortcuts** menu can help you navigate the 3D point cloud and use tools to add and edit cuboids.
Before you start your task, it is recommended that you review the Shortcuts menu and become acquainted with these commands. You need to use some of the 3D cuboid controls to edit your cuboid.

**Saving Your Work and Submitting**

You should periodically save your work. Ground Truth will automatically save your work every 15 minutes.

When you open a task, you must complete your work on it before pressing **Submit**. If you select **Stop Working** you will lose that task, and other workers will be able to start working on it.

**3D Point Cloud Object Tracking**

Use this page to become familiar with the user interface and tools available to complete your 3D point cloud object detection task.

**Topics**

- Your Task (p. 276)
- Navigate the UI (p. 281)
- Icon Guide (p. 286)
- Shortcuts (p. 287)
- Saving Your Work and Submitting (p. 287)

**Your Task**

When you work on a 3D point cloud object tracking task, you need to select a category from the **Annotations** menu on the right side of your worker portal using the **Label Categories** menu. After you've selected a category, use the add cuboid and fit cuboid tools to fit a cuboid around objects in the 3D point cloud that this category applies to. After you place a cuboid, you can modify its location, dimensions, and orientation directly in the point cloud, and the three panels shown on the right. If you see one or more images in your worker portal, you can also modify cuboids in the images or in the 3D point cloud and the edits will show up in the other medium.

**Important**

If you see cuboids have already been added to the 3D point cloud frames when you open your task, adjust those cuboids and add additional cuboids as needed.

To edit a cuboid, including moving, re-orienting, and changing cuboid dimensions, you must use shortcut keys. You can see a full list of shortcut keys in the **Shortcuts** menu in your UI. The following are important key-combinations that you should become familiar with before starting your labeling task.

<table>
<thead>
<tr>
<th>Mac Command</th>
<th>Windows Command</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cmd + Drag</td>
<td>Ctrl + Drag</td>
<td>Modify the dimensions of the cuboid.</td>
</tr>
<tr>
<td>Option + Drag</td>
<td>Alt + Drag</td>
<td>Move the cuboid.</td>
</tr>
<tr>
<td>Shift + Drag</td>
<td>Shift + Drag</td>
<td>Rotate the cuboid.</td>
</tr>
<tr>
<td>Option + O</td>
<td>Alt + O</td>
<td>Fit the cuboid tightly around the points it has been drawn around. Before using the option, make sure the cuboid fully-surrounds the object of interest.</td>
</tr>
<tr>
<td>Option + G</td>
<td>Alt + G</td>
<td>Set the cuboid to the ground.</td>
</tr>
</tbody>
</table>
When you open your task, two frames will be loaded. If your task includes more than two frames, you need to use the navigation bar in the lower-left corner, or the load frames icon to load additional frames. You should annotate and adjust labels in all frames before submitting.

After you fit a cuboid tightly around the boundaries of an object, navigate to another frame using the navigation bar in the lower-right corner of the UI. If that same object has moved to a new location, add another cuboid and fit it tightly around the boundaries of the object. Each time you manually add a cuboid, you see the frame sequence bar in the lower-left corner of the screen turn red where that frame is located temporally in the sequence.

The following video shows how, if you add a cuboid in one frame, and then adjust it in another, your UI will automatically infer the location of the cuboid in all of the frames in-between.
Your UI automatically infers the location of that object in all other frames after you've placed a cuboid. You can see the movement of that object, and the inferred and manually created cuboids using the arrows. Adjust inferred cuboids as needed. The following video demonstrates how to navigate between frames.
Individual labels may have one or more label attributes. If a label has a label attribute associated with it, it will appear when you select the downward pointing arrow next to the label from the Label Id menu. Fill in required values for all label attributes.

You may see frame attributes under the Label Id menu. These attributes will appear on each frame in your task. Use these attribute prompts to enter additional information about each frame.

Navigate the UI

You can navigate in the 3D scene using your keyboard and mouse. You can:

- Double click on specific objects in the point cloud to zoom into them.
- You can use the [ and ] keys on your keyboard to zoom into and move from one label to the next. If no label is selected, when you select [ or ], the UI will zoom into the first label in the Label Id list.
- Use a mouse-scroller or trackpad to zoom in and out of the point cloud.
• Use both keyboard arrow keys and Q, E, A, and D keys to move Up, Down, Left, Right. Use keyboard keys W and S to zoom in and out.

Once you place a cuboids in the 3D scene, a side-view will appear with three projected views: top, side, and back. These side-views show points in and around the placed cuboid and help workers refine cuboid boundaries in that area. Workers can zoom in and out of each of those side-views using their mouse.

The following video demonstrates movements around the 3D point cloud and in the side-view.
Point cloud View x

Q W E
A S D
When you are in the worker UI, you see the following menus:

- **Instructions** – Review these instructions before starting your task.
- **Shortcuts** – Use this menu to view keyboard shortcuts that you can use to navigate the point cloud and use the annotation tools provided.
- **Label** – Use this menu to modify a cuboid. First, select a cuboid, and then choose an option from this menu. This menu includes assistive labeling tools like setting a cuboid to the ground and automatically fitting the cuboid to the object's boundaries.
- **View** – Use this menu to toggle different view options on and off. For example, you can use this menu to add a ground mesh to the point cloud, and to choose the projection of the point cloud.
- **3D Point Cloud** – Use this menu to add additional attributes to the points in the point cloud, such as color, and pixel intensity. Note that these options may not be available.

When you open a task, the move scene icon is on, and you can move around the point cloud using your mouse and the navigation buttons in the point cloud area of the screen. To return to the original view you see when you first opened the task, choose the reset scene icon.

After you select the add cuboid icon, you can add cuboids to the point cloud and images (if included). You must select the move scene icon again to move to another area in the 3D point cloud or image.

To collapse all panels on the right and make the 3D point cloud full-screen, choose the full screen icon.

If camera images are included, you may have the following view options:

- **C** – View the camera angle on point cloud view.
- **F** – View the frustum, or field of view, of the camera used to capture that image on point cloud view.
- **P** – View the point cloud overlaid on the image.
- **B** – View cuboids in the image.

The following video demonstrates how to use these view options. The **F** option is used to view the field of view of the camera (the gray area), the **C** options shows the direction the camera is facing and angle of the camera (blue lines), and the **B** option is used to view the cuboid.
Icon Guide

Use this table to learn about the icons you see in your worker task portal.

<table>
<thead>
<tr>
<th>Icon</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>add cuboid</td>
<td>Choose this icon to add a cuboid. Each cuboid you add is associated with the category you chose.</td>
</tr>
<tr>
<td>edit cuboid</td>
<td>Choose this icon to edit a cuboid. After you add a cuboid, you can edit its dimensions, location, and orientation. After a cuboid is added, it automatically switches to edit cuboid mode.</td>
</tr>
<tr>
<td>reset scene</td>
<td>Choose this icon to reset the view of the point cloud, side panels, and if applicable, all images to their original position when the task was first opened.</td>
</tr>
<tr>
<td>move scene</td>
<td>Choose this icon to move the scene. By default, this icon is chosen when you first start a task.</td>
</tr>
<tr>
<td>full screen</td>
<td>Choose this icon to make the 3D point cloud visualization full screen and to collapse all side panels.</td>
</tr>
<tr>
<td>load frames</td>
<td>Choose this icon to load additional frames.</td>
</tr>
<tr>
<td>hide labels</td>
<td>Hide labels in the 3D point cloud visualization, and if applicable, in images.</td>
</tr>
<tr>
<td>show labels</td>
<td>Show labels in the 3D point cloud visualization, and if applicable, in images.</td>
</tr>
<tr>
<td>hide labels</td>
<td>Delete a label. This option can only be used to delete labels you have manually created or adjusted.</td>
</tr>
</tbody>
</table>
Shortcuts

The shortcuts listed in the **Shortcuts** menu can help you navigate the 3D point cloud and use tools to add and edit cuboids.

Before you start your task, it is recommended that you review the **Shortcuts** menu and become acquainted with these commands. You need to use some of the 3D cuboid controls to edit your cuboid.

Saving Your Work and Submitting

You should periodically save your work. Ground Truth will automatically save your work every 15 minutes.

When you open a task, you must complete your work on it before pressing **Submit**. If you select **Stop Working** you will lose that task, and other workers will be able to start working on it.

Verify and Adjust Labels

When the labels on a dataset need to be validated, Amazon SageMaker Ground Truth provides functionality to have workers verify that labels are correct or to adjust previous labels.

These types of jobs fall into two distinct categories:

- **Label verification** — Workers indicate if the existing labels are correct, or rate their quality, and can add comments to explain their reasoning. Workers will not be able to modify or adjust labels.
- **Label adjustment** — Workers adjust prior annotations to correct them.

The following Ground Truth built-in task types support adjustment and verification labeling jobs:

- Bounding box
- Semantic segmentation
- 3D point cloud object detection, 3D point cloud object tracking, and 3D point cloud semantic segmentation
- All video frame object detection and video frame object tracking task types — bounding box, polyline, polygon and keypoint

You can start a label verification and adjustment jobs using the SageMaker console or the API.

Create and Start a Label Verification Job (Console)

Bounding box and semantic segmentation label verification jobs can be created in the console. You must use the Ground Truth API to create a 3D point cloud or video frame verification job. To learn how, see the section 3D Point Cloud and Video Frame (p. 289) under Start a Label Verification or Adjustment Job (API).

**To start a label verification job (console)**

1. Open the SageMaker console: console.aws.amazon.com/sagemaker/ and choose **Labeling jobs**.
2. Start a new labeling job by chaining (p. 412) a prior job or start from scratch, specifying an input manifest that contains labeled data objects.
3. In the **Task type** pane, select **Label verification**.
4. Choose **Next**.
5. In the **Existing-labels display options** pane, the system shows the available label attribute names in your manifest. Choose the label attribute name for the labeling job that you want to verify.
6. Use the instructions areas of the tool designer to provide context about what the previous labellers were asked to do and what the current verifiers need to check.

You can add new labels that workers choose from to verify labels. For example, you can ask workers to verify the image quality, and provide the labels *Clear* and *Blurry*. Workers will also have the option to add a comment to explain their selection.

7. Choose **See preview** to check that the tool is displaying the prior labels correctly and presents the label verification task clearly.

8. Select **Create**. This will create and start your labeling job.

### Start a Label Verification or Adjustment Job (API)

Start a label verification or adjustment job by chaining a successfully completed job or starting a new job from scratch using the `CreateLabelingJob` operation. The procedure is almost the same as setting up a new labeling job with `CreateLabelingJob`, with a few modifications. Use the following sections to learn what modifications are required to chain a labeling job to create an adjustment or verification labeling job.

**Important**

The label category configuration file you identify for an adjustment or verification job in `LabelCategoryConfigS3Uri` of `CreateLabelingJob` must contain the same labels used in the original labeling job. You can add new labels. For 3D point cloud and video frame jobs, you can add new label category and frame attributes to the label category configuration file.

**Bounding Box and Semantic Segmentation**

To create a bounding box or semantic segmentation label verification or adjustment job, use the following guidelines to specify API attributes for the `CreateLabelingJob` operation.

- Use the `LabelAttributeName` parameter to specify the output label name that you want to use for verified or adjusted labels.
Verify and Adjust Labels

If you are chaining the job, the labels from the previous labeling job to be adjusted or verified will be specified in the custom UI template. To learn how to create a custom template, see Create Custom Worker Task Template (p. 1566).

Identify the location of the UI template in the `UiTemplateS3Uri` parameter. SageMaker provides widgets that you can use in your custom template to display old labels. Use the `initial-value` attribute in one of the following crowd elements to extract the labels that need verification or adjustment and include them in your task template:

- `crowd-semantic-segmentation` (p. 506)—Use this crowd element in your custom UI task template to specify semantic segmentation labels that need to be verified or adjusted.
- `crowd-bounding-box` (p. 454)—Use this crowd element in your custom UI task template to specify bounding box labels that need to be verified or adjusted.

The `LabelCategoryConfigS3Uri` parameter must contain the same label categories as the previous labeling job.

Use the bounding box or semantic segmentation adjustment or verification lambda ARNs for `PreHumanTaskLambdaArn` and `AnnotationConsolidationLambdaArn`:

- For bounding box, the adjustment labeling job lambda function ARNs end with `AdjustmentBoundingBox` and the verification lambda function ARNs end with `VerificationBoundingBox`.
- For semantic segmentation, the adjustment labeling job lambda function ARNs end with `AdjustmentSemanticSegmentation` and the verification lambda function ARNs end with `VerificationSemanticSegmentation`.

3D Point Cloud and Video Frame

Use the `LabelAttributeName` parameter to specify the output label name that you want to use for verified or adjusted labels.

You must use the human task UI Amazon Resource Name (ARN) (`HumanTaskUiArn`) used for the original labeling job. To see supported ARNs, see `HumanTaskUiArn`.

In the label category configuration file, you must specify the label attribute name (`LabelAttributeName`) of the previous labeling job that you use to create the adjustment or verification labeling job in the `auditLabelAttributeName` parameter.

You specify whether your labeling job is a verification or adjustment labeling job using the `editsAllowed` parameter in your label category configuration file identified by the `LabelCategoryConfigS3Uri` parameter.

For verification labeling jobs, you must use the `editsAllowed` parameter to specify that all labels cannot be modified. `editsAllowed` must be set to "none" in each entry in labels. Optionally, you can specify whether or not label categories attributes and frame attributes can be adjusted by workers.

Optionally, for adjustment labeling jobs, you can use the `editsAllowed` parameter to specify labels, label category attributes, and frame attributes that can or cannot be modified by workers. If you do not use this parameter, all labels, label category attributes, and frame attributes will be adjustable.

To learn more about the `editsAllowed` parameter and configuring your label category configuration file, see Label Category Configuration File Schema (p. 326).

Use the 3D point cloud or video frame adjustment lambda ARNs for `PreHumanTaskLambdaArn` and `AnnotationConsolidationLambdaArn` for both adjustment and verification labeling jobs:

- For 3D point clouds, the adjustment and verification labeling job lambda function ARNs end with `Adjustment3DPointCloudSemanticSegmentation`, `Adjustment3DPointCloudObjectTracking`, and `Adjustment3DPointCloudObjectDetection` for 3D point cloud semantic segmentation, object detection, and object tracking respectively.
• For video frames, the adjustment and verification labeling job lambda function ARNs end with `AdjustmentVideoObjectDetection` and `AdjustmentVideoObjectTracking` for video frame object detection and object tracking respectively.

Ground Truth stores the output data from a label verification or adjustment job in the S3 bucket that you specified in the `S3OutputPath` parameter of the `CreateLabelingJob` operation. For more information about the output data from a label verification or adjustment labeling job, see Label Verification and Adjustment Data in the Output Manifest (p. 290).

**Label Verification and Adjustment Data in the Output Manifest**

Amazon SageMaker Ground Truth writes label verification data to the output manifest within the metadata for the label. It adds two properties to the metadata:

• A `type` property, with a value of "groundtruth/label-verification".

• A `worker-feedback` property, with an array of `comment` values. This property is added when the worker enters comments. If there are no comments, the field doesn't appear.

The following example output manifest shows how label verification data appears:

```
{
  "source-ref": "S3 bucket location",
  "verify-bounding-box": "1",
  "verify-bounding-box-metadata": {
    "class-name": "bad",
    "confidence": 0.93,
    "type": "groundtruth/label-verification",
    "job-name": "verify-bounding-boxes",
    "human-annotated": "yes",
    "creation-date": "2018-10-18T22:18:13.527256",
    "worker-feedback": [
      {"comment": "The bounding box on the bird is too wide on the right side."},
      {"comment": "The bird on the upper right is not labeled."}
    ]
  }
}
```

The worker output of adjustment tasks resembles the worker output of the original task, except that it contains the adjusted values and an `adjustment-status` property with the value of `adjusted` or `unadjusted` to indicate whether an adjustment was made.

For more examples of the output of different tasks, see Output Data (p. 381).

**Cautions and Considerations**

To get expected behavior when creating a label verification or adjustment job, carefully verify your input data.

• If you are using image data, verify that your manifest file contains hexadecimal RGB color information.

• To save money on processing costs, filter your data to ensure you are not including unwanted objects in your labeling job input manifest.

• Add required Amazon S3 permissions to ensure your input data is processed correctly.
Color Information Requirements for Semantic Segmentation Jobs

To properly reproduce color information in verification or adjustment tasks, the tool requires hexadecimal RGB color information in the manifest (for example, #FFFFFF for white). When you set up a Semantic Segmentation verification or adjustment job, the tool examines the manifest to determine if this information is present. If it can't find it, Amazon SageMaker Ground Truth displays an error message and ends job setup.

In prior iterations of the Semantic Segmentation tool, category color information wasn't output in hexadecimal RGB format to the output manifest. That feature was introduced to the output manifest at the same time the verification and adjustment workflows were introduced. Therefore, older output manifests aren't compatible with this new workflow.

Filter Your Data Before Starting the Job

Amazon SageMaker Ground Truth processes all objects in your input manifest. If you have a partially labeled data set, you might want to create a custom manifest using an Amazon S3 Select query on your input manifest. Unlabeled objects individually fail, but they don't cause the job to fail, and they might incur processing costs. Filtering out objects you don't want verified reduces your costs.

If you create a verification job using the console, you can use the filtering tools provided there. If you create jobs using the API, make filtering your data part of your workflow where needed.

Creating Custom Labeling Workflows

This document guides you through the process of setting up a workflow with a custom labeling template. For more information about starting a labeling job, see Getting started (p. 162). In that section, when you choose the Task type, select Custom labeling task, and then follow this section's instructions to configure it.

Topics
- Step 1: Setting up your workforce (p. 291)
- Step 2: Creating your custom labeling task template (p. 292)
- Demo Template: Annotation of Images with crowd-bounding-box (p. 297)
- Demo Template: Labeling Intents with crowd-classifier (p. 302)
- Step 3: Processing with AWS Lambda (p. 308)
- Custom Workflows via the API (p. 311)

Step 1: Setting up your workforce

In this step you use the console to establish which worker type to use and make the necessary sub-selections for the worker type. It assumes you have already completed the steps up to this point in the Getting started (p. 162) section and have chosen the Custom labeling task as the Task type.

To configure your workforce.

1. First choose an option from the Worker types. There are three types currently available:
   - Public uses an on-demand workforce of independent contractors, powered by Amazon Mechanical Turk. They are paid on a per-task basis.
   - Private uses your employees or contractors for handling data that needs to stay within your organization.
   - Vendor uses third party vendors that specialize in providing data labeling services, available via the AWS Marketplace.
2. If you choose the Public option, you are asked to set the number of workers per dataset object. Having more than one worker perform the same task on the same object can help increase the accuracy of your results. The default is three. You can raise or lower that depending on the accuracy you need.

You are also asked to set a price per task by using a drop-down menu. The menu recommends price points based on how long it will take to complete the task.

The recommended method to determine this is to first run a short test of your task with a private workforce. The test provides a realistic estimate of how long the task takes to complete. You can then select the range your estimate falls within on the Price per task menu. If your average time is more than 5 minutes, consider breaking your task into smaller units.

Next

Step 2: Creating your custom labeling task template (p. 292)

Step 2: Creating your custom labeling task template

Topics
• Starting with a base template (p. 292)
• Developing templates locally (p. 292)
• Using External Assets (p. 293)
• Track your variables (p. 293)
• A simple sample (p. 293)
• Adding automation with Liquid (p. 295)
• End-to-end demos (p. 297)
• Next (p. 297)

Starting with a base template

To get you started, the Task type starts with a drop-down menu listing a number of our more common task types, plus a custom type. Choose one and the code editor area will be filled with a sample template for that task type. If you prefer not to start with a sample, choose Custom HTML for a minimal template skeleton.

If you've already created a template, upload the file directly using the Upload file button in the upper right of the task setup area or paste your template code into the editor area. For a repository of demo templates for a variety of labeling job task types, see Amazon SageMaker Ground Truth Sample Task UIs.

Developing templates locally

While you need to be in the console to test how your template will process incoming data, you can test the look and feel of your template's HTML and custom elements in your browser by adding this code to the top of your HTML file.

Example

```html
<script src="https://assets.crowd.aws/crowd-html-elements.js"></script>
```

This loads the necessary code to render the custom HTML elements. Use this if you want to develop your template's look and feel in your preferred editor rather than in the console.
Remember, though, this will not parse your variables. You may want to replace them with sample content while developing locally.

**Using External Assets**

Amazon SageMaker Ground Truth custom templates allow external scripts and style sheets to be embedded.

**Example**

```html
<script src="https://www.example.com/my-enhancement-script.js"></script>
<link rel="stylesheet" type="text/css" href="https://www.example.com/my-enhancement-styles.css">
```

If you encounter errors, ensure that your originating server is sending the correct MIME type and encoding headers with the assets.

For example, the MIME and encoding types for remote scripts: `application/javascript; CHARSET=UTF-8`.

The MIME and encoding type for remote stylesheets: `text/css; CHARSET=UTF-8`.

**Track your variables**

In the process of building the sample below, there will be a step that adds variables to it to represent the pieces of data that may change from task to task, worker to worker. If you're starting with one of the sample templates, you will need to make sure you're aware of the variables it already uses. When you create your pre-annotation AWS Lambda script, its output will need to contain values for any of those variables you choose to keep.

The values you use for the variables can come from your manifest file. All the key-value pairs in your data object are provided to your pre-annotation Lambda. If it's a simple pass-through script, matching keys for values in your data object to variable names in your template is the easiest way to pass those values through to the tasks forms your workers see.

**A simple sample**

All tasks begin and end with the `<crowd-form> </crowd-form>` elements. Like standard HTML `<form>` elements, all of your form code should go between them.

For a simple tweet-analysis task, use the `<crowd-classifier>` element. It requires the following attributes:

- `name` - the variable name to use for the result in the form output.
- `categories` - a JSON formatted array of the possible answers.
- `header` - a title for the annotation tool

As children of the `<crowd-classifier>` element, you must have three regions.

- `<classification-target>` - the text the worker will classify based on the options specified in the categories attribute above.
- `<full-instructions>` - instructions that are available from the "View full instructions" link in the tool. This can be left blank, but it is recommended that you give good instructions to get better results.
- `<short-instructions>` - a more brief description of the task that appears in the tool's sidebar. This can be left blank, but it is recommended that you give good instructions to get better results.
A simple version of this tool would look like this.

**Example of using crowd-classifier**

```html
<script src="https://assets.crowd.aws/crowd-html-elements.js"></script>
<crowd-form>
  <crowd-classifier
    name="tweetFeeling"
    categories="["positive","negative","neutral","unclear"]"
    header="Which term best describes this tweet?"
  >
    <classification-target>
      My favorite football team won today!
      Bring on the division finals!
    </classification-target>

    <full-instructions header="Sentiment Analysis Instructions">
      Try to determine the sentiment the author of the tweet is trying to express.
      If none seem to match, choose "cannot determine."
    </full-instructions>

    <short-instructions>
      Pick the term best describing the sentiment of the tweet.
    </short-instructions>
  </crowd-classifier>
</crowd-form>
```

You can copy and paste the code into the editor in the Ground Truth labeling job creation workflow to preview the tool, or try out a demo of this code on CodePen.
Adding automation with Liquid

Our custom template system uses Liquid for automation. It is an open source inline markup language. For more information and documentation, visit the Liquid homepage.

The most common use of Liquid will be to parse the data coming from your pre-annotation Lambda and pull out the relevant variables to create the task. In Liquid, the text between single curly braces and percent symbols is an instruction or "tag" that creates control flow. Text between double curly braces is a variable or "object" which outputs its value.

The taskInput object returned by your Pre-annotation Lambda (p. 308) will be available as the task.input object in your templates.

The properties in your manifest's data objects are passed into your Pre-annotation Lambda (p. 308) as the event.dataObject. A simple pass-through script simply returns that object as the taskInput object. You would represent values from your manifest as variables as follows.

Example Manifest data object

```json
{
  "source": "This is a sample text for classification",
  "labels": [ "angry", "sad", "happy", "inconclusive" ],
  "header": "What emotion is the speaker feeling?"
}
```

Example Sample HTML using variables

```html
<crowd-classifier
  name='tweetFeeling'
  categories='{{ task.input.labels | to_json }}'
  header='{{ task.input.header }}'>
  {{ task.input.source }}
</classification-target>
```

Note the addition of " | to_json" to the labels property above. That's a filter to turn the array into a JSON representation of the array. Variable filters are explained next.

Variable filters

In addition to the standard Liquid filters and actions, Ground Truth offers a few additional filters. Filters are applied by placing a pipe (|) character after the variable name, then specifying a filter name. Filters can be chained in the form of:

Example

```liquid
{{ <content> | <filter> | <filter> }}
```

Autoescape and explicit escape

By default, inputs will be HTML escaped to prevent confusion between your variable text and HTML. You can explicitly add the escape filter to make it more obvious to someone reading the source of your template that the escaping is being done.

escape_once

escape_once ensures that if you've already escaped your code, it doesn't get re-escaped on top of that. For example, so that & doesn't become &amp;.
skip_autoescape

skip_autoescape is useful when your content is meant to be used as HTML. For example, you might have a few paragraphs of text and some images in the full instructions for a bounding box.

Use skip_autoescape sparingly

The best practice in templates is to avoid passing in functional code or markup with skip_autoescape unless you are absolutely sure you have strict control over what's being passed. If you're passing user input, you could be opening your workers up to a Cross Site Scripting attack.

to_json

to_json will encode what you feed it to JSON (JavaScript Object Notation). If you feed it an object, it will serialize it.

grant_read_access

grant_read_access takes an S3 URI and encodes it into an HTTPS URL with a short-lived access token for that resource. This makes it possible to display to workers photo, audio, or video objects stored in S3 buckets that are not otherwise publicly accessible.

Example of the filters

Input

| auto-escape: {{ "Have you read 'James & the Giant Peach'?" }} |
| explicit escape: {{ "Have you read 'James & the Giant Peach'?? | escape }} |
| explicit escape_once: {{ "Have you read 'James &; the Giant Peach'?? | escape_once }} |
| skip_autoescape: {{ "Have you read 'James & the Giant Peach'?? | skip_autoescape }} |
| to_json: {{ jsObject | to_json }} |
| grant_read_access: {{ "s3://mybucket/myphoto.png" | grant_read_access }} |

Example

Output

| auto-escape: Have you read 'James & the Giant Peach'?? |
| explicit escape: Have you read 'James & the Giant Peach'?? |
| explicit escape_once: Have you read 'James &; the Giant Peach'?? |
| skip_autoescape: Have you read 'James & the Giant Peach'?? |
| to_json: { "point_number": 8, "coords": [ 59, 76 ] } |
| grant_read_access: https://s3.amazonaws.com/mybucket/myphoto.png?access token and other params |

Example of an automated classification template.

To automate the simple text classification sample, replace the tweet text with a variable.

The text classification template is below with automation added. The changes/additions are highlighted in bold.

```html
<script src="https://assets.crowd.aws/crowd-html-elements.js"></script>
<crowd-form>
  <crowd-classifier
    name="tweetFeeling"
    categories="['positive', 'negative', 'neutral', 'cannot determine']"
    header="Which term best describes this tweet?"
`
The tweet text that was in the prior sample is now replaced with an object. The `entry.taskInput` object uses `source` (or another name you specify in your pre-annotation Lambda) as the property name for the text and it is inserted directly in the HTML by virtue of being between double curly braces.

### End-to-end demos

You can view the following end-to-end demos which include sample Lambdas:

- Demo Template: Annotation of Images with `crowd-bounding-box` (p. 297)
- Demo Template: Labeling Intents with `crowd-classifier` (p. 302)

Next

Step 3: Processing with AWS Lambda (p. 308)

**Demo Template: Annotation of Images with crowd-bounding-box**

When you chose to use a custom template as your task type in the Amazon SageMaker Ground Truth console, you reach the Custom labeling task panel. There you can choose from multiple base templates. The templates represent some of the most common tasks and provide a sample to work from as you create your customized labeling task's template. If you are not using the console, or as an additional recourse, see Amazon SageMaker Ground Truth Sample Task UIs for a repository of demo templates for a variety of labeling job task types.

This demonstration works with the BoundingBox template. The demonstration also works with the AWS Lambda functions needed for processing your data before and after the task. In the Github repository above, to find templates that work with AWS Lambda functions, look for `{{ task.input.<property name> }}` in the template.

**Topics**

- Starter Bounding Box custom template (p. 298)
- Your own Bounding Box custom template (p. 298)
- Your manifest file (p. 299)
- Your pre-annotation Lambda function (p. 300)
- Your post-annotation Lambda function (p. 300)
• The output of your labeling job (p. 301)

Starter Bounding Box custom template

This is the starter bounding box template that is provided.

```html
<script src="https://assets.crowd.aws/crowd-html-elements.js"></script>
<crowd-form>
  <crowd-bounding-box
    name="boundingBox"
    src="{{ task.input.taskObject | grant_read_access }}"
    header="{{ task.input.header }}"
    labels="{{ task.input.labels | to_json | escape }}"
  >
    <!-- The <full-instructions> tag is where you will define the full instructions of your task. -->
    <full-instructions header="Bounding Box Instructions" >
      <p>Use the bounding box tool to draw boxes around the requested target of interest:</p>
      <ol>
        <li>Draw a rectangle using your mouse over each instance of the target.</li>
        <li>Make sure the box does not cut into the target, leave a 2 - 3 pixel margin.</li>
        <li>When targets are overlapping, draw a box around each object, include all contiguous parts of the target in the box. Do not include parts that are completely overlapped by another object.</li>
        <li>Do not include parts of the target that cannot be seen, even though you think you can interpolate the whole shape of the target.</li>
        <li>Avoid shadows, they're not considered as a part of the target.</li>
        <li>If the target goes off the screen, label up to the edge of the image.</li>
      </ol>
    </full-instructions>
    <!-- The <short-instructions> tag allows you to specify instructions that are displayed in the left hand side of the task interface. It is a best practice to provide good and bad examples in this section for quick reference. -->
    <short-instructions>
      Use the bounding box tool to draw boxes around the requested target of interest.
    </short-instructions>
  </crowd-bounding-box>
</crowd-form>
```

The custom templates use the Liquid template language, and each of the items between double curly braces is a variable. The pre-annotation AWS Lambda function should provide an object named `taskInput` and that object's properties can be accessed as `{{ task.input.<property name> }}` in your template.

Your own Bounding Box custom template

As an example, assume you have a large collection of animal photos in which you know the kind of animal in an image from a prior image-classification job. Now you want to have a bounding box drawn around it.

In the starter sample, there are three variables: `taskObject`, `header`, and `labels`.

Each of these would be represented in different parts of the bounding box.
• **taskObject** is an HTTP(S) URL or S3 URI for the photo to be annotated. The added `grant_read_access` is a filter that will convert an S3 URI to an HTTPS URL with short-lived access to that resource. If you're using an HTTP(S) URL, it's not needed.
• **header** is the text above the photo to be labeled, something like "Draw a box around the bird in the photo."
• **labels** is an array, represented as `[ 'item1', 'item2', ... ]`. These are labels that can be assigned by the worker to the different boxes they draw. You can have one or many.

Each of the variable names come from the JSON object in the response from your pre-annotation Lambda, The names above are merely suggested, Use whatever variable names make sense to you and will promote code readability among your team.

**Only use variables when necessary**

If a field will not change, you can remove that variable from the template and replace it with that text, otherwise you have to repeat that text as a value in each object in your manifest or code it into your pre-annotation Lambda function.

### Example: Final Customized Bounding Box Template

To keep things simple, this template will have one variable, one label, and very basic instructions. Assuming your manifest has an "animal" property in each data object, that value can be re-used in two parts of the template.

```html
<script src="https://assets.crowd.aws/crowd-html-elements.js"></script>
<crowd-form>
  <crowd-bounding-box
    name="boundingBox"
    labels="[ '{{ task.input.animal }}' ]"
    src="{{ task.input.source-ref | grant_read_access }}"
    header="Draw a box around the {{ task.input.animal }}."
  >
    <full-instructions header="Bounding Box Instructions">
      <p>Draw a bounding box around the {{ task.input.animal }} in the image. If there is more than one {{ task.input.animal }} per image, draw a bounding box around the largest one.</p>
      <p>The box should be tight around the {{ task.input.animal }} with no more than a couple of pixels of buffer around the edges.</p>
      <p>If the image does not contain a {{ task.input.animal }}, check the <strong>Nothing to label</strong> box.</p>
    </full-instructions>
    <short-instructions>
      <p>Draw a bounding box around the {{ task.input.animal }} in each image. If there is more than one {{ task.input.animal }} per image, draw a bounding box around the largest one.</p>
    </short-instructions>
  </crowd-bounding-box>
</crowd-form>
```

Note the re-use of `{{ task.input.animal }}` throughout the template. If your manifest had all of the animal names beginning with a capital letter, you could use `{{ task.input.animal | downcase }}`, incorporating one of Liquid's built-in filters in sentences where it needed to be presented lowercase.

**Your manifest file**

Your manifest file should provide the variable values you're using in your template. You can do some transformation of your manifest data in your pre-annotation Lambda, but if you don't need to, you maintain a lower risk of errors and your Lambda will run faster. Here's a sample manifest file for the template.
Your pre-annotation Lambda function

As part of the job set-up, provide the ARN of an AWS Lambda function that can be called to process your manifest entries and pass them to the template engine.

**Naming your Lambda function**

The best practice in naming your function is to use one of the following four strings as part of the function name: SageMaker, Sagemaker, sagemaker, or LabelingFunction. This applies to both your pre-annotation and post-annotation functions.

When you're using the console, if you have AWS Lambda functions that are owned by your account, a drop-down list of functions meeting the naming requirements will be provided to choose one.

In this very basic example, you're just passing through the information from the manifest without doing any additional processing on it. This sample pre-annotation function is written for Python 3.7.

```python
import json
def lambda_handler(event, context):
    return {
        "taskInput": event['dataObject']
    }
```

The JSON object from your manifest will be provided as a child of the `event` object. The properties inside the `taskInput` object will be available as variables to your template, so simply setting the value of `taskInput` to `event['dataObject']` will pass all the values from your manifest object to your template without having to copy them individually. If you wish to send more values to the template, you can add them to the `taskInput` object.

Your post-annotation Lambda function

As part of the job set-up, provide the ARN of an AWS Lambda function that can be called to process the form data when a worker completes a task. This can be as simple or complex as you want. If you want to do answer consolidation and scoring as it comes in, you can apply the scoring and/or consolidation algorithms of your choice. If you want to store the raw data for offline processing, that is an option.

**Provide permissions to your post-annotation Lambda**

The annotation data will be in a file designated by the `s3Uri` string in the `payload` object. To process the annotations as they come in, even for a simple pass through function, you need to assign S3ReadOnly access to your Lambda so it can read the annotation files.

In the Console page for creating your Lambda, scroll to the **Execution role** panel. Select **Create a new role from one or more templates**. Give the role a name. From the **Policy templates** drop-down, choose **Amazon S3 object read-only permissions**. Save the Lambda and the role will be saved and selected.

The following sample is in Python 2.7.

```python
import json
import boto3
from urlparse import urlparse
def lambda_handler(event, context):
    consolidated_labels = []
```
parsed_url = urlparse(event['payload']['s3Uri']);
s3 = boto3.client('s3')
textFile = s3.get_object(Bucket = parsed_url.netloc, Key = parsed_url.path[1:])
filecont = textFile['Body'].read()
annotations = json.loads(filecont);

for dataset in annotations:
    for annotation in dataset['annotations']:
        new_annotation = json.loads(annotation['annotationData']['content'])
        label = {
            'datasetObjectId': dataset['datasetObjectId'],
            'consolidatedAnnotation': {
                'content': {
                    event['labelAttributeName']: {
                        'workerId': annotation['workerId'],
                        'boxesInfo': new_annotation,
                        'imageSource': dataset['dataObject']
                    }
                }
            }
        }
        consolidated_labels.append(label)

return consolidated_labels

The post-annotation Lambda will often receive batches of task results in the event object. That batch will be the payload object the Lambda should iterate through. What you send back will be an object meeting the API contract (p. 308).

The output of your labeling job

You'll find the output of the job in a folder named after your labeling job in the target S3 bucket you specified. It will be in a subfolder named manifests.

For a bounding box task, the output you find in the output manifest will look a bit like the demo below. The example has been cleaned up for printing. The actual output will be a single line per record.

Example : JSON in your output manifest

```json
{
"source-ref": "<URL>",
"<label attribute name>": {
    "workerId": "<URL>",
    "imageSource": "<image URL>",
    "boxesInfo": "{\"boundingBox\":{\"boundingBoxes\":{\"height\":878, \"label\":\"bird \", \"left\":208, \"top\":6, \"width\":809}}, \"inputImageProperties\":{\"height\":924, \"width\":1280}}\"},
"<label attribute name>-metadata": {
    "type": "groundTruth/custom",
    "job_name": "<Labeling job name>",
    "human-annotated": "yes"
},
"animal": "bird"
}
```

Note how the additional animal attribute from your original manifest is passed to the output manifest on the same level as the source-ref and labeling data. Any properties from your input manifest, whether they were used in your template or not, will be passed to the output manifest.

This should help you create your own custom template.
**Demo Template: Labeling Intents with crowd-classifier**

If you choose a custom template, you'll reach the Custom labeling task panel. There you can select from multiple starter templates that represent some of the more common tasks. The templates provide a starting point to work from in building your customized labeling task's template.

In this demonstration, you work with the Intent Detection template, which uses the crowd-classifier (p. 462) element, and the AWS Lambda functions needed for processing your data before and after the task.

**Topics**
- Starter Intent Detection custom template (p. 302)
- Your Intent Detection custom template (p. 302)
- Your pre-annotation Lambda function (p. 306)
- Your post-annotation Lambda function (p. 306)
- Your labeling job output (p. 307)

**Starter Intent Detection custom template**

This is the intent detection template that is provided as a starting point.

```html
<script src="https://assets.crowd.aws/crowd-html-elements.js"></script>
<crowd-form>
  <crowd-classifier
    name="intent"
    categories="{{ task.input.labels | to_json | escape }}"
    header="Pick the most relevant intention expressed by the below text"
  >
    <classification-target>
      {{ task.input.utterance }}
    </classification-target>

    <full-instructions header="Intent Detection Instructions">
      <p>Select the most relevant intention expressed by the text.</p>
      <div>
        <p><strong>Example:</strong> I would like to return a pair of shoes</p>
        <p><strong>Intent:</strong> Return</p>
      </div>
    </full-instructions>

    <short-instructions>
      Pick the most relevant intention expressed by the text
    </short-instructions>
  </crowd-classifier>
</crowd-form>
```

The custom templates use the Liquid template language, and each of the items between double curly braces is a variable. The pre-annotation AWS Lambda function should provide an object named taskInput and that object's properties can be accessed as `{{ task.input.<property name> }}` in your template.

**Your Intent Detection custom template**

In the starter template, there are two variables: the task.input.labels property in the crowd-classifier element opening tag and the task.input.utterance in the classification-target region's content.
Unless you need to offer different sets of labels with different utterances, avoiding a variable and just using text will save processing time and creates less possibility of error. The template used in this demonstration will remove that variable, but variables and filters like to_json are explained in more detail in the crowd-bounding-box demonstration article.

Styling Your Elements

Two parts of these custom elements that sometimes get overlooked are the <full-instructions> and <short-instructions> regions. Good instructions generate good results.

In the elements that include these regions, the <short-instructions> appear automatically in the "Instructions" pane on the left of the worker's screen. The <full-instructions> are linked from the "View full instructions" link near the top of that pane. Clicking the link opens a modal pane with more detailed instructions.

You can not only use HTML, CSS, and JavaScript in these sections, you are encouraged to if you believe you can provide a strong set of instructions and examples that will help workers complete your tasks with better speed and accuracy.

Example Try out a sample with JSFiddle

Try out an example <crowd-classifier> task. The example is rendered by JSFiddle, therefore all the template variables are replaced with hard-coded values. Click the "View full instructions" link to see a set of examples with extended CSS styling. You can fork the project to experiment with your own changes to the CSS, adding sample images, or adding extended JavaScript functionality.

Example : Final Customized Intent Detection Template

This uses the example <crowd-classifier> task, but with a variable for the <classification-target>. If you are trying to keep a consistent CSS design among a series of different labeling jobs, you can include an external stylesheet using a <link rel...> element the same way you'd do in any other HTML document.

<script src="https://assets.crowd.aws/crowd-html-elements.js"></script>
<crowd-form>
  <crowd-classifier
    name="intent"
    categories="["buy", 'eat', 'watch', 'browse', 'leave"]"
    header="Pick the most relevant intent expressed by the text below"
  
  <classification-target>
    {{ task.input.source }}
  </classification-target>

  <full-instructions header="Emotion Classification Instructions">
    <p>In the statements and questions provided in this exercise, what category of action is the speaker interested in doing?</p>
    <table>
      <tr>
        <th>Example Utterance</th>
        <th>Good Choice</th>
      </tr>
      <tr>
        <td>When is the Seahawks game on?</td>
        <td>
          eat<br>
          <greenbg>watch</greenbg>
          <botchoice>browse</botchoice>
        </td>
      </tr>
      <tr>
        <td>Example Utterance</td>
        <th>Bad Choice</th>
      </tr>
      <tr>
        <td>When is the Seahawks game on?</td>
        <td>
          buy<br>
          <greenbg>eat</greenbg>
          <botchoice>watch</botchoice>
        </td>
      </tr>
    </table>
  </full-instructions>

  <short-instructions>
    What is the speaker expressing they would like to do next?
  </short-instructions>
</crowd-classifier>
</crowd-form>

<greenbg {
  background: #feee23;
  display: block;
}

table {
  border-collapse: collapse; /* IE7 and lower */
  border-spacing: 0;
}

th, tfoot, .fakehead {
  background-color: #8888ee;
  color: #f3f3f3;
  font-weight: 700;
}

th, td, tfoot {
  border: 1px solid blue;
Example: Your manifest file

If you are preparing your manifest file manually for a text-classification task like this, have your data formatted in the following manner.

```json
{"source": "Roses are red"
{"source": "Violets are Blue"
{"source": "Ground Truth is the best"}
```
This differs from the manifest file used for the "Demo Template: Annotation of Images with crowd-bounding-box (p. 297)" demonstration in that `source-ref` was used as the property name instead of `source`. The use of `source-ref` designates S3 URIs for images or other files that must be converted to HTTP. Otherwise, `source` should be used like it is with the text strings above.

**Your pre-annotation Lambda function**

As part of the job set-up, provide the ARN of an AWS Lambda that can be called to process your manifest entries and pass them to the template engine.

This Lambda function is required to have one of the following four strings as part of the function name: SageMaker, Sagemaker, sagemaker, or LabelingFunction.

This applies to both your pre-annotation and post-annotation Lambdas.

When you're using the console, if you have Lambdas that are owned by your account, a drop-down list of functions meeting the naming requirements will be provided to choose one.

In this very basic sample, where you have only one variable, it's primarily a pass-through function. Here's a sample pre-labeling Lambda using Python 3.7.

```python
import json

def lambda_handler(event, context):
    return {
        "taskInput": event['dataObject']
    }
```

The `dataObject` property of the `event` contains the properties from a data object in your manifest.

In this demonstration, which is a simple pass through, you just pass that straight through as the `taskInput` value. If you add properties with those values to the `event['dataObject']` object, they will be available to your HTML template as Liquid variables with the format `{{ task.input.<property name> }}`.

**Your post-annotation Lambda function**

As part of the job set up, provide the ARN of an Lambda function that can be called to process the form data when a worker completes a task. This can be as simple or complex as you want. If you want to do answer-consolidation and scoring as data comes in, you can apply the scoring or consolidation algorithms of your choice. If you want to store the raw data for offline processing, that is an option.

Set permissions for your post-annotation Lambda function

The annotation data will be in a file designated by the `s3Uri` string in the `payload` object. To process the annotations as they come in, even for a simple pass through function, you need to assign S3ReadOnly access to your Lambda so it can read the annotation files.

In the Console page for creating your Lambda, scroll to the **Execution role** panel. Select **Create a new role from one or more templates**. Give the role a name. From the **Policy templates** drop-down, choose **Amazon S3 object read-only permissions**. Save the Lambda and the role will be saved and selected.

The following sample is for Python 3.7.

```python
import json
import boto3
from urllib.parse import urlparse

def lambda_handler(event, context):
    consolidated_labels = []
```
parsed_url = urlparse(event['payload']['s3Uri']);
s3 = boto3.client('s3')
textFile = s3.get_object(Bucket = parsed_url.netloc, Key = parsed_url.path[1:])["Body"].read()
annotations = json.loads(textFile)

for dataset in annotations:
    for annotation in dataset['annotations']:
        new_annotation = json.loads(annotation['annotationData']['content'])
        label = {
            'datasetObjectId': dataset['datasetObjectId'],
            'consolidatedAnnotation': {
                'content': {
                    event['labelAttributeName']: {
                        'workerId': annotation['workerId'],
                        'result': new_annotation,
                        'labeledContent': dataset['dataObject']
                    }
                }
            }
        }
        consolidated_labels.append(label)

return consolidated_labels

Your labeling job output

The post-annotation Lambda will often receive batches of task results in the event object. That batch will be the payload object the Lambda should iterate through.

You'll find the output of the job in a folder named after your labeling job in the target S3 bucket you specified. It will be in a subfolder named manifests.

For an intent detection task, the output in the output manifest will look a bit like the demo below. The example has been cleaned up and spaced out to be easier for humans to read. The actual output will be more compressed for machine reading.

Example : JSON in your output manifest

```json
[
    {
        "datasetObjectId":"<Number representing item's place in the manifest>",
        "consolidatedAnnotation": {
            "content": {
                "<name of labeling job>": {
                    "workerId":"private.us-east-1.XXXXXXXXXXXXXXXXXXXX",
                    "result": {
                        "intent": {
                            "label":"<label chosen by worker>"
                        },
                        "labeledContent": {
                            "content":"<text content that was labeled>"
                        }
                    }
                }
            }
        }
    }
]```
Step 3: Processing with AWS Lambda

In this step, you set which AWS Lambda functions to trigger on each dataset object prior to sending it to workers and which function will be used to process the results once the task is submitted. These functions are required.

You will first need to visit the AWS Lambda console or use AWS Lambda's APIs to create your functions. The AmazonSageMakerFullAccess policy is restricted to invoking AWS Lambda functions with one of the following four strings as part of the function name: SageMaker, Sagemaker, sagemaker, or LabelingFunction. This applies to both your pre-annotation and post-annotation Lambdas. If you choose to use names without those strings, you must explicitly provide lambda:InvokeFunction permission to the IAM role used for creating the labeling job.

Select your lambdas from the Lambda functions section that comes after the code editor for your custom HTML in the Ground Truth console.

If you need an example, there is an end-to-end demo, including Python code for the Lambdas, in the "Demo Template: Annotation of Images with crowd-bounding-box (p. 297)" document.

Pre-annotation Lambda

Before a labeling task is sent to the worker, your AWS Lambda function will be sent a JSON formatted request to provide details.

Example of a Pre-annotation request

```json
{
   "version": "2018-10-16",
   "labelingJobArn": <labelingJobArn>
   "dataObject" : {
       "source-ref": "s3://mybucket/myimage.png"
   }
}
```
The `dataObject` will contain the JSON formatted properties from your manifest's data object. For a very basic image annotation job, it might just be a `source-ref` property specifying the image to be annotated. The JSON line objects in your manifest can be up to 100 kilobytes in size and contain a variety of data.

In return, Ground Truth will require a response formatted like this:

**Example of expected return data**

```json
{
    "taskInput": <json object>,
    "isHumanAnnotationRequired": <boolean> # Optional
}
```

In the previous example, the `<json object>` needs to contain all the data your custom form will need. If you're doing a bounding box task where the instructions stay the same all the time, it may just be the HTTP(S) or S3 resource for your image file. If it's a sentiment analysis task and different objects may have different choices, it would be the object reference as a string and the choices as an array of strings.

**Implications of `isHumanAnnotationRequired`**

This value is optional because it will default to `true`. The primary use case for explicitly setting it is when you want to exclude this data object from being labeled by human workers.

If you have a mix of objects in your manifest, with some requiring human annotation and some not needing it, you can include a `isHumanAnnotationRequired` value in each data object. You can then use code in your pre-annotation Lambda to read the value from the data object and set the value in your Lambda output.

**The pre-annotation Lambda runs first**

Before any tasks are available to workers, your entire manifest will be processed into an intermediate form, using your Lambda. This means you won't be able to change your Lambda part of the way through a labeling job and see that have an impact on the remaining tasks.

**Post-annotation Lambda**

When all workers have annotated the data object or when `TaskAvailabilityLifetimeInSeconds` has been reached, whichever comes first, Ground Truth will send those annotations to your Post-annotation Lambda. This Lambda is generally used for Consolidate Annotations (p. 405). The request object will come in like this:

**Example of a post-labeling task request**

```json
{
    "version": "2018-10-16",
    "labelingJobArn": <labelingJobArn>,
    "labelCategories": [<string>],
    "labelAttributeName": <string>,
    "roleArn": "string",
    "payload": {
        "$3Uri": <string>
    }
}
```

**Note**

If no worker work on the data object and `TaskAvailabilityLifetimeInSeconds` has been reached, data object will be marked as failed and not included as part of post annotation lambda invocation.
Post-labeling task Lambda permissions

The actual annotation data will be in a file designated by the `s3Uri` string in the `payload` object. To process the annotations as they come in, even for a simple pass through function, you need to assign the necessary permissions to your Lambda to read files from your S3 bucket.

In the Console page for creating your Lambda, scroll to the **Execution role** panel. Select **Create a new role from one or more templates**. Give the role a name. From the **Policy templates** drop-down, choose **Amazon S3 object read-only permissions**. Save the Lambda and the role will be saved and selected.

Example of an annotation data file

```
[
  {
    "datasetObjectId": <string>,
    "dataObject": {
      "s3Uri": <string>,
      "content": <string>
    },
    "annotations": [{
      "workerId": <string>,
      "annotationData": {
        "content": <string>,
        "s3Uri": <string>
      }
    }]
  }
]
```

Essentially, all the fields from your form will be in the `content` object. At this point you can start running data consolidation algorithms on the data, using an AWS database service to store results. Or you can pass some processed/optimized results back to Ground Truth for storage in your consolidated annotation manifests in the S3 bucket you specify for output during the configuration of the labeling job.

In return, Ground Truth will require a response formatted like this:

Example of expected return data

```
[
  {
    "datasetObjectId": <string>,
    "consolidatedAnnotation": {
      "content": {
        "<labelattributename>": {
          # ... label content
        }
      }
    }
  },
  {
    "datasetObjectId": <string>,
    "consolidatedAnnotation": {
      "content": {
        "<labelattributename>": {
          # ... label content
        }
      }
    }
  }
]
```
At this point, all the data you're sending to your S3 bucket, other than the `datasetObjectId` will be in the `content` object.

That will result in an entry in your job's consolidation manifest like this:

**Example of label format in output manifest**

```json
{
    "source": "<s3uri or content>",
    "labelAttributeName": {
        # ... label content from you
    },
    "labelAttributeName-metadata": {
        # This will be added by Ground Truth
        "job_name": <labelingJobName>,
        "type": "groundTruth/custom",
        "human-annotated": "yes",
        "creation_date": <date> # Timestamp of when received from Post-labeling Lambda
    }
}
```

Because of the potentially complex nature of a custom template and the data it collects, Ground Truth does not offer further processing of the data or insights into it.

**Next**

Custom Workflows via the API (p. 311)

**Custom Workflows via the API**

When you have created your custom UI template (Step 2) and processing Lambda functions (Step 3), you should place the template in an Amazon S3 bucket with a file name format of: `<FileName>.liquid.html`.

Use the `CreateLabelingJob` action to configure your task. You'll use the location of a custom template (Step 2: Creating your custom labeling task template (p. 292)) stored in a `<filename>.liquid.html` file on S3 as the value for the `UiTemplateS3Uri` field in the `UiConfig` object within the `HumanTaskConfig` object.

For the AWS Lambda tasks described in Step 3: Processing with AWS Lambda (p. 308), the post-annotation task's ARN will be used as the value for the `AnnotationConsolidationLambdaArn` field, and the pre-annotation task will be used as the value for the `PreHumanTaskLambdaArn` field.

**Create a Labeling Job**

You can create a labeling job in the Amazon SageMaker console and by using an AWS SDK in your preferred language to run `CreateLabelingJob`. After a labeling job has been created, you can track worker metrics (for private workforces) and your labeling job status using CloudWatch.

After you have chosen your task type, use the topics on this page to learn how to create a labeling job. If you are a new Ground Truth user, we recommend that you start by walking through the demo in Getting started (p. 162).

**Important**

Ground Truth requires all S3 buckets that contain labeling job input image data have a CORS policy attached. To learn more about this change, see CORS Permission Requirement (p. 415).

**Topics**

- Built-in Task Types (p. 312)
Built-in Task Types

Amazon SageMaker Ground Truth has several built-in task types. Ground Truth provides a worker task template for built-in task types. Additionally, some built in task types support Automate Data Labeling (p. 406). The following topics describe each built-in task type and demo the worker task templates that are provided by Ground Truth in the console. To learn how to create a labeling job in the console using one of these task types, select the task type page.

<table>
<thead>
<tr>
<th>Label Images</th>
<th>Label Text</th>
<th>Label Videos and Video Frames</th>
<th>Label 3D Point Clouds</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bounding Box (p. 166)</td>
<td>Named Entity Recognition (p. 185)</td>
<td>Video Classification (p. 195)</td>
<td>3D Point Cloud Object Detection (p. 229)</td>
</tr>
<tr>
<td>Image Classification (Single Label) (p. 178)</td>
<td>Text Classification (Single Label) (p. 188)</td>
<td>Video Frame Object Detection (p. 199)</td>
<td>3D Point Cloud Object Tracking (p. 236)</td>
</tr>
<tr>
<td>Image Classification (Multi-label) (p. 180)</td>
<td>Text Classification (Multi-label) (p. 191)</td>
<td>Video Frame Object Tracking (p. 204)</td>
<td>3D Point Cloud Semantic Segmentation (p. 246)</td>
</tr>
<tr>
<td>Image Semantic Segmentation (p. 171)</td>
<td>Verify and Adjust Labels (p. 287)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note

Each of the video frame and 3D point cloud task types has an adjustment task type that you use to verify and adjust labels from a previous labeling job. Select a video frame or 3D point cloud task type page above to learn how to adjust labels created using that task type.

Creating Instruction Pages

Create custom instructions for labeling jobs to improve your worker's accuracy in completing their task. You can modify the default instructions that are provided in the console or you can create your own. The instructions are shown to the worker on the page where they complete their labeling task.

There are two kinds of instructions:

- Short instructions—instructions that are shown on the same webpage where the worker completes their task. These instructions should provide an easy reference to show the worker the correct way to label an object.
- Full instructions—instructions that are shown on a dialog box that overlays the page where the worker completes their task. We recommend that you provide detailed instructions for completing the task with multiple examples showing edge cases and other difficult situations for labeling objects.

Create instructions in the console when you are creating your labeling job. Start with the existing instructions for the task and use the editor to modify them to suit your labeling job.
Note
Once you create your labeling job, it will automatically start and you will not be able to modify your worker instructions. If you need to change your worker instructions, stop the labeling job that you created, clone it, and modify your worker instructions before creating a new job. You can clone a labeling job in the console by selecting the labeling job and then selecting Clone in the Actions menu.
To clone a labeling job using the Amazon SageMaker API or your preferred Amazon SageMaker SDK, make a new request to the CreateLabelingJob operation with the same specifications as your original job after modifying your worker instructions.

Short Instructions

Short instructions appear on the same web page that workers use to label your data object. For example, the following is the editing page for a bounding box task. The short instructions panel is on the left.

Keep in mind that a worker will only spend seconds looking at the short instructions. Workers must be able to scan and understand your information quickly. In all cases it should take less time to understand the instructions than it takes to complete the task. Keep these points in mind:

- Your instructions should be clear and simple.
- Pictures are better than words. Create a simple illustration of your task that your workers can immediately understand.
- If you must use words, use short, concise examples.
Your short instructions are more important than your full instructions.

The Amazon SageMaker Ground Truth console provides an editor so that you can create your short instructions. Replace the placeholder text and images with instructions for your task. Preview the worker's task page by choosing Preview. The preview will open in a new window, be sure to turn off pop-up blocking so that the window will show.

Full Instructions

You can provide additional instructions for your workers in a dialog box that overlays the page where workers label your data objects. Use full instructions to explain more complex tasks and to show workers the proper way to label edge cases or other difficult objects.

You can create full instructions using an editor in the Ground Truth console. As with quick instructions, keep the following in mind:

- Workers will want detailed instruction the first few times that the complete your task. Any information that they must have should be in the quick instructions.
- Pictures are more important than words.
- Text should be concise.
- Full instructions should supplement the short instructions. Don't repeat information that appears in the short instructions.

The Ground Truth console provides an editor so that you can create your full instructions. Replace the placeholder text and images with instructions for your task. Preview the full instruction page by choosing Preview. The preview will open in a new window, be sure to turn off pop-up blocking so that the window will show.

Add example images to your instructions

Images provide useful examples for your workers. To add a publicly accessible image to your instructions:

- Place the cursor where the image should go in the instructions editor.
- Click the image icon in the editor toolbar.
- Enter the URL of your image.

If your instruction image in Amazon S3 is not publicly accessible:

- As the image URL, enter: {{ 'https://s3.amazonaws.com/your-bucket-name/image-file-name' | grant_read_access }}.
- This renders the image URL with a short-lived, one-time access code appended so the worker's browser can display it. A broken image icon is displayed in the instructions editor, but previewing the tool displays the image in the rendered preview.

Create a Labeling Job (Console)

You can use the Amazon SageMaker console to create a labeling job for all of the Ground Truth built-in task types and custom labeling workflows. For built-in task types, we recommend that you use this page with the page for your task type. Each task type page includes specific details on creating a labeling job using that task type.

You need to provide the following to create a labeling job in the SageMaker console:
• An input manifest file in Amazon S3. You can place your input dataset in Amazon S3 and automatically generate a manifest file using the Ground Truth console (not supported for 3D point cloud labeling jobs).

Alternatively, you can manually create an input manifest file. To learn how, see Input Data (p. 341).

• An Amazon S3 bucket to store your output data.

• An IAM role with permission to access your resources in Amazon S3 and with a SageMaker execution policy attached. For a general solution, you can attach the managed policy, AmazonSageMakerFullAccess, to an IAM role and include sagemaker in your bucket name.

For more granular policies, see the section called "IAM Permissions" (p. 416).

Note that 3D point cloud task types have additional security considerations. Learn more.

• A work team. You create a work team from a workforce made up of Amazon Mechanical Turk workers, vendors, or your own private workers. Learn more.

Note: You cannot use the Mechanical Turk workforce for 3D point cloud or video frame labeling jobs.

• If you are using a custom labeling workflow, you must save a worker task template in Amazon S3 and provide an Amazon S3 URI for that template. For more information, see Step 2: Creating your custom labeling task template (p. 292).

• (Optional) An AWS KMS key ARN if you want SageMaker to encrypt the output of your labeling job using your own AWS KMS encryption key instead of the default Amazon S3 service key.

• (Optional) Existing labels for the dataset you use for your labeling job. Use this option if you want workers to adjust, or approve and reject labels.

Important
Your work team, input manifest file, output bucket, and other resources in Amazon S3 must be in the same AWS Region you use to create your labeling job.

When you create a labeling job using the SageMaker console, you add worker instructions and labels to the worker UI that Ground Truth provides. You can preview and interact with the worker UI while creating your labeling job in the console. You can also see a preview of the worker UI on your built-in task type page.

To create a labeling job (console)

2. In the left navigation pane, choose Labeling jobs.
4. For Job name, enter a name for your labeling job.
5. (Optional) If you want to identify your labels with a key, select I want to specify a label attribute name different from the labeling job name. If you do not select this option, the labeling job name you specified in the previous step will be used to identify your labels in your output manifest file.
6. Choose a data setup to setup to set up a connection between your input dataset and Ground Truth.

   • For Automated data setup:
     • Follow the instructions in Automated Data Setup (p. 342) for image, text, and video clip labeling jobs.
     • Follow the instructions in Automated Video Frame Input Data Setup (p. 378) for video frame labeling jobs.

   • For Manual data setup:
     • For Input dataset location, provide the location in Amazon S3 in which your input manifest file is located. For example, if your input manifest file, manifest.json, is located in example-bucket, enter s3://example-bucket/manifest.json.
• For **Output dataset location**, provide the location in Amazon S3 where you want Ground Truth to store the output data from your labeling job.

7. For **IAM Role**, choose an existing IAM role or create an IAM role with permission to access your resources in Amazon S3, to write to the output Amazon S3 bucket specified above, and with a SageMaker execution policy attached.

8. (Optional) For **Additional configuration**, you can specify how much of your dataset you want workers to label, and if you want SageMaker to encrypt the output data for your labeling job using an AWS KMS encryption key. To encrypt your output data, you must have the required AWS KMS permissions attached to the IAM role you provided in the previous step. For more details, see the section called “IAM Permissions” (p. 416).

9. In the **Task type** section, under **Task category**, use the dropdown list to select your task category.

10. In **Task selection**, choose your task type.

11. (Optional) Provide tags for your labeling job to make it easier to find in the console later.

12. Choose **Next**.

13. In the **Workers** section, choose the type of workforce you would like to use. For more details about your workforce options see Create and Manage Workforces (p. 427).

14. (Optional) After you've selected your workforce, specify the **Task timeout**. This is the maximum amount of time a worker has to work on a task.

   For 3D point cloud annotation tasks, the default task timeout is 3 days. The default timeout for text and image classification and label verification labeling jobs is 5 minutes. The default timeout for all other labeling jobs is 60 minutes.

   If you set your task timeout to be greater than one hour, you must **increase the max session duration** of your execution role to be greater than or equal to the task timeout. If you choose a task time that is greater than 8 hours for 3D point cloud labeling jobs, refer to **Increase MaxSessionDuration for Execution Role** (p. 257).

15. (Optional) For bounding box, semantic segmentation, and point cloud task types, you can select **Display existing labels** if you want to display labels for your input data set for workers to verify or adjust. If you choose this option, select the label attribute name for the labels that you want to verify or adjust. This can be found in the output manifest file in your Amazon S3 bucket. For more information, see **Verify and Adjust Labels** (p. 287).

16. In the next section, specify your worker instructions and labels. You can select **See preview** to preview your worker instructions, labels, and interact with the worker UI.

   For the point cloud and video frame labeling task types, you can also specify label category and frame attributes. To learn more, see **Worker User Interface (UI)** (p. 255) and **Worker User Interface (UI)** (p. 210) respectively.

   For more details about the worker UI for each task type, see the page for your **built-in task type**.

17. (Optional) If you are not using a point cloud task type, you can add **Additional instructions** to help your worker complete your task.

18. Choose **Create**.

After you've successfully created your labeling job, you are redirected to the **Labeling jobs** page. The status of the labeling job you just created is **in progress**. This status progressively updates as workers complete your tasks. When all tasks are successfully completed, the status changes to **Completed**.

If an issue occurs while creating the labeling job, its status changes to **Failed**.

To view more details about the job, choose the labeling job name.
Next Steps

After your labeling job status changes to **Completed**, you can view your output data in the Amazon S3 bucket that you specified while creating that labeling job. For details about the format of your output data, see [Output Data](#).

**Create a Labeling Job (API)**

To create a labeling job using the Amazon SageMaker API, you use the `CreateLabelingJob` operation. For specific instructions on creating a labeling job for a built-in task type, see that task type page. To learn how to create a streaming labeling job, which is a a labeling job that runs perpetually, see Create a Streaming Labeling Job (p. 321).

To use the `CreateLabelingJob` operation, you need the following:

- A worker task template (`UiTemplateS3Uri`) or human task UI ARN (`HumanTaskUiArn`) in Amazon S3.
- For 3D point cloud labeling jobs, use the ARN listed in `HumanTaskUiArn` for your task type.
- If you are using a built-in task type other than 3D point cloud tasks, you can add your worker instructions to one of the pre-built templates and save the template (using a .html or .liquid extension) in your S3 bucket. Find the pre-build templates on your task type page.
- If you are using a custom labeling workflow, you can create a custom template and save the template in your S3 bucket. To learn how to build a custom worker template, see Step 2: Creating your custom labeling task template (p. 292). For custom HTML elements that you can use to customize your template, see Crowd HTML Elements Reference (p. 448). For a repository of demo templates for a variety of labeling tasks, see Amazon SageMaker Ground Truth Sample Task UIs.
- At least one S3 bucket to store your input and output data.
- An input manifest file that specifies your input data. For information about creating an input manifest, see Input Data (p. 341).
- A label category configuration file. Each label category name must be unique.

For image classification and text classification (single and multi-label) you must specify at least two label categories. For all other task types, the minimum number of label categories required is one.

For 3D point cloud and video frame task type, use the format in Create a Labeling Category Configuration File with Label Category and Frame Attributes (p. 326).

For all other built-in task types and custom tasks, your label category configuration file must be a JSON file in the following format. Identify the labels you want to use by replacing `label_1`, `label_2`, ..., `label_n` with your label categories.

```json
{
    "document-version": "2018-11-28"
    "labels": [
        {"label": "label_1"},
        {"label": "label_2"},
        ...
        {"label": "label_n"}
    ]
}
```

- An AWS Identity and Access Management (IAM) role with the `AmazonSageMakerGroundTruthExecution` managed IAM policy attached and with permissions to access your S3 buckets. To learn more about this policy, see Grant General Permissions To Get Started Using Ground Truth (p. 417). If you require more granular permissions, see the section called “IAM Permissions” (p. 416).
Create a Labeling Job

If your input or output bucket name does not contain `sagemaker`, you can attach a policy similar to the following to the role that is passed to the `CreateLabelingJob` operation.

```json
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": ["s3:GetObject"],
            "Resource": [
                "arn:aws:s3:::my_input_bucket/*"
            ]
        },
        {
            "Effect": "Allow",
            "Action": ["s3:PutObject"],
            "Resource": [
                "arn:aws:s3:::my_output_bucket/*"
            ]
        }
    ]
}
```

- A pre-annotation and post-annotation (or annotation-consolidation) AWS Lambda function Amazon Resource Name (ARN) to process your input and output data.
- Lambda functions are predefined in each AWS Region for built-in task types. To find the pre-annotation Lambda ARN for your Region, see `PreHumanTaskLambdaArn`. To find the annotation-consolidation Lambda ARN for your Region, see `AnnotationConsolidationLambdaArn`.
- For custom labeling workflows, you must provide a custom pre- and post-annotation Lambda ARN. To learn how to create these Lambda functions, see Step 3: Processing with AWS Lambda (p. 308).
- A work team ARN. To learn more about work teams and workforces, see Create and Manage Workforces (p. 427).

If you use the Amazon Mechanical Turk workforce, use the `ContentClassifiers` parameter in `CreateLabelingJob` to declare that your content is free of personally identifiable information or adult content. If your content contains personally identifiable information or adult content, SageMaker might restrict the Amazon Mechanical Turk workers that can view your task.

If you are creating a labeling job for a point cloud task type, you cannot use the Amazon Mechanical Turk workforce.

- (Optional) For some task types, you can have multiple workers label a single data object by inputting a number greater than one for the `NumberOfHumanWorkersPerDataObject` parameter. For more information about annotation consolidation, see Consolidate Annotations (p. 405).

The following is an example of an AWS Python SDK (Boto3) request to create a labeling job for a built-in task type in the US East (N. Virginia) Region.

**AWS SDK for Python (Boto3)**

The following is an example of an AWS Python SDK (Boto3) request to create a labeling job for a built-in task type in the US East (N. Virginia) Region using a private workforce. Replace all *red-italicized text* with your labeling job resources and specifications.

```python
response = client.create_labeling_job(
```

318
Create a Labeling Job

```python
LabelingJobName="example-labeling-job",
LabelAttributeName="label",
InputConfig={
  'DataSource': {
    'S3DataSource': {
      'ManifestS3Uri': "s3://bucket/path/manifest-with-input-data.json"
    }
  },
  'DataAttributes': {
    'ContentClassifiers': [
      "FreeOfPersonallyIdentifiableInformation","FreeOfAdultContent",
    ]
  }
},
OutputConfig={
  'S3OutputPath': "s3://bucket/path/file-to-store-output-data",
  'KmsKeyId': "string"
},
RoleArn="arn:aws:iam::*:role/*",
LabelCategoryConfigS3Uri="s3://bucket/path/label-categories.json",
StoppingConditions={
  'MaxHumanLabeledObjectCount': 123,
  'MaxPercentageOfInputDatasetLabeled': 123
},
HumanTaskConfig={
  'WorkteamArn': "arn:aws:sagemaker:region::*:workteam/private-crowd/*",
  'UiConfig': {
    'UiTemplateS3Uri': "s3://bucket/path/custom-worker-task-template.html"
  },
  'TaskKeywords': ['Images', 'Classification', 'Multi-label'],
  'TaskTitle': "Multi-label image classification task",
  'TaskDescription': "Select all labels that apply to the images shown",
  'NumberOfHumanWorkersPerDataObject': 1,
  'TaskTimeLimitInSeconds': 3600,
  'TaskAvailabilityLifetimeInSeconds': 21600,
  'MaxConcurrentTaskCount': 1000,
  'AnnotationConsolidationConfig': {
  },
  Tags=[
    {
      'Key': "string",
      'Value': "string"
    }
  ]
},
```

AWS CLI

The following is an example of an AWS CLI request to create a labeling job for a built-in task type in the US East (N. Virginia) Region using the Amazon Mechanical Turk workforce. For more information, see start-human-loop in the AWS CLI Command Reference. Replace all red-italized text with your labeling job resources and specifications.

```
# aws --region us-east-1 sagemaker create-labeling-job \
--labeling-job-name "example-labeling-job" \
--label-attribute-name "label"
```

319
--role-arn "arn:aws:iam::account-id:role/role-name" \
--input-config '{
  "DataAttributes": {
    "ContentClassifiers": [
      "FreeOfPersonallyIdentifiableInformation",
      "FreeOfAdultContent"
    ],
  },
  "DataSource": {
    "S3DataSource": {
      "ManifestS3Uri": "s3://bucket/path/manifest-with-input-data.json"
    }
  }
}',
--output-config '{
  "KmsKeyId": "",
  "S3OutputPath": "s3://bucket/path/file-to-store-output-data"
}',
--human-task-config '{
  "AnnotationConsolidationConfig": {
    "AnnotationConsolidationLambdaArn": "arn:aws:lambda:us-east-1:432418664414:function:ACS-"\n  },
  "TaskAvailabilityLifetimeInSeconds": 21600,
  "TaskTimeLimitInSeconds": 3600,
  "NumberOfHumanWorkersPerDataObject": 1,
  "WorkteamArn": "arn:aws:sagemaker:us-east-1:394669845002:workteam/public-crowd/default",
  "PublicWorkforceTaskPrice": {
    "AmountInUsd": {
      "Dollars": 0,
      "TenthFractionsOfACent": 6,
      "Cents": 3
    }
  },
  "TaskDescription": "Select all labels that apply to the images shown",
  "MaxConcurrentTaskCount": 1000,
  "TaskTitle": "Multi-label image classification task",
  "TaskKeywords": [
    "Images",
    "Classification",
    "Multi-label"
  ],
  "UiConfig": {
    "UiTemplateS3Uri": "s3://bucket/path/custom-worker-task-template.html"
  }
}'

**Important**

If you set TaskTimeLimitInSeconds to be greater than one hour (3,600 seconds), you must increases the max session duration of your execution role to be greater than or equal to the task timeout.

You can modify the max session duration of your execution role using the IAM console, AWS CLI, and IAM API. To modify your execution role, go to Modifying a Role in the IAM User Guide, select your preferred method (console, CLI, or API) to modify the role from the Topics list, and then select Modifying a Role Maximum Session Duration to view the instructions. For 3D point cloud task types, refer to Increase MaxSessionDuration for Execution Role (p. 257).

For more information about this operation, see CreateLabelingJob. For information about how to use other language-specific SDKs, see See Also in the CreateLabelingJobs topic.
Create a Streaming Labeling Job

To create a streaming labeling job, you must create two Amazon SNS topics: an input topic and an output topic and specify these topics in CreateLabelingJob parameters InputConfig and OutputConfig respectively using SnsDataSource.

**Important**
If you are a new user of Ground Truth streaming labeling jobs, it is recommended that you review Ground Truth Streaming Labeling Jobs (p. 343) before creating a streaming labeling job.

Use the following sections to create the resources that you need and can use to create a streaming labeling job:

- Learn how to create SNS topics with the permissions required for Ground Truth streaming labeling jobs by following the steps in Create Amazon SNS Input and Output Topics (p. 321). Your SNS topics must be created in the same AWS Region as your labeling job.
- See Subscribe an Endpoint to Your Amazon SNS Output Topic (p. 323) to learn how to set up an endpoint to receive labeling task output data at a specified endpoint each time a labeling task is completed.
- To learn how to configure your Amazon S3 bucket to send notifications to your Amazon SNS input topic, see Set up Amazon S3 Bucket Event Notifications (p. 323).
- Optionally, add data objects that you want to have labeled as soon as the labeling job starts to your input manifest. For more information, see Create a Manifest File (Optional) (p. 324).
- There are other resources required to create a labeling job, such as an IAM role, Amazon S3 bucket, a worker task template and label categories. These are described in the Ground Truth documentation on creating a labeling job. For more information, see Create a Labeling Job (p. 311).

**Important**
When you create a labeling job you must provide an IAM execution role. Attach the AWS managed policy AmazonSageMakerGroundTruthExecution to this role to ensure it has required permissions to execute your labeling job.

When you submit a request to create a streaming labeling job, the state of your labeling job is Initializing. Once the labeling job is active, the state changes to InProgress. Do not send new data objects to your labeling job or attempt to stop your labeling job while it is in the Initializing state. Once the state changes to InProgress, you can start sending new data objects using Amazon SNS and the Amazon S3 configuration.

**Topics**
- Create Amazon SNS Input and Output Topics (p. 321)
- Set up Amazon S3 Bucket Event Notifications (p. 323)
- Create a Manifest File (Optional) (p. 324)
- Example: Use SageMaker API To Create Streaming Labeling Job (p. 324)
- Stop a Streaming Labeling Job (p. 325)

**Create Amazon SNS Input and Output Topics**

You need to create Amazon SNS input and output topics to create a streaming labeling job. Once you create the topics, note down the topic Amazon Resource Name (ARN) of each topic. These ARNs will be the input values for the parameter SnsTopicArn in InputConfig and OutputConfig when you create a labeling job.
Create an Input Topic

Your input topic is used to send new data objects to Ground Truth. To create an input topic, follow the instructions in Creating an Amazon SNS topic in the Amazon Simple Notification Service Developer Guide.

Note down your input topic ARN and use it as input for the CreateLabelingJob parameter SnsTopicArn in InputConfig.

Create an Output Topic

Your output topic is used to send notifications when a data object is labeled. When you create a topic, you have the option to add an encryption key. Use this option to add an AWS Key Management Service customer managed key (CMK) to your topic to encrypt the output data of your labeling job before it is published to your output topic.

To create an output topic, follow the instructions in Creating an Amazon SNS topic in the Amazon Simple Notification Service Developer Guide.

If you add encryption, you must attach additional permission to the topic. See Add Encryption to Your Output Topic (Optional) (p. 322) for more information.

Important
To add a CMK to your output topic while creating a topic in the console, do not use the (Default) alias/aws/sns option. Select a CMK key that you created.

Note down your input topic ARN and use it in your CreateLabelingJob request in the parameter SnsTopicArn in OutputConfig.

Add Encryption to Your Output Topic (Optional)

To encrypt messages published to your output topic, you need to provide an AWS KMS customer managed key (CMK) to your topic. Modify the following policy and add it to your CMK to give Ground Truth permission to encrypt output data before publishing it to your output topic.

Replace <account_id> with the ID of the account that you are using to create your topic. To learn how to find your AWS account ID, see Finding Your AWS Account ID.

```json
{
    "Id": "key-console-policy",
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "Enable IAM User Permissions",
            "Effect": "Allow",
            "Principal": {
                "AWS": "arn:aws:iam::<account_id>:root"
            },
            "Action": "kms:*",
            "Resource": "*"
        },
        {
            "Sid": "Allow access for Key Administrators",
            "Effect": "Allow",
            "Principal": {
                "AWS": "arn:aws:iam::<account_id>:role/Admin"
            },
            "Action": [
                "kms:Create*",
                "kms:Describe*",
                "kms:Enable*",
                "kms:List*",
                "kms:Put*",
                "kms:Get*",
                "kms:Tag*",
                "kms:Untag*",
                "kms:EnableKeyRotation*",
                "kms:GetKeyRotationStatus*",
                "kms:ScheduleKeyDeletion*",
                "kms:CancelKeyDeletion*",
                "kms:GetKeyPolicy*",
                "kms:SetKeyPolicy*",
                "kms:DescribeKey*",
                "kms:GetEffectiveKeyPolicy*"
            ],
            "Resource": "*"
        }
    ]
}```
Additionally, you must modify and add the following policy to the execution role that you use to create your labeling job (the input value for RoleArn).

Replace `<account_id>` with the ID of the account that you are using to create your topic. Replace `<region>` with the AWS Region you are using to create your labeling job. Replace `<key_id>` with your CMK ID.

```json
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "sid1",
      "Effect": "Allow",
      "Action": [
        "kms:Decrypt",
        "kms:GenerateDataKey"
      ],
      "Resource": "arn:aws:kms:<region>:<account_id>:key/<key_id>"
    }
  ]
}
```

For more information on creating and securing keys, see Creating Keys and Using Key Policies in the AWS Key Management Service Developer Guide.

**Subscribe an Endpoint to Your Amazon SNS Output Topic**

When a worker completes a labeling job task from a Ground Truth streaming labeling job, Ground Truth uses your output topic to publish output data to one or more endpoints that you specify. To receive notifications when a worker finishes a labeling task, you must subscribe an endpoint to your Amazon SNS output topic.

To learn how to add endpoints to your output topic, see Subscribe an Amazon SNS topic in the Amazon Simple Notification Service Developer Guide.

To learn more about the output data format that is published to these endpoints, see Output Data (p. 381).

**Important**

If you do not subscribe an endpoint to your Amazon SNS output topic, you will not receive notifications when new data objects are labeled.

**Set up Amazon S3 Bucket Event Notifications**

You can add an event notification to your Amazon S3 bucket using the Amazon S3 console, API, and language specific AWS SDKs, or the AWS Command Line Interface. Set up this event to send notifications
to the same Amazon SNS input topic that you specify using SnsTopicArn in InputConfig when you create a labeling job. Do not set up event notifications using the same Amazon S3 location that you specified for S3OutputPath in OutputConfig – doing so may result in unwanted data objects being processed by Ground Truth for labeling.

You decide the types of events that you want to send to your Amazon SNS topic. Ground Truth creates a labeling job when you send object creation events.

The event structure sent to your Amazon SNS input topic must be a JSON message formatted using the same structure found in Event message structure.

To see examples of how you can set up an event notification for your Amazon S3 bucket using the Amazon S3 console, AWS SDK for .NET, and AWS SDK for Java, follow this walkthrough, Walkthrough: Configure a bucket for notifications (SNS topic or SQS queue) in the Amazon Simple Storage Service Developer Guide.

Create a Manifest File (Optional)

When you create a streaming labeling job, you have the one time option to add objects (such as images or text) to an input manifest file that you specify in ManifestS3Uri of CreateLabelingJob. When the streaming labeling job starts, these objects are sent to workers or added to the Amazon SQS queue if the total number of objects exceed MaxConcurrentTaskCount. The results are added to the Amazon S3 path that you specify when creating the labeling job periodically as workers complete labeling tasks. Output data is sent to any endpoint that you subscribe to your output topic.

If you want to provide initial objects to be labeled, create a manifest file that identifies these objects and place it in Amazon S3. Specify the S3 URI of this manifest file in ManifestS3Uri within InputConfig.

To learn how to format your manifest file, see Input Data (p. 341). To use the SageMaker console to automatically generate a manifest file (not supported for 3D point cloud task types), see Automated Data Setup (p. 342).

Example: Use SageMaker API To Create Streaming Labeling Job

The following is an example of an AWS Python SDK (Boto3) request that you can use to start a streaming labeling job for a built-in task type in the US East (N. Virginia) Region. For more details about each parameter below see CreateLabelingJob. To learn how you can create a labeling job using this API and associated language specific SDKs, see Create a Labeling Job (API).

In this example, note the following parameters:

- **SnsDataSource** – This parameter appears in InputConfig and OutputConfig and is used to identify your input and output Amazon SNS topics respectively.
- **S3DataSource** – This parameter is optional. Use this parameter if you want to include an input manifest file of data objects that you want labeled as soon as the labeling job starts.
- **StoppingConditions** – This parameter is ignored when you create a streaming labeling job. To learn more about stopping a streaming labeling job, see Stop a Streaming Labeling Job (p. 325).

```python
response = client.create_labeling_job(
    LabelingJobName='example-labeling-job',
    LabelAttributeName='label',
    InputConfig={
        'DataSource': {
            'S3DataSource': {
                'ManifestS3Uri': 's3://bucket/path/manifest-with-input-data.json'
            }
        }
    }
)`
Stop a Streaming Labeling Job

You can manually stop your streaming labeling job using the operation `StopLabelingJob`.

If your labeling job remains idle for over 10 days, it is automatically stopped by Ground Truth. In this context, a labeling job is considered idle if no objects are sent to the Amazon SNS input topic and no objects remain in your Amazon SQS queue, waiting to be labeled. For example, if no data objects are fed to the Amazon SNS input topic and all the objects fed to the labeling job are already labeled, Ground Truth starts a timer. After the timer starts, if no items are received within a 10 day period, the labeling job is stopped.

If this 10 day limit poses a problem for your business use case, please contact AWS Support.

When a labeling job is stopped, its status is STOPPING while Ground Truth cleans up labeling job resources and unsubscribes your Amazon SNS topic from your Amazon SQS queue. The Amazon SQS is not deleted by Ground Truth because this queue may contain unprocessed data objects. You should manually delete the queue if you want to avoid incurring additional charges from Amazon SQS. To learn more, see Amazon SQS pricing.
Create a Labeling Category Configuration File with Label Category and Frame Attributes

When you create a 3D point cloud or video frame labeling job using the Amazon SageMaker API operation `CreateLabelingJob`, you use a label category configuration file to specify your labels and worker instructions. Optionally, you can also provide the following in your label category attribute file:

- You can provide **label category attributes** for video frame and 3D point cloud object tracking and object detection task types. Workers can use one or more attributes to give more information about an object. For example, you may want to use the attribute `occluded` to have workers identify when an object is partially obstructed. You can either specify a label category attribute for a single label using the categoryAttributes parameter, or for all labels using the categoryGlobalAttributes parameter.

- You can provide **frame attributes** for video frame and 3D point cloud object tracking and object detection task types using frameAttributes. When you create a frame attribute, it appears on each frame or point cloud in the worker task. In video frame labeling jobs, these are attributes that workers assign to an entire video frame. For 3D point cloud labeling jobs, these attributes are applied to a single point cloud. Use frame attributes to have workers provide more information about the scene in a specific frame or point cloud.

- For video frame labeling jobs, you use the label category configuration file to specify the task type (bounding box, polyline, polygon, or keypoint) sent to workers.

For workers, specifying values for label category attributes and frame attributes will be optional.

**Important**
You should only provide a label attribute name in auditLabelAttributeName if you are running an audit job to verify or adjust labels. Use this parameter to input the LabelAttributeName used in the labeling job that generated the annotations you want your worker to adjust. When you create a labeling job in the console, if you did not specify a label attribute name, the **Name** of your job is used as the LabelAttributeName.

**Topics**
- Label Category Configuration File Schema (p. 326)
- Example: Label Category Configuration Files for 3D Point Cloud Labeling Jobs (p. 332)
- Example: Label Category Configuration Files for Video Frame Labeling Jobs (p. 336)
- Creating Worker Instructions (p. 340)

**Label Category Configuration File Schema**

The following table lists elements you can and must include in your label category configuration file.

**Note**
The parameter annotationType is only supported for video frame labeling jobs.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Required</th>
<th>Accepted Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>frameAttributes</td>
<td>No</td>
<td>A list of JSON objects.</td>
<td>Use this parameter to create a frame attribute that is applied to all frames or 3D point clouds in your labeling job.</td>
</tr>
<tr>
<td>Required Parameters in each JSON Object:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>name, type, description</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>minimum and maximum are required if type is “number”</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

326
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Required</th>
<th>Accepted Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>categoryGlobalAttributes</td>
<td>Required</td>
<td>A list of JSON objects.</td>
<td>Use this parameter to create label category attributes that are applied to all labels you specify in labels. See the third table in this section for more information.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Required Parameters in each JSON Object:</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>name, type</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>minimum and maximum are required if type is &quot;number&quot;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Optional Parameters in each JSON Object:</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>description, enum, editsAllowed</td>
<td></td>
</tr>
<tr>
<td>labels</td>
<td>Yes</td>
<td>A list of up to 30 JSON objects</td>
<td>Use this parameter to specify your labels, or classes. Add one label for each class.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Required Parameters in each JSON Object:</strong></td>
<td>To add a label category attribute to a label, add categoryAttributes to that label.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>label</td>
<td>Use editsAllowed to specify whether or not a label can be edited in an adjustment labeling job. Set editsAllowed to &quot;none&quot; for verification labeling jobs.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Optional Parameters in each JSON Object:</strong></td>
<td>See the following table for more information.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>categoryAttributes, editsAllowed</td>
<td></td>
</tr>
<tr>
<td>Parameter</td>
<td>Required</td>
<td>Accepted Values</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------------</td>
<td>----------</td>
<td>---------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>annotationType</td>
<td>No</td>
<td>String</td>
<td>Use this to specify the task type for your video frame labeling jobs. For example, for a polygon video frame object detection task, choose Polygon. If you do not specify an annotationType when you create a video frame labeling job, Ground Truth will use BoundingBox by default.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BoundingBox, Polyline, Polygon, Keypoint</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Default: BoundingBox</td>
<td></td>
</tr>
<tr>
<td>instructions</td>
<td>No</td>
<td>A JSON object</td>
<td>Use this parameter to add worker instructions to help your workers complete their tasks. For more information about worker instructions, see Worker Instructions (p. 256). Short instructions must be under 255 characters and long instruction must be under 2,048 characters. For more information, see Creating Worker Instructions (p. 340).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Required Parameters in each JSON Object:</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;shortInstruction&quot;, &quot;fullInstruction&quot;</td>
<td></td>
</tr>
<tr>
<td>auditLabelAttributeName</td>
<td>Required</td>
<td>String</td>
<td>Enter the LabelAttributeName used in the labeling job you want to adjust annotations of. Only use this parameter if you are creating an adjustment job for video frame and 3D point cloud object detection, object tracking, or 3D point cloud semantic segmentation.</td>
</tr>
</tbody>
</table>

The following table describes the parameters that you can and must use to create a list of Labels. Each parameter should be included in a JSON object.
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Required</th>
<th>Accepted Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>label</td>
<td>Yes</td>
<td>String</td>
<td>The name of the label category that is displayed to workers. Each label category name must be unique.</td>
</tr>
<tr>
<td>categoryAttributes</td>
<td>No</td>
<td>A list of JSON objects.</td>
<td>Use this parameter to add label category attributes to specific labels you specify in labels. To add one or more label category attributes to a label, include the categoryAttributes JSON object in the same labels JSON object as that label. See the following table for more information.</td>
</tr>
<tr>
<td>editsAllowed</td>
<td>No</td>
<td>String</td>
<td>Specifies whether or not a label can be edited by workers. For video frame or 3D point cloud adjustment labeling jobs, add this parameter to one or more JSON objects in the labels list to specify whether or not a worker can edit a label. For 3D point cloud and video frame verification labeling jobs, add this parameter with the value &quot;none&quot; to each JSON object in the labels list. This will make all labels uneditable.</td>
</tr>
</tbody>
</table>

The following table describes the parameters that you can and must use to create a frame attributes using frameAttributes and label category attribute using the categoryGlobalAttributes and categoryAttributes parameters.
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Required</th>
<th>Accepted Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Yes</td>
<td>String</td>
<td>Use this parameter to assign a name to your label category or frame attribute. This is the attribute name that workers see. Each label category attribute name in your label category configuration file must be unique. Global label category attributes and label specific label category attributes cannot have the same name.</td>
</tr>
<tr>
<td>type</td>
<td>Yes</td>
<td>String</td>
<td>Use this parameter to define the label category or frame attribute type. Required Values: &quot;string&quot; or &quot;number&quot; If you specify &quot;string&quot; for type and provide an enum value for this attribute, workers will be able to choose from one of the choices you provide. If you specify &quot;string&quot; for type and do not provide an enum value, workers can enter free form text. If you specify number for type, worker can enter a number between the minimum and maximum numbers you specify.</td>
</tr>
<tr>
<td>enum</td>
<td>Yes</td>
<td>List of strings</td>
<td>Use this parameter to define options that workers can choose from for this label category or frame attribute. Workers can choose one value specified in enum. For example, if you specify [&quot;foo&quot;, &quot;buzz&quot;,</td>
</tr>
</tbody>
</table>
Create a Labeling Job

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Required</th>
<th>Accepted Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>description</td>
<td>frameAttributes: Yes categoryAttributes or categoryGlobalAttributes: No</td>
<td>String</td>
<td>Use this parameter to add a description of the label category or frame attribute. You can use this field to give workers more information about the attribute. This field is only required for frame attributes.</td>
</tr>
<tr>
<td>minimum and maximum</td>
<td>Required if attribute type is &quot;number&quot;</td>
<td>Integers</td>
<td>Use these parameters to specify minimum and maximum (inclusive) values workers can enter for numeric label category or frame attributes. You must specify &quot;number&quot; for type to use minimum and maximum.</td>
</tr>
<tr>
<td>editsAllowed</td>
<td>No</td>
<td>String</td>
<td>Specifies whether or not a label category or frame attribute can be edited by workers. For video frame or 3D point cloud adjustment and verification labeling jobs, add this parameter to label category and frame attribute JSON objects to specify whether or not a worker can edit an attribute.</td>
</tr>
</tbody>
</table>

**Label and label category attribute quotas**

You can specify up to 10 label category attributes per class. This 10-attribute quotas includes global label category attributes. For example, if you create four global label category attributes, and then assign three label category attributes to label X, that label will have 4+3=7 label category attributes in total. For all label category and label category attribute limits, refer to the following table.
### Example: Label Category Configuration Files for 3D Point Cloud Labeling Jobs

Select a tab in the following tables to see examples of 3D point cloud label category configuration files for object detection, object tracking, semantic segmentation, adjustment, and verification labeling jobs.

**3D Point Cloud Object Tracking and Object Detection**

The following is an example of a label category configuration file that includes label category attributes for a 3D point cloud object detection or object tracking labeling job. This example includes a two frame attributes, which will be added to all point clouds submitted to the labeling job. The *Car* label will include four label category attributes—X, Y, Z, and the global attribute, W.

```json
{
    "Car": {
        "frameAttributes": [
            "location.x",
            "location.y",
            "location.z",
            "global.attribute"
        ]
    }
}
```
### 3D Point Cloud Semantic Segmentation

The following is an example of a label category configuration file for a 3D point cloud semantic segmentation labeling job.

Label category attributes are not supported for 3D point cloud semantic segmentation task types. Frame attributes are supported. If you provide label category attributes for a semantic segmentation labeling job, they will be ignored.
Select a tab in the following table to see an example of a label category configuration file for 3D point cloud verification or adjustment labeling jobs.

**3D Point Cloud Adjustment**

The following is an example of a label category configuration file for a 3D point cloud object detection or object tracking adjustment labeling job. For 3D point cloud semantic segmentation adjustment labeling jobs, categoryGlobalAttributes and categoryAttributes are not supported.

You must include auditLabelAttributeName to specify the label attribute name of the previous labeling job that you use to create the adjustment labeling job. Optionally, you can use the editsAllowed parameter to specify whether or not a label or frame attribute can be edited.

```json
{
    "documentVersion": "2020-03-01",
    "frameAttributes": [
    {
        "name": "count players",
        "description": "How many players to you see in the scene?",
        "type": "number"
    },
    {
        "name": "select one",
        "description": "describe the scene",
        "type": "string",
        "enum": ["clear","blurry"]
    },
    ],
    "labels": [
    {
        "label": "Car",
    },
    {
        "label": "Pedestrian",
    },
    {
        "label": "Cyclist",
    }
    ],
    "instructions": {"shortInstruction":"Select the appropriate label and paint all objects in the point cloud that it applies to the same color",
    "fullInstruction": "<html markup>"}
}
```
Create a Labeling Job

```
{
  "description": "label-attributes-for-all-labels",
  "type": "string",
  "enum": ["foo", "buzz", "biz"]
}
]
"labels": [
  {
    "label": "Car",
    "editsAllowed": "any",
    "categoryAttributes": [
      {
        "name": "X",
        "description": "enter a number",
        "type": "number"
      },
      {
        "name": "Y",
        "description": "select an option",
        "type": "string",
        "enum": ["y1", "y2"],
        "editsAllowed": "any"
      },
      {
        "name": "Z",
        "description": "submit a free-form response",
        "type": "string",
        "editsAllowed": "none"
      }
    ]
  },
  {
    "label": "Pedestrian",
    "categoryAttributes": [
    ...]
  }
],
"instructions": {"shortInstruction": "Draw a tight Cuboid", "fullInstruction": "<html
markup>"},
// include auditLabelAttributeName for label adjustment jobs
"auditLabelAttributeName": "myPrevJobLabelAttributeName"
}
```

3D Point Cloud Verification

The following is an example of a label category configuration file you may use for a 3D point cloud object detection or object tracking verification labeling job. For a 3D point cloud semantic segmentation verification labeling job, categoryGlobalAttributes and categoryAttributes are not supported.

You must include auditLabelAttributeName to specify the label attribute name of the previous labeling job that you use to create the verification labeling job. Additionally, you must use the editsAllowed parameter to specify that no labels can be edited.

```
{
  "documentVersion": "2020-03-01",
  "frameAttributes": [
    {
      "name": "count players",
      "editsAllowed": "any",
      "description": "How many players do you see in the scene?",
      "type": "number"
    },
    {
      "name": "select one",
      "editsAllowed": "any",
      "description": "Select one of the players in the scene",
      "type": "string"
    }
  ]
}
```
Example: Label Category Configuration Files for Video Frame Labeling Jobs

The annotation tools available to your worker and task type used depends on the value you specify for `annotationType`. For example, if you want workers to use key points to track changes in the pose of specific objects across multiple frames, you would specify `Keypoint` for the `annotationType`. If you do not specify an annotation type, `BoundingBox` will be used by default.

The following is an example of a video frame keypoint label category configuration file with label category attributes. This example includes two frame attributes, which will be added to all frames submitted to the labeling job. The `Car` label will include four label category attributes—`X`, `Y`, `Z`, and the global attribute, `W`.
Select a tab from the following table to see examples of label category configuration files for video frame adjustment and verification labeling jobs.

Video Frame Adjustment

The following is an example of a label category configuration file you may use for a video frame adjustment labeling job.

```json
{
   "documentVersion": "2020-03-01",
   "frameAttributes": [
      {
         "name": "count players",
         "description": "How many players do you see in the scene?",
         "type": "number"
      },
      {
         "name": "select one",
         "description": "Describe the scene",
         "type": "string",
         "enum": ["clear", "blurry"]
      }
   ],
   "categoryGlobalAttributes": [
      {
         "name": "W",
         "description": "label-attributes-for-all-labels",
         "type": "string",
         "enum": ["foo", "buz", "buz2"]
      }
   ],
   "labels": [
      {
         "label": "Car",
         "categoryAttributes": [
            {
               "name": "X",
               "description": "Enter a number",
               "type": "number"
            },
            {
               "name": "Y",
               "description": "Select an option",
               "type": "string",
               "enum": ["y1", "y2"]
            },
            {
               "name": "Z",
               "description": "Submit a free-form response",
               "type": "string"
            }
         ]
      },
      {
         "label": "Pedestrian",
         "categoryAttributes": [...
      }
   ],
   "annotationType": "Keypoint",
   "instructions": {
      "shortInstruction": "Add example short instructions here",
      "fullInstruction": "<html markup>"
   }
}
```
You must include `auditLabelAttributeName` to specify the label attribute name of the previous labeling job that you use to create the verification labeling job. Optionally, you can use the `editsAllowed` parameter to specify whether or not labels, label category attributes, or frame attributes can be edited.

```json
{
  "documentVersion": "2020-03-01",
  "frameAttributes": [
    {
      "name": "count players",
      "editsAllowed": "none",
      "description": "How many players do you see in the scene?",
      "type": "number"
    },
    {
      "name": "select one",
      "description": "Describe the scene",
      "type": "string",
      "enum": ["clear", "blurry"]
    }
  ],
  "categoryGlobalAttributes": [
    {
      "name": "W",
      "editsAllowed": "any",
      "description": "label-attributes-for-all-labels",
      "type": "string",
      "enum": ["foo", "buz", "buz2"]
    }
  ],
  "labels": [
    {
      "label": "Car",
      "editsAllowed": "any",
      "categoryAttributes": [
        {
          "name": "X",
          "description": "enter a number",
          "type": "number",
          "editsAllowed": "any"
        },
        {
          "name": "Y",
          "description": "select an option",
          "type": "string",
          "enum": ["y1", "y2"],
          "editsAllowed": "any"
        },
        {
          "name": "Z",
          "description": "submit a free-form response",
          "type": "string",
          "editsAllowed": "none"
        }
      ]
    },
    {
      "label": "Pedestrian",
      "editsAllowed": "none",
      "categoryAttributes": ["..."
    }
  ],
  "annotationType": "Keypoint",
  "instructions": {
    "shortInstruction": "add example short instructions here",
    "fullInstruction": "<html markup>"}
}
```
/**
   * include auditLabelAttributeName for label adjustment jobs
   *
   * auditLabelAttributeName: "myPrevJobLabelAttributeName"
   */

Video Frame Verification

The following is an example of a label category configuration file for a video frame labeling job.

You must include auditLabelAttributeName to specify the label attribute name of the previous labeling job that you use to create the verification labeling job. Additionally, you must use the editsAllowed parameter to specify that no labels can be edited.

```json
{
   "documentVersion": "2020-03-01",
   "frameAttributes": [
      {
         "name": "count players",
         "editsAllowed": "none",
         "description": "How many players to you see in the scene?",
         "type": "number"
      },
      {
         "name": "select one",
         "editsAllowed": "any",
         "description": "describe the scene",
         "type": "string",
         "enum": ["clear", "blurry"]
      }
   ],
   "categoryGlobalAttributes": [
      {
         "name": "W",
         "editsAllowed": "none",
         "description": "label-attributes-for-all-labels",
         "type": "string",
         "enum": ["foo", "buz", "buz2"]
      }
   ],
   "labels": [
      {
         "label": "Car",
         "editsAllowed": "none",
         "categoryAttributes": [
            {
               "name": "X",
               "description": "enter a number",
               "type": "number",
               "editsAllowed": "any"
            },
            {
               "name": "Y",
               "description": "select an option",
               "type": "string",
               "enum": ["y1", "y2"],
               "editsAllowed": "any"
            },
            {
               "name": "Z",
               "description": "submit a free-form response",
               "type": "string",
               "editsAllowed": "none"
            }
         ]
      }
   ]
}
```
Creating Worker Instructions

Create custom instructions for labeling jobs to improve your worker's accuracy in completing their task. Your instructions are accessible when workers select the Instructions menu option in the worker UI. Short instructions must be under 255 characters and long instructions must be under 2,048 characters.

There are two kinds of instructions:

- **Short instructions** – These instructions are shown to works when they select Instructions in the worker UI menu. They should provide an easy reference to show the worker the correct way to label an object.

- **Full instructions** – These instructions are shown when workers select More Instructions in instructions the pop-up window. We recommend that you provide detailed instructions for completing the task with multiple examples showing edge cases and other difficult situations for labeling objects.

For 3D point cloud and video frame labeling jobs, you can add worker instructions to your label category configuration file. You can use a single string to create instructions or you can add HTML mark up to customize the appearance of your instructions and add images. Make sure that any images you include in your instructions are publicly available, or if your instructions are in Amazon S3, that your workers have read-access so that they can view them.

Use Input and Output Data

The input data that you provide to Amazon SageMaker Ground Truth is sent to your workers for labeling. You choose the data to send to your workers by creating a single manifest file that defines all of the data that requires labeling or by sending input data objects to an ongoing, streaming labeling job to be labeled in real time.

The output data is the result of your labeling job. The output data file, or augmented manifest file, contains label data for each object you send to the labeling job and metadata about the label assigned to data objects.

After you create an augmented manifest file, you can use it in a training job. For a demonstration of how to use an augmented manifest to train an object detection machine learning model with Amazon SageMaker, see object_detection_augmented_manifest_training.ipynb. For more information, see Provide Dataset Metadata to Training Jobs with an Augmented Manifest File (p. 1124).

**Topics**

- Input Data (p. 341)
- 3D Point Cloud Input Data (p. 351)
- Video Frame Input Data (p. 375)
- Output Data (p. 381)
Input Data

The input data are the data objects that you send to your workforce to be labeled. There are two ways to send data objects to Ground Truth for labeling:

- Send a list of data objects that require labeling using an input manifest file.
- Send individual data objects in real time to a perpetually running, streaming labeling job.

If you have a dataset that needs to be labeled one time, and you do not require an ongoing labeling job, create a standard labeling job using an input manifest file.

If you want to regularly send new data objects to your labeling job after it has started, create a streaming labeling job. When you create a streaming labeling job, you can optionally use an input manifest file to specify a group of data that you want labeled immediately when the job starts. You can continuously send new data objects to a streaming labeling job as long as it is active.

**Note**

Streaming labeling jobs are only supported through the SageMaker API. You cannot create a streaming labeling job using the SageMaker console.

The following task types have special input data requirements and options:

- For **3D point cloud** labeling job input data requirements, see 3D Point Cloud Input Data (p. 351).
- For **video frame** labeling job input data requirements, see Video Frame Input Data (p. 375).

**Topics**

- Use an Input Manifest File (p. 341)
- Ground Truth Streaming Labeling Jobs (p. 343)
- Input Data Quotas (p. 347)
- Filter and Select Data for Labeling (p. 350)

Use an Input Manifest File

Each line in an input manifest file is an entry containing an object, or a reference to an object, to label. An entry can also contain labels from previous jobs and for some task types, additional information.

Input data and the manifest file must be stored in Amazon Simple Storage Service (Amazon S3). Each has specific storage and access requirements, as follows:

- The Amazon S3 bucket that contains the input data must be in the same AWS Region in which you are running Amazon SageMaker Ground Truth. You must give Amazon SageMaker access to the data stored in the Amazon S3 bucket so that it can read it. For more information about Amazon S3 buckets, see Working with Amazon S3 buckets.

- The manifest file must be in the same AWS Region as the data files, but it doesn't need to be in the same location as the data files. It can be stored in any Amazon S3 bucket that is accessible to the AWS Identity and Access Management (IAM) role that you assigned to Ground Truth when you created the labeling job.

**Note**

3D point cloud and video frame task types have different input manifest requirements and attributes.

For **3D point cloud** task types, refer to Create an Input Manifest File for a 3D Point Cloud Labeling Job (p. 353).
For **video frame task types**, refer to Create a Video Frame Input Manifest File (p. 379).

The manifest is a UTF-8 encoded file in which each line is a complete and valid JSON object. Each line is delimited by a standard line break, \n or \r\n. Because each line must be a valid JSON object, you can’t have unescaped line break characters. For more information about data format, see [JSON Lines](#).

Each JSON object in the manifest file can be no larger than 100,000 characters. No single attribute within an object can be larger than 20,000 characters. Attribute names can’t begin with $ (dollar sign).

Each JSON object in the manifest file must contain one of the following keys: `source-ref` or `source`. The value of the keys are interpreted as follows:

- **source-ref** – The source of the object is the Amazon S3 object specified in the value. Use this value when the object is a binary object, such as an image.
- **source** – The source of the object is the value. Use this value when the object is a text value.

The following is an example of a manifest file for files stored in an Amazon S3 bucket:

```json
{ "source-ref": "S3 bucket location 1" }
{ "source-ref": "S3 bucket location 2" }
...
{ "source-ref": "S3 bucket location n" }
```

Use the `source-ref` key for image files for bounding box, image classification (single and multi-label), semantic segmentation, and video clips for video classification labeling jobs. 3D point cloud and video frame labeling jobs also use the `source-ref` key but these labeling jobs require additional information in the input manifest file. For more information see [3D Point Cloud Input Data](#) and [Video Frame Input Data](#).

The following is an example of a manifest file with the input data stored in the manifest:

```json
{ "source": "Lorem ipsum dolor sit amet" }
{ "source": "consectetur adipiscing elit" }
...
{ "source": "mollit anim id est laborum" }
```

Use the `source` key for single and multi-label text classification and named entity recognition labeling jobs.

You can include other key-value pairs in the manifest file. These pairs are passed to the output file unchanged. This is useful when you want to pass information between your applications. For more information, see [Output Data](#).

### Automated Data Setup

You can used the automated data setup to create manifest files for your labeling jobs in the Ground Truth console using images, videos, video frames, text (.txt) files, and comma-separated value (.csv) files stored in Amazon S3.

**Important**

Ground Truth does not support automated data setup using AWS KMS encrypted buckets or objects in S3.

Before using the following procedure, ensure that your input images or files are correctly formatted:

- **Image files** – Image files must comply with the size and resolution limits listed in the tables found in [Input File Size Quota](#).
• Text files – Text data can be stored in one or more .txt files. Each item that you want labeled must be separated by a standard line break.
• CSV files – Text data can be stored in one or more .csv files. Each item that you want labeled must be in a separate row.
• Videos – Video files can be any of the following formats: MP4, OGG, and WEBM. If you want to extract video frames from your video files for object detection or object tracking, see Provide Video Files (p. 377).
• Video frames – Video frames are images extracted from a videos. All images extracted from a single video are referred to as a sequence of video frames. Each sequence of video frames must have unique prefix keys in Amazon S3. See Provide Video Frames (p. 376). For this data type, see Automated Video Frame Input Data Setup (p. 378)

**Important**
For video frame object detection and video frame object tracking labeling jobs, see Automated Video Frame Input Data Setup (p. 378) to learn how to use the automated data setup.

Use these instructions to automatically set up your input dataset connection with Ground Truth.

**Automatically connect your data in Amazon S3 with Ground Truth**

1. Navigate to the Create labeling job page in the Amazon SageMaker console: https://console.aws.amazon.com/sagemaker/.
   This link puts you in the North Virginia (us-east-1) AWS Region. If your input data is in an Amazon S3 bucket in another Region, switch to that Region. To change your AWS Region, on the navigation bar, choose the name of the currently displayed Region.
2. Select Create labeling job.
3. Enter a Job name.
4. In the section Input data setup, select Automated data setup.
5. Enter an Amazon S3 URI for S3 location for input datasets.
6. Specify your S3 location for output datasets. This is where your output data is stored.
7. Choose your Data type using the dropdown list.
8. Use the drop down menu under IAM Role to select an execution role. If you select Create a new role, specify the S3 buckets that you want grant this role permission to access. This role must have permission to access the S3 buckets you specified in Steps 5 and 6.
9. Select Complete data setup.

The following GIF demonstrates how to use the automated data setup for image data. This example will create a file, dataset-YYYYMMDDTHHmmSS.manifest in the S3 bucket example-groundtruth-images where YYYYMMDDTHHmmSS indicates the year (YY), month (MM), day (DD) and time in hours (HH), minutes (mm) and seconds (ss), that the input manifest file was created.

**Ground Truth Streaming Labeling Jobs**

If you want to perpetually send new data objects to Amazon SageMaker Ground Truth to be labeled, use a streaming labeling job. Streaming labeling jobs allow you to:

• Send new dataset objects to workers in real time using a perpetually running labeling job. Workers continuously receive new data objects to label as long as the labeling job is active and new objects are being sent to it.
• Gain visibility into the number of objects that have been queued and are waiting to be labeled. Use this information to control the flow of data objects sent to your labeling job.
• Receive label data for individual data objects in real time as workers finish labeling them.
Ground Truth streaming labeling jobs remain active until they are manually stopped or have been idle for more than 10 days. You can intermittently send new data objects to workers while the labeling job is active.

If you are a new user of Ground Truth streaming labeling jobs, it is recommended that you review How It Works (p. 344).

Use Create a Streaming Labeling Job (p. 321) to learn how to create a streaming labeling job.

**Note**
Ground Truth streaming labeling jobs are only supported through the SageMaker API.

**Topics**
- How It Works (p. 344)
- Send Data to a Streaming Labeling Job (p. 344)
- Manage Labeling Requests with an Amazon SQS Queue (p. 346)
- Receive Output Data from a Streaming Labeling Job (p. 346)
- Duplicate Message Handling (p. 346)

**How It Works**

When you create a Ground Truth streaming labeling job, the job remains active until it is manually stopped, remains idle for more than 10 days, or is unable to access input data sources. You can intermittently send new data objects to workers while it is active. A worker can continue to receive new data objects in real time as long as the total number of tasks currently available to the worker is less than the value in `MaxConcurrentTaskCount`. Otherwise, the data object is sent to a queue that Ground Truth creates on your behalf in Amazon Simple Queue Service (Amazon SQS) for later processing. These tasks are sent to workers as soon as the total number of tasks currently available to a worker falls below `MaxConcurrentTaskCount`. If a data object is not sent to a worker after 14 days, it expires. You can view the number of tasks pending in the queue and adjust the number of objects you send to the labeling job. For example, you may decrease the speed at which you send objects to the labeling job if the backlog of pending objects moves above a threshold.

**Send Data to a Streaming Labeling Job**

You can optionally submit input data to a streaming labeling job one time when you create the labeling job using an input manifest file. Once the labeling job has started and the state is `InProgress`, you can submit new data objects to your labeling job in real time using your Amazon SNS input topic and Amazon S3 event notifications.

**Submit Data Objects When you Start the Labeling Job (One Time):**
- **Use an Input Manifest File** – You can optionally specify an input manifest file Amazon S3 URI in `ManifestS3Uri` when you create the streaming labeling job. Ground Truth sends each data object in the manifest file to workers for labeling as soon as the labeling job starts. To learn more, see Create a Manifest File (Optional) (p. 324).

  After you submit a request to create the streaming labeling job, its status will be `Initializing`. Once the labeling job is active, the state changes to `InProgress` and you can start using the real-time options to submit additional data objects for labeling.

**Submit Data Objects in Real Time:**
- **Send data objects using Amazon SNS messages** – You can send Ground Truth new data objects to label by sending an Amazon SNS message. You will send this message to an Amazon SNS input topic that you create and specify when you create your streaming labeling job. For more information, see Send Data Objects Using Amazon SNS (p. 345).
• **Send data objects by placing them in an Amazon S3 bucket** – Each time you add a new data object to an Amazon S3 bucket, you can prompt Ground Truth to process that object for labeling. To do this, you add an event notification to the bucket so that it notifies your Amazon SNS input topic each time a new object is added to (or created in) that bucket. For more information, see **Send Data Objects using Amazon S3** (p. 345). This option is not available for text-based labeling jobs such as text classification and named entity recognition.

  **Important**
  If you use the Amazon S3 configuration, do not use the same S3 location for your input data configuration and your output data. You specify the S3 prefix for your output data when you create a labeling job.

**Send Data Objects Using Amazon SNS**

You can send data objects to your streaming labeling job using Amazon Simple Notification Service (Amazon SNS). Amazon SNS is a web service that coordinates and manages the delivery of messages to and from endpoints (for example, an email address or AWS Lambda function). An Amazon SNS topic acts as a communication channel between two or more endpoints. You use Amazon SNS to send, or publish, new data objects to the topic specified in the `CreateLabelingJob` parameter `SnsTopicArn` in `InputConfig`. The format of these messages is the same as a single line from an input manifest file.

For example, you may send a piece of text to an active text classification labeling job by publishing it to your input topic. The message that you publish may look similar to the following:

```json
{"source": "Lorem ipsum dolor sit amet"}
```

To send a new image object to an image classification labeling job, your message may look similar to the following:

```json
{"source-ref": "s3://awsexamplebucket/example-image.jpg"}
```

**Note**

You can also include custom deduplication IDs and deduplication keys in your Amazon SNS messages. To learn more, see **Duplicate Message Handling** (p. 346).

When Ground Truth creates your streaming labeling job, it subscribes to your Amazon SNS input topic.

**Send Data Objects using Amazon S3**

You can send one or more new data objects to a streaming labeling job by placing them in an Amazon S3 bucket that is configured with an Amazon SNS event notification. You can set up an event to notify your Amazon SNS input topic anytime a new object is created in your bucket. You must specify this same Amazon SNS input topic in the `CreateLabelingJob` parameter `SnsTopicArn` in `InputConfig`.

Anytime you configure an S3 bucket to send notifications to Amazon SNS, Ground Truth will publish a test event, "s3:TestEvent", to ensure that the topic exists and that the owner of the Amazon S3 bucket specified has permission to publish to the specified topic. It is recommended that you set up your Amazon S3 connection with Amazon SNS before starting a streaming labeling job. If you do not, this test event may register as a data object and be sent to Ground Truth for labeling.

**Important**

If you use the Amazon S3 configuration, do not use the same S3 location for your input data configuration and your output data. You specify the S3 prefix for your output data when you create a labeling job.

Once you have configured your Amazon S3 bucket and created your labeling job, you can add objects to your bucket and Ground Truth either sends that object to workers or places it on your Amazon SQS queue.
To learn more, see Set up Amazon S3 Bucket Event Notifications (p. 323).

Important
This option is not available for text-based labeling jobs such as text classification and named entity recognition.

Manage Labeling Requests with an Amazon SQS Queue

When Ground Truth creates your streaming labeling job, it creates an Amazon SQS queue in the AWS account used to create the labeling job. The queue name is `GroundTruth-labeling_job_name` where `labeling_job_name` is the name of your labeling job, in lowercase letters. When you send data objects to your labeling job, Ground Truth either sends the data objects directly to workers or places the task in your queue to be processed at a later time. If a data object is not sent to a worker after 14 days, it expires and is removed from the queue. You can setup an alarm in Amazon SQS to detect when objects expire and use this mechanism to control the volume of objects you send to your labeling job.

Important
Modifying, deleting, or sending objects directly to the Amazon SQS queue associated with your streaming labeling job may lead to job failures.

Receive Output Data from a Streaming Labeling Job

Your Amazon S3 output bucket is periodically updated with new output data from your streaming labeling job. Additionally, each time a worker submits a labeled object, a notification with the output data is sent to the Amazon SNS output topic you specify.

You can subscribe an endpoint to your SNS output topic to receive notifications or trigger events when you receive output data from a labeling task. To learn more, see Subscribe an Endpoint to Your Amazon SNS Output Topic (p. 323).

Duplicate Message Handling

For data objects sent in real time, Ground Truth guarantees idempotency by ensuring each unique object is only sent for labeling once, even if the input message referring to that object is received multiple times (duplicate messages). To do this, each data object sent to a streaming labeling job is assigned a deduplication ID, which is identified with a deduplication key.

If you send your requests to label data objects directly through your Amazon SNS input topic using Amazon SNS messages, you can optionally choose a custom deduplication key and deduplication IDs for your objects. For more information, see Specify A Deduplication Key and ID in an Amazon SNS Message (p. 346).

If you do not provide your own deduplication key, or if you use the Amazon S3 configuration to send data objects to your labeling job, Ground Truth uses one of the following for the deduplication ID:

- For messages sent directly to your Amazon SNS input topic, Ground Truth uses the SNS message ID.
- For messages that come from an Amazon S3 configuration, Ground Truth creates a deduplication ID by combining the Amazon S3 URI of the object with the sequencer token in the message.

Specify A Deduplication Key and ID in an Amazon SNS Message

When you send a data object to your streaming labeling job using an Amazon SNS message, you have the option to specify your deduplication key and deduplication ID in one of the following ways. In all of these scenarios, identify your deduplication key with `dataset-objectid-attribute-name`.

Bring Your Own Deduplication Key and ID

Create your own deduplication key and deduplication ID by configuring your Amazon SNS message as follows. Replace `byo-key` with your key and `UniqueId` with the deduplication ID for that data object.
Use an Existing Key for your Deduplication Key

You can use an existing key in your message as the deduplication key. When you do this, the value associated with that key is used for the deduplication ID.

For example, you can specify use the source-ref key as your deduplication key by formatting your message as follows:

```json
{
    "source-ref": "s3://bucket/prefix/object1",
    "dataset-objectid-attribute-name": "byo-key",
    "byo-key": "UniqueId"
}
```

In this example, Ground Truth uses "s3://bucket/prefix/object1" for the deduplication id.

Find Deduplication Key and ID in Your Output Data

You can see the deduplication key and ID in your output data. The deduplication key is identified by dataset-objectid-attribute-name.

When you use your own custom deduplication key, your output contains something similar to the following:

```json
"dataset-objectid-attribute-name": "byo-key",
"byo-key": "UniqueId",
```

When you do not specify a key, you can find the deduplication ID that Ground Truth assigned to your data object as follows. The $label-attribute-name-object-id parameter identifies your deduplication ID.

```json
{
    "source-ref": "s3://bucket/prefix/object1",
    "dataset-objectid-attribute-name": "$label-attribute-name-object-id",
    "$label-attribute-name": 0,
    "$label-attribute-name-metadata": {"...",
    "$label-attribute-name-object-id": "<service-generated-key>"
}
```

For <service-generated-key>, if the data object came through an Amazon S3 configuration, Ground Truth adds a unique value used by the service and emits a new field keyed by $sequencer which shows the S3 sequencer used. If object was fed to SNS directly, Ground Truth use the SNS message ID.

Note
Do not use the $ character in your label attribute name.

Input Data Quotas

Input datasets used in semantic segmentation labeling jobs have a quota of 20,000 items. For all other labeling job types, the dataset size quota is 100,000 items. To request an increase to the quota for
labeling jobs other than semantic segmentation jobs, review the procedures in AWS Service Quotas to request a quota increase.

Input image data for active and non-active learning labeling jobs must not exceed size and resolution quotas. Active learning refers to labeling job that use automated data labeling. Non-active learning refers to labeling jobs that don't use automated data labeling.

Additional quotas apply for label categories for all task types, and for input data and labeling category attributes for 3D point cloud and video frame task types.

**Input File Size Quota**

Input files can't exceed the following size- quotas for both active and non-active learning labeling jobs.

<table>
<thead>
<tr>
<th>Labeling Job Task Type</th>
<th>Input File Size Quota</th>
</tr>
</thead>
<tbody>
<tr>
<td>Image classification</td>
<td>40 MB</td>
</tr>
<tr>
<td>Bounding box (Object detection)</td>
<td>40 MB</td>
</tr>
<tr>
<td>Semantic segmentation</td>
<td>40 MB</td>
</tr>
<tr>
<td>Image classification label adjustment</td>
<td>40 MB</td>
</tr>
<tr>
<td>Bounding box (Object detection) label adjustment</td>
<td>40 MB</td>
</tr>
<tr>
<td>Semantic segmentation label adjustment</td>
<td>40 MB</td>
</tr>
<tr>
<td>Image classification label verification</td>
<td>40 MB</td>
</tr>
<tr>
<td>Bounding box (Object detection) label verification</td>
<td>40 MB</td>
</tr>
<tr>
<td>Semantic segmentation label verification</td>
<td>40 MB</td>
</tr>
</tbody>
</table>

**Input Image Resolution Quotas**

Image file resolution refers to the number of pixels in an image, and determines the amount of detail an image holds. Image resolution quotas differ depending on the labeling job type and the SageMaker built-in algorithm used. The following table lists the resolution quotas for images used in active and non-active learning labeling jobs.

<table>
<thead>
<tr>
<th>Labeling Job Task Type</th>
<th>Resolution Quota - Non Active Learning</th>
<th>Resolution Quota - Active Learning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Image classification</td>
<td>100 million pixels</td>
<td>3840 x 2160 pixels (4 K)</td>
</tr>
<tr>
<td>Bounding box (Object detection)</td>
<td>100 million pixels</td>
<td>3840 x 2160 pixels (4 K)</td>
</tr>
<tr>
<td>Semantic segmentation</td>
<td>100 million pixels</td>
<td>1920 x 1080 pixels (1080 p)</td>
</tr>
<tr>
<td>Image classification label adjustment</td>
<td>100 million pixels</td>
<td>3840 x 2160 pixels (4 K)</td>
</tr>
<tr>
<td>Object detection label adjustment</td>
<td>100 million pixels</td>
<td>3840 x 2160 pixels (4 K)</td>
</tr>
<tr>
<td>Semantic segmentation label adjustment</td>
<td>100 million pixels</td>
<td>1920 x 1080 pixels (1080 p)</td>
</tr>
</tbody>
</table>
Labeling Job Task Type | Resolution Quota - Non Active Learning | Resolution Quota - Active Learning
---|---|---
Image classification label verification | 100 million pixels | Not available
Object detection label verification | 100 million pixels | Not available
Semantic segmentation label verification | 100 million pixels | Not available

**Label Category Quotas**

Each labeling job task type has a quota for the number of label categories you can specify. Workers select label categories to create annotations. For example, you may specify label categories *car*, *pedestrian*, and *biker* when creating a bounding box labeling job and workers will select the *car* category before drawing bounding boxes around cars.

**Important**

- Label category names cannot exceed 256 characters.
- All label categories must be unique. You cannot specify duplicate label categories.

The following label category limits apply to labeling jobs. Quotas for label categories depend on whether you use the SageMaker API operation `CreateLabelingJob` or the console to create a labeling job.

<table>
<thead>
<tr>
<th>Labeling Job Task Type</th>
<th>Label Category Quota - API</th>
<th>Label Category Quota - Console</th>
</tr>
</thead>
<tbody>
<tr>
<td>Image classification (Multi-label)</td>
<td>50</td>
<td>50</td>
</tr>
<tr>
<td>Image classification (Single label)</td>
<td>Unlimited</td>
<td>30</td>
</tr>
<tr>
<td>Bounding box (Object detection)</td>
<td>50</td>
<td>50</td>
</tr>
<tr>
<td>Label verification</td>
<td>Unlimited</td>
<td>30</td>
</tr>
<tr>
<td>Semantic segmentation (with active learning)</td>
<td>20</td>
<td>10</td>
</tr>
<tr>
<td>Semantic segmentation (without active learning)</td>
<td>Unlimited</td>
<td>10</td>
</tr>
<tr>
<td>Named entity recognition</td>
<td>Unlimited</td>
<td>30</td>
</tr>
<tr>
<td>Text classification (Multi-label)</td>
<td>50</td>
<td>50</td>
</tr>
<tr>
<td>Text classification (Single label)</td>
<td>Unlimited</td>
<td>30</td>
</tr>
<tr>
<td>Video classification</td>
<td>30</td>
<td>30</td>
</tr>
<tr>
<td>Video frame object detection</td>
<td>30</td>
<td>30</td>
</tr>
<tr>
<td>Video frame object tracking</td>
<td>30</td>
<td>30</td>
</tr>
<tr>
<td>3D point cloud object detection</td>
<td>30</td>
<td>30</td>
</tr>
<tr>
<td>3D point cloud object tracking</td>
<td>30</td>
<td>30</td>
</tr>
</tbody>
</table>
3D Point Cloud and Video Frame Labeling Job Quotas

The following quotas apply for 3D point cloud and video frame labeling job input data.

<table>
<thead>
<tr>
<th>Labeling Job Task Type</th>
<th>Input Data Quota</th>
</tr>
</thead>
<tbody>
<tr>
<td>Video frame object detection</td>
<td>2,000 video frames (images) per sequence</td>
</tr>
<tr>
<td>Video frame object detection</td>
<td>10 video frame sequences per manifest file</td>
</tr>
<tr>
<td>Video frame object tracking</td>
<td>2,000 video frames (images) per sequence</td>
</tr>
<tr>
<td>Video frame object tracking</td>
<td>10 video frame sequences per manifest file</td>
</tr>
<tr>
<td>3D point cloud object detection</td>
<td>100,000 point cloud frames per labeling job</td>
</tr>
<tr>
<td>3D point cloud object tracking</td>
<td>100,000 point cloud frame sequences per labeling job</td>
</tr>
<tr>
<td>3D point cloud object tracking</td>
<td>500 point cloud frames in each sequence file</td>
</tr>
</tbody>
</table>

When you create a video frame or 3D point cloud labeling job, you can add one or more label category attributes to each label category that you specify to have workers provide more information about an annotation.

Each label category attribute has a single label category attribute name, and a list of one or more options (values) to choose from. To learn more, see Worker User Interface (UI) (p. 255) for 3D point cloud labeling jobs and Worker User Interface (UI) (p. 210) for video frame labeling jobs.

The following quotas apply to the number of label category attributes names and values you can specify for labeling jobs.

<table>
<thead>
<tr>
<th>Labeling Job Task Type</th>
<th>Label Category Attribute (name) Quota</th>
<th>Label Category Attribute Values Quota</th>
</tr>
</thead>
<tbody>
<tr>
<td>Video frame object detection</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>Video frame object tracking</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>3D point cloud object detection</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>3D point cloud object tracking</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>3D point cloud semantic segmentation</td>
<td>10</td>
<td>10</td>
</tr>
</tbody>
</table>

Filter and Select Data for Labeling

You can use the Amazon SageMaker console to select a portion of your dataset for labeling. The data must be stored in an Amazon S3 bucket. You have three options:
• Use the full dataset.
• Choose a randomly selected sample of the dataset.
• Specify a subset of the dataset using a query.

The following options are available in the Labeling jobs section of the SageMaker console after selecting Create labeling job. To learn how to create a labeling job in the console, see Getting started (p. 162). To configure the dataset that you use for labeling, in the Job overview section, choose Additional configuration.

Use the Full Dataset

When you choose to use the Full dataset, you must provide a manifest file for your data objects. You can provide the path of the Amazon S3 bucket that contains the manifest file or use the SageMaker console to create the file. To learn how to create a manifest file using the console, see Automated Data Setup (p. 342).

Choose a Random Sample

When you want to label a random subset of your data, select Random sample. The dataset is stored in the Amazon S3 bucket specified in the Input dataset location field.

After you have specified the percentage of data objects that you want to include in the sample, choose Create subset. SageMaker randomly picks the data objects for your labeling job. After the objects are selected, choose Use this subset.

SageMaker creates a manifest file for the selected data objects. It also modifies the value in the Input dataset location field to point to the new manifest file.

Specify a Subset

You can specify a subset of your data objects using an Amazon S3 SELECT query on the object file names.

The SELECT statement of the SQL query is defined for you. You provide the WHERE clause to specify which data objects should be returned.

For more information about the Amazon S3 SELECT statement, see Selecting Content from Objects.

Choose Create subset to start the selection, and then choose Use this subset to use the selected data.

SageMaker creates a manifest file for the selected data objects. It also updates the value in the Input dataset location field to point to the new manifest file.

3D Point Cloud Input Data

To create a 3D point cloud labeling job, you must create an input manifest file. Use this topic to learn the formatting requirements of the input manifest file for each task type. To learn about the raw input data formats Ground Truth accepts for 3D point cloud labeling jobs, see the section Accepted Raw 3D Data Formats (p. 352).

Use your labeling job task type to choose a topics on Create an Input Manifest File for a 3D Point Cloud Labeling Job (p. 353) to learn about the formatting requirements for each line of your input manifest file.

Topics
• Accepted Raw 3D Data Formats (p. 352)
Accepted Raw 3D Data Formats

Ground Truth uses your 3D point cloud data to render a 3D scenes that workers annotate. This section describes the raw data formats that are accepted for point cloud data and sensor fusion data for a point cloud frame. To learn how to create an input manifest file to connect your raw input data files with Ground Truth, see Create an Input Manifest File for a 3D Point Cloud Labeling Job (p. 353).

For each frame, Ground Truth supports Compact Binary Pack Format (.bin) and ASCII (.txt) files. These files contain information about the location (x, y, and z coordinates) of all points that make up that frame, and, optionally, information about the pixel color of each point for colored point clouds. When you create a 3D point cloud labeling job input manifest file, you can specify the format of your raw data in the `format` parameter.

The following table lists elements that Ground Truth supports in point cloud frame files to describe individual points.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>The x coordinate of the point.</td>
</tr>
<tr>
<td>y</td>
<td>The y coordinate of the point.</td>
</tr>
<tr>
<td>z</td>
<td>The z coordinate of the point.</td>
</tr>
<tr>
<td>i</td>
<td>The intensity of the point.</td>
</tr>
<tr>
<td>r</td>
<td>The red color channel component. An 8-bit value (0-255).</td>
</tr>
<tr>
<td>g</td>
<td>The green color channel component. An 8-bit value (0-255)</td>
</tr>
<tr>
<td>b</td>
<td>The blue color channel component. An 8-bit value (0-255)</td>
</tr>
</tbody>
</table>

Ground Truth assumes the following about your input data:

- All of the positional coordinates (x, y, z) are in meters.
- All the pose headings (qx, qy, qz, qw) are measured in Spatial Quaternions.

Compact Binary Pack Format

The Compact Binary Pack Format represents a point cloud as an ordered set of a stream of points. Each point in the stream is an ordered binary pack of 4-byte float values in some variant of the form \texttt{xyzirgb}. The x, y, and z elements are required and additional information about that pixel can be included in a variety of ways using i, r, g, and b.

To use a binary file to input point cloud frame data to a Ground Truth 3D point cloud labeling job, enter `binary/` in the `format` parameter for your input manifest file and replace `.` with the order of elements in each binary pack. For example, you may enter one of the following for the `format` parameter.

- `binary/xyzii–` When you use this format, your point element stream would be in the following order: `x1y1z1i1i2x2y2z2i2i3...`
• binary/xyzrgb – When you use this format, your point element stream would be in the following order: x1y1z1r1g1b1x2y2z2r2g2b2 ...
• binary/xyzirgb – When you use this format, your point element stream would be in the following order: x1y1z1i1r1g1b1x2y2z2i2r2g2b2 ...

When you use a binary file for your point cloud frame data, if you do not enter a value for format, the default pack format binary/xyz is used.

**ASCII Format**

The ASCII format uses a text file to represent a point cloud, where each line in the ASCII point cloud file represents a single point. Each point is a line the text file and contains white space separated values, each of which is a 4-byte float ASCII values. The x, y, and z elements are required for each point and additional information about that point can be included in a variety of ways using i, r, g, and b.

To use a text file to input point cloud frame data to a Ground Truth 3D point cloud labeling job, enter text/ in the format parameter for your input manifest file and replace with the order of point elements on each line.

For example, if you enter text/xyzi for format, your text file for each point cloud frame should look similar to the following:

```
x1 y1 z1 i1
x2 y2 z2 i2
...
```

If you enter text/xyzrgb, your text file should look similar to the following:

```
x1 y1 z1 r1 g1 b1
x2 y2 z2 r2 g2 b1
...
```

When you use a text file for your point cloud frame data, if you do not enter a value for format, the default format text/xyzi will be used.

**Point Cloud Resolution Limits**

Ground Truth does not have a resolution limit for 3D point cloud frames. However, we recommend that you limit each point cloud frame to 500K points for optimal performance. When Ground Truth renders the 3D point cloud visualization, it must be viewable on your workers' computers, which depends on workers' computer hardware. Point cloud frames that are larger than 1 million points may not render on standard machines, or may take too long to load.

**Create an Input Manifest File for a 3D Point Cloud Labeling Job**

When you create a labeling job, you provide an input manifest file where each line of the manifest describes a unit of task to be completed by annotators. The format of your input manifest file depends on your task type.

• If you are creating a 3D point cloud object detection or semantic segmentation labeling job, each line in your input manifest file contains information about a single 3D point cloud frame. This is called a point cloud frame input manifest. To learn more, see Create a Point Cloud Frame Input Manifest File (p. 354).
• If you are creating a 3D point cloud object tracking labeling job, each line of your input manifest file contains a sequence of 3D point cloud frames and associated data. This is called a point cloud sequence input manifest. To learn more, see Create a Point Cloud Sequence Input Manifest (p. 360).
Create a Point Cloud Frame Input Manifest File

In the single-frame input manifest file, each line in the manifest contains data for a single point cloud frame. The point cloud frame data can either be stored in binary or ASCII format (see Accepted Raw 3D Data Formats (p. 352)). This is the manifest file formatting required for 3D point cloud object detection and semantic segmentation. Optionally, you can also provide camera sensor fusion data for each point cloud frame.

Ground Truth supports point cloud and video camera sensor fusion in the world coordinate system (p. 366) for all modalities. If you can obtain your 3D sensor extrinsic (like a LiDAR extrinsic), we recommend that you transform 3D point cloud frames into the world coordinate system using the extrinsic. For more information, see Sensor Fusion (p. 368).

However, if you cannot obtain a point cloud in world coordinate system, you can provide coordinates in the original coordinate system that the data was captured in. If you are providing camera data for sensor fusion, it is recommended that you provide LiDAR sensor and camera pose in the world coordinate system.

To create a single-frame input manifest file, you will identify the location of each point cloud frame that you want workers to label using the `source-ref` key. Additionally, you must use the `source-ref-metadata` key to identify the format of your dataset, a timestamp for that frame, and, optionally, sensor fusion data and video camera images.

The following example demonstrates the syntax used for an input manifest file for a single-frame point cloud labeling job. The example includes two point cloud frames. For details about each parameter, see the table following this example.

**Important**
The following code block shows an input manifest file with two JSON objects. Each JSON object is used to point to and provide details about a single point cloud frame. The JSON objects have been expanded for readability, but you must minimize each JSON object to fit on a single line when creating an input manifest file. An example is provided under this code block.

```json
{
    "source-ref": "s3://awsexamplebucket/examplefolder/frame1.bin",
    "source-ref-metadata": {
        "format": "binary/xyziz",
        "unix-timestamp": 1566861644.759115,
        "ego-vehicle-pose": {
            "position": {
                "x": -2.7161461413869947,
                "y": 116.25822288149078,
                "z": 1.8348751887989483
            },
            "heading": {
                "qx": -0.02111296123795955,
                "qy": -0.006495469416730261,
                "qz": -0.008024565904865688,
                "qw": 0.9997181192298087
            }
        },
        "prefix": "s3://awsexamplebucket/lidar_singleframe_dataset/someprefix/",
        "images": [ {
            "image-path": "images/frame300.bin_camera0.jpg",
            "unix-timestamp": 1566861644.759115,
            "fx": 847.7962624528487,
            "fy": 850.0340893791985,
            "cx": 576.2129134707038,
            "cy": 317.2423573573745,
            "k1": 0,
            "k2": 0,
            "k3": 0,
        } ]
```
"k4": 0,
"p1": 0,
"p2": 0,
"skew": 0,
"position": {
  "x": -2.2722515189268138,
  "y": 116.86003310568965,
  "z": 1.456614668542299
},
"heading": {
  "qx": 0.7594754093069037,
  "qy": 0.02181790885672969,
  "qz": -0.02461725233103356,
  "qw": -0.6496916273040025
},
"camera-model": "pinhole"}
}
}
}
{
"source-ref": "s3://awsexamplebucket/examplefolder/frame2.bin",
"source-ref-metadata": {
  "format": "binary/xyzi",
  "unix-timestamp": 1566861632.759133,
  "ego-vehicle-pose": {
    "position": {
      "x": -2.7161461413869947,
      "y": 116.25822288149078,
      "z": 1.8348751887989483
    },
    "heading": {
      "qx": -0.02111296123795955,
      "qy": -0.006495469416730261,
      "qz": -0.008024565904865688,
      "qw": 0.9997181192298087
    },
    "camera-model": "pinhole"
  }
},
"prefix": "s3://awsexamplebucket/lidar_singleframe_dataset/someprefix/",
"images": [
  {
    "image-path": "images/frame300.bin_Camera0.jpg",
    "unix-timestamp": 1566861644.759115,
    "fx": 847.7962624528487,
    "fy": 850.0340893791985,
    "cx": 576.2129134707038,
    "cy": 317.2423573573745,
    "k1": 0,
    "k2": 0,
    "k3": 0,
    "k4": 0,
    "p1": 0,
    "p2": 0,
    "skew": 0,
    "position": {
      "x": -2.2722515189268138,
      "y": 116.86003310568965,
      "z": 1.456614668542299
    },
    "heading": {
      "qx": 0.7594754093069037,
      "qy": 0.02181790885672969,
      "qz": -0.02461725233103356,
      "qw": -0.6496916273040025
    },
    "camera-model": "pinhole"
  }
]
When you create an input manifest file, you must collapse your JSON objects to fit on a single line. For example, the code block above would appear as follows in an input manifest file:

```json
{
"source-ref":"s3://awsexamplebucket/examplefolder/frame1.bin","source-ref-metadata":{
"format":"binary/xyzi","unix-timestamp":1566861644.759115,"ego-vehicle-pose":{"position":
{x":-2.7161461413869947,"y":116.2582288149078,"z":1.8348751887948343},"heading":
{"qx":-0.02111296123795955,"qy":-0.006495469416730261,"qz":-0.008024565904865688,"qw":0.99971811921298088},
"prefix":"s3://awsexamplebucket/lidar_singleframe_dataset/someprefix/","images":
{"image-path":"images/frame300.bin_camera0.jpg","unix-timestamp":1566861644.759115,"fx":847.7962624528487,"fy":850.0340893791985,"cx":576.2129134707038,"cy":317.2423573573745,"k1":0,"k2":0,"k3":0,"k4":0,"p1":0,"p2":0,"skew":0,"position":
{x":-2.2722515189268138,"y":116.86003310568965,"z":1.454614668542299},"heading":
{"qx":0.7594754093069037,"qy":0.021817908856729969,"qz":-0.0246172523103356,"qw":-0.64969162730400255},
"camera-model":"pinhole"}}}}
{
"source-ref":"s3://awsexamplebucket/examplefolder/frame2.bin","source-ref-metadata":{
"format":"binary/xyzi","unix-timestamp":1566861632.759133,"ego-vehicle-pose":{"position":
{x":-2.7161461413869947,"y":116.2582288149078,"z":1.8348751887948343},"heading":
{"qx":-0.02111296123795955,"qy":-0.006495469416730261,"qz":-0.008024565904865688,"qw":0.99971811921298088},
"prefix":"s3://awsexamplebucket/lidar_singleframe_dataset/someprefix/","images":
{"image-path":"images/frame300.bin_camera0.jpg","unix-timestamp":1566861644.759115,"fx":847.7962624528487,"fy":850.0340893791985,"cx":576.2129134707038,"cy":317.2423573573745,"k1":0,"k2":0,"k3":0,"k4":0,"p1":0,"p2":0,"skew":0,"position":
{x":-2.2722515189268138,"y":116.86003310568965,"z":1.454614668542299},"heading":
{"qx":0.7594754093069037,"qy":0.021817908856729969,"qz":-0.0246172523103356,"qw":-0.64969162730400255},
"camera-model":"pinhole"}}}}
```

The following table shows the parameters you can include in your input manifest file:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Required</th>
<th>Accepted Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>source-ref</td>
<td>Yes</td>
<td>String</td>
<td>Accepted string value format: s3://&lt;bucket-name&gt;/&lt;folder-name&gt;/point-cloud-frame-file</td>
</tr>
<tr>
<td>source-ref-metadata</td>
<td>Yes</td>
<td>JSON object</td>
<td>Use this parameter to include additional information about the point cloud in source-ref, and to provide camera data for sensor fusion.</td>
</tr>
<tr>
<td>format</td>
<td>No</td>
<td>String</td>
<td>Accepted string values: &quot;binary/xyz&quot;, &quot;binary/xyzi&quot;, &quot;binary/xyzrgb&quot;, &quot;binary/xyzirgb&quot;, &quot;text/xyz&quot;, &quot;text/xyzi&quot;, (\text{Accepted Raw 3D Data Formats (p. 352).})</td>
</tr>
<tr>
<td>Parameter</td>
<td>Required</td>
<td>Accepted Values</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------</td>
<td>----------</td>
<td>----------------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;text/xyzrgb&quot;, &quot;text/xyzirgb&quot;</td>
<td>Default Values:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>When the file identified in source-ref has a .bin extension, binary/xyz</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>When the file identified in source-ref has a .txt extension, text/xyz</td>
</tr>
<tr>
<td>unix-timestamp</td>
<td>Yes</td>
<td>Number</td>
<td>The unix timestamp is the number of seconds since January 1st, 1970 until the UTC time that the data was collected by a sensor.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>A unix timestamp.</td>
<td></td>
</tr>
<tr>
<td>ego-vehicle-pose</td>
<td>No</td>
<td>JSON object</td>
<td>The pose of the device used to collect the point cloud data. For more information about this parameter, see Include Vehicle Pose Information in Your Input Manifest (p. 358).</td>
</tr>
<tr>
<td>prefix</td>
<td>No</td>
<td>String</td>
<td>The location in Amazon S3 where your metadata, such as camera images, is stored for this frame.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Accepted string value format:</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>s3://&lt;bucket-name&gt;/&lt;folder-name&gt;/</td>
<td>The prefix must end with a forward slash: /.</td>
</tr>
<tr>
<td>images</td>
<td>No</td>
<td>List</td>
<td>A list of parameters describing color camera images used for sensor fusion. You can include up to 8 images in this list. For more information about the parameters required for each image, see Include Camera Data in Your Input Manifest (p. 358).</td>
</tr>
</tbody>
</table>
Include Vehicle Pose Information in Your Input Manifest

Use the ego-vehicle location to provide information about the location of the vehicle used to capture point cloud data. Ground Truth use this information to compute LiDAR extrinsic matrix.

Ground Truth uses extrinsic matrices to project labels to and from the 3D scene and 2D images. For more information, see Sensor Fusion (p. 368).

The following table provides more information about the position and orientation (heading) parameters that are required when you provide ego-vehicle information.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Required</th>
<th>Accepted Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>position</td>
<td>Yes</td>
<td>JSON object</td>
<td>The translation vector of the ego vehicle in the world coordinate system. Required Parameters: x, y, and z. Enter numbers for these parameters.</td>
</tr>
<tr>
<td>heading</td>
<td>Yes</td>
<td>JSON Object</td>
<td>The orientation of the frame of reference of the device or sensor mounted on the vehicle sensing the surrounding, measured in quaternions, (qx, qy, qz, qw) in the a coordinate system. Required Parameters: qx, qy, qz, and qw. Enter numbers for these parameters.</td>
</tr>
</tbody>
</table>

Include Camera Data in Your Input Manifest

If you want to include video camera data with a frame, use the following parameters to provide information about each image. The Required column below applies when the images parameter is included in the input manifest file under source-ref-metadata. You are not required to include images in your input manifest file.

If you include camera images, you must include information about the camera position and heading used the capture the images in the world coordinate system.

If your images are distorted, Ground Truth can automatically undistort them using information you provide about the image in your input manifest file, including distortion coefficients (k1, k2, k3, k4, p1, p1), the camera model and the camera intrinsic matrix. The intrinsic matrix is made up of focal length (fx, fy), and the principal point (cx, cy). See Intrinsic Matrix (p. 370) to learn how Ground Truth uses the camera intrinsic. If distortion coefficients are not included, Ground Truth will not undistort an image.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Required</th>
<th>Accepted Values</th>
<th>Description</th>
</tr>
</thead>
</table>
| image-path | Yes      | String          | The relative location, in Amazon S3 of your image file. This relative path will be appended to the path you specify in prefix. Example of format: 

```
<folder-name>/
<imagefile.png>
```
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Required</th>
<th>Accepted Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>unix-timestamp</td>
<td>Yes</td>
<td>Number</td>
<td>The unix timestamp is the number of seconds since January 1st, 1970 until the UTC time that the data was collected by a camera.</td>
</tr>
<tr>
<td>camera-model</td>
<td>No</td>
<td>String:</td>
<td>The model of the camera used to capture the image. This information is used to undistort camera images.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Accepted Values</td>
<td>&quot;pinhole&quot;, &quot;fisheye&quot;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Default:</td>
<td>&quot;pinhole&quot;</td>
</tr>
<tr>
<td>fx, fy</td>
<td>Yes</td>
<td>Numbers</td>
<td>The focal length of the camera, in the x (fx) and y (fy) directions.</td>
</tr>
<tr>
<td>cx, cy</td>
<td>Yes</td>
<td>Numbers</td>
<td>The x (cx) and y (cy) coordinates of the principal point.</td>
</tr>
<tr>
<td>k1, k2, k3, k4</td>
<td>No</td>
<td>Number</td>
<td>Radial distortion coefficients. Supported for both fisheye and pinhole camera models.</td>
</tr>
<tr>
<td>p1, p2</td>
<td>No</td>
<td>Number</td>
<td>Tangential distortion coefficients. Supported for pinhole camera models.</td>
</tr>
<tr>
<td>skew</td>
<td>No</td>
<td>Number</td>
<td>A parameter to measure the skew of an image.</td>
</tr>
<tr>
<td>position</td>
<td>Yes</td>
<td>JSON object</td>
<td>The location or origin of the frame of reference of the camera mounted on the vehicle capturing images.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Required Parameters:</td>
<td>x, y, and z. Enter numbers for these parameters.</td>
</tr>
<tr>
<td>heading</td>
<td>Yes</td>
<td>JSON Object</td>
<td>The orientation of the frame of reference of the camera mounted on the vehicle capturing images, measured using quaternions, (qx, qy, qz, qw), in the world coordinate system.</td>
</tr>
</tbody>
</table>
Point Cloud Frame Limits

You can include up to 100,000 point cloud frames in your input manifest file. 3D point cloud labeling job have longer pre-processing times than other Ground Truth task types. For more information, see Job Pre-processing Time (p. 254).

Create a Point Cloud Sequence Input Manifest

In the point cloud sequence input manifest file, each line in the manifest contains a sequence of point cloud frames. The point cloud data for each frame in the sequence can either be stored in binary or ASCII format. For more information, see Accepted Raw 3D Data Formats (p. 352). This is the manifest file formatting required for 3D point cloud object tracking. Optionally, you can also provide point attribute and camera sensor fusion data for each point cloud frame. When you create a sequence input manifest file, you must provide LiDAR and video camera sensor fusion data in a world coordinate system (p. 366).

The following example demonstrates the syntax used for an input manifest file when each line in the manifest is a sequence file.

```json
{"source-ref": "s3://awsexamplebucket/example-folder/seq1.json"}
{"source-ref": "s3://awsexamplebucket/example-folder/seq2.json"}
```

The data for each sequence of point cloud frames needs to be stored in a JSON data object. The following is an example of the format you use for a sequence file. Information about each frame is included as a JSON object and is listed in the frames list. This is an example of a sequence file with two point cloud frame files, frame300.bin and frame303.bin. The ... is used to indicated where you should include information for additional frames. Add a JSON object for each frame in the sequence.

The following code block includes a JSON object for a single sequence file. The JSON object has been expanded for readability.

```json
{
  "seq-no": 1,
  "prefix": "s3://awsexamplebucket/example_lidar_sequence_dataset/seq1/",
  "number-of-frames": 100,
  "frames": [
    {
      "frame-no": 300,
      "unix-timestamp": 1566861644.759115,
      "frame": "example_lidar_frames/frame300.bin",
      "format": "binary/xyzi",
      "ego-vehicle-pose": {
        "position": {
          "x": -2.7161461413869947,
          "y": 116.25822288149078,
          "z": 1.8348751887989483
        },
        "heading": {
          "qx": -0.02111296123795955,
          "qy": -0.006495469416730261,
          "qz": -0.008024565904865688,
          "qw": 0.9997181192298087
        }
      },
      "images": [
        {
          "image-path": "example_images/frame300.bin_camera0.jpg",
          "unix-timestamp": 1566861644.759115,
          "fx": 847.7962624528487,
          "fy": 850.0340893791985,
          "cx": 576.2129134707038,
          "cy": 317.2423573573745
        }
      ]
    }
  ]
}```
The following table provides details about the top-level parameters of a sequence file. For detailed information about the parameters required for individual frames in the sequence file, see Parameters for Individual Point Cloud Frames (p. 362).

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Required</th>
<th>Accepted Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>seq-no</td>
<td>Yes</td>
<td>Integer</td>
<td>The ordered number of the sequence.</td>
</tr>
</tbody>
</table>
| prefix            | Yes      | String              | The Amazon S3 location where the sequence files are located. The prefix must end with a forward slash: /.
|                   |          | **Accepted Values:** |                                                                             |
|                   |          | s3://<bucket-name>/<prefix>/ |
| number-of-frames  | Yes      | Integer             | The total number of frames included in the sequence file. This number must match the total number of frames listed in the frames parameter in the next row. |
| frames            | Yes      | List of JSON objects| A list of frame data. The length of the list                                |
Parameters for Individual Point Cloud Frames

The following table shows the parameters you can include in your input manifest file.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Required</th>
<th>Accepted Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>frame-no</td>
<td>No</td>
<td>Integer</td>
<td>A frame number. This is an optional identifier specified by the customer to identify the frame within a sequence. It is not used by Ground Truth.</td>
</tr>
<tr>
<td>unix-timestamp</td>
<td>Yes</td>
<td>Number</td>
<td>The unix timestamp is the number of seconds since January 1st, 1970 until the UTC time that the data was collected by a sensor.</td>
</tr>
<tr>
<td>frame</td>
<td>Yes</td>
<td>String</td>
<td>The relative location, in Amazon S3 of your sequence file. This relative path will be appended to the path you specify in prefix.</td>
</tr>
<tr>
<td>format</td>
<td>No</td>
<td>String</td>
<td>Use this parameter to specify the format of your point cloud data. For more information, see Accepted Raw 3D Data Formats (p. 352).</td>
</tr>
</tbody>
</table>

For details about the format of each frame, see Parameters for Individual Point Cloud Frames (p. 362).
Use Input and Output Data

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Required</th>
<th>Accepted Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ego-vehicle-pose</td>
<td>No</td>
<td>JSON object</td>
<td>The pose of the device used to collect the point cloud data. For more information about this parameter, see Include Vehicle Pose Information in Your Input Manifest (p. 363).</td>
</tr>
</tbody>
</table>
| prefix             | No       | String                                                                           | The location in Amazon S3 where your metadata, such as camera images, is stored for this frame. The prefix must end with a forward slash: /.
|                    |          | Accepted string value format:                                                    | s3://<bucket-name>/<folder-name>/                                                                                          |
| images             | No       | List                                                                             | A list parameters describing color camera images used for sensor fusion. You can include up to 8 images in this list. For more information about the parameters required for each image, see Include Camera Data in Your Input Manifest (p. 364). |

Include Vehicle Pose Information in Your Input Manifest

Use the ego-vehicle location to provide information about the pose of the vehicle used to capture point cloud data. Ground Truth use this information to compute LiDAR extrinsic matrices.

Ground Truth uses extrinsic matrices to project labels to and from the 3D scene and 2D images. For more information, see Sensor Fusion (p. 368).

The following table provides more information about the position and orientation (heading) parameters that are required when you provide ego-vehicle information.
Use Input and Output Data

### Parameter | Required | Accepted Values | Description
--- | --- | --- | ---
position | Yes | JSON object | The translation vector of the ego vehicle in the world coordinate system.

**Required Parameters:**
- x, y, and z. Enter numbers for these parameters.

heading | Yes | JSON Object | The orientation of the frame of reference of the device or sensor mounted on the vehicle sensing the surrounding, measured in quaternions, (qx, qy, qz, qw) in the a coordinate system.

**Required Parameters:**
- qx, qy, qz, and qw. Enter numbers for these parameters.

---

**Include Camera Data in Your Input Manifest**

If you want to include color camera data with a frame, use the following parameters to provide information about each image. The **Required** column in the following table applies when the images parameter is included in the input manifest file. You are not required to include images in your input manifest file.

If you include camera images, you must include information about the **position** and orientation (**heading**) of the camera used to capture the images.

If your images are distorted, Ground Truth can automatically undistort them using information you provide about the image in your input manifest file, including distortion coefficients (k1, k2, k3, k4, p1, p1), camera model and focal length (fx, fy), and the principal point (cx, cy). To learn more about these coefficients and undistorting images, see [Camera calibration With OpenCV](https://docs.opencv.org/master/d0/d1d/tutorial_py_calibration.html). If distortion coefficients are not included, Ground Truth will not undistort an image.

### Parameter | Required | Accepted Values | Description
--- | --- | --- | ---
image-path | Yes | String | The relative location, in Amazon S3 of your image file. This relative path will be appended to the path you specify in prefix.

**Example of format:**
- `<folder-name>/imagefile.png`

unix-timestamp | Yes | Number | The timestamp of the image.

camera-model | No | String: | The model of the camera used to capture the image. This information is used to undistort camera images.

**Accepted Values:**
- "pinhole",
- "fisheye"

**Default:**
### Use Input and Output Data

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Required</th>
<th>Accepted Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;pinhole&quot;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fx, fy</td>
<td>Yes</td>
<td>Numbers</td>
<td>The focal length of the camera, in the x (fx) and y (fy) directions.</td>
</tr>
<tr>
<td>cx, cy</td>
<td>Yes</td>
<td>Numbers</td>
<td>The x (cx) and y (cy) coordinates of the principal point.</td>
</tr>
<tr>
<td>k1, k2, k3, k4</td>
<td>No</td>
<td>Number</td>
<td>Radial distortion coefficients. Supported for both fisheye and pinhole camera models.</td>
</tr>
<tr>
<td>p1, p2</td>
<td>No</td>
<td>Number</td>
<td>Tangential distortion coefficients. Supported for pinhole camera models.</td>
</tr>
<tr>
<td>skew</td>
<td>No</td>
<td>Number</td>
<td>A parameter to measure any known skew in the image.</td>
</tr>
<tr>
<td>position</td>
<td>Yes</td>
<td>JSON object</td>
<td>The location or origin of the frame of reference of the camera mounted on the vehicle capturing images.</td>
</tr>
<tr>
<td>heading</td>
<td>Yes</td>
<td>JSON Object</td>
<td>The orientation of the frame of reference of the camera mounted on the vehicle capturing images, measured using quaternions, (qx, qy, qz, qw).</td>
</tr>
</tbody>
</table>

#### Sequence File and Point Cloud Frame Limits

You can include up to 100,000 point cloud frame sequences in your input manifest file. You can include up to 500 point cloud frames in each sequence file.

Keep in mind that 3D point cloud labeling job have longer pre-processing times than other Ground Truth task types. For more information, see Job Pre-processing Time (p. 254).

#### Understand Coordinate Systems and Sensor Fusion

Point cloud data is always located in a coordinate system. This coordinate system may be local to the vehicle or the device sensing the surroundings, or it may be a world coordinate system. When you use Ground Truth 3D point cloud labeling jobs, all the annotations are generated using the coordinate system of your input data. For some labeling job task types and features, you must provide data in a world coordinate system.

In this topic, you'll learn the following:
• When you are required to provide input data in a world coordinate system or global frame of reference.
• What a world coordinate is and how you can convert point cloud data to a world coordinate system.
• How you can use your sensor and camera extrinsic matrices to provide pose data when using sensor fusion.

Coordinate System Requirements for Labeling Jobs

If your point cloud data was collected in a local coordinate system, you can use an extrinsic matrix of the sensor used to collect the data to convert it to a world coordinate system or a global frame of reference. If you cannot obtain an extrinsic for your point cloud data and, as a result, cannot obtain point clouds in a world coordinate system, you can provide point cloud data in a local coordinate system for 3D point cloud object detection and semantic segmentation task types.

For object tracking, you must provide point cloud data in a world coordinate system. This is because when you are tracking objects across multiple frames, the ego vehicle itself is moving in the world and so all of the frames need a point of reference.

If you include camera data for sensor fusion, it is recommended that you provide camera poses in the same world coordinate system as the 3D sensor (such as a LiDAR sensor).

Using Point Cloud Data in a World Coordinate System

This section explains what a world coordinate system (WCS), also referred to as a global frame of reference, is and explains how you can provide point cloud data in a world coordinate system.

What is a World Coordinate System?

A WCS or global frame of reference is a fixed universal coordinate system in which vehicle and sensor coordinate systems are placed. For example, if multiple point cloud frames are located in different coordinate systems because they were collected from two sensors, a WCS can be used to translate all of the coordinates in these point cloud frames into a single coordinate system, where all frames have the same origin, (0,0,0). This transformation is done by translating the origin of each frame to the origin of the WCS using a translation vector, and rotating the three axes (typically x, y, and z) to the right orientation using a rotation matrix. This rigid body transformation is called a homogeneous transformation.

A world coordinate system is important in global path planning, localization, mapping, and driving scenario simulations. Ground Truth uses the right-handed Cartesian world coordinate system such as the one defined in ISO 8855, where the x axis is forward toward the car's movement, y axis is left, and the z axis points up from the ground.

The global frame of reference depends on the data. Some datasets use the LiDAR position in the first frame as the origin. In this scenario, all the frames use the first frame as a reference and device heading and position will be near the origin in the first frame. For example, KITTI datasets have the first frame as a reference for world coordinates. Other datasets use a device position that is different from the origin.

Note that this is not the GPS/IMU coordinate system, which is typically rotated by 90 degrees along the z-axis. If your point cloud data is in a GPS/IMU coordinate system (such as OxTS in the open source AV KITTI dataset), then you need to transform the origin to a world coordinate system (typically the vehicle's reference coordinate system). You apply this transformation by multiplying your data with transformation metrics (the rotation matrix and translation vector). This will transform the data from its original coordinate system to a global reference coordinate system. Learn more about this transformation in the next section.

Convert 3D Point Cloud Data to a WCS

Ground Truth assumes that your point cloud data has already been transformed into a reference coordinate system of your choice. For example, you can choose the reference coordinate system of the
sensor (such as LiDAR) as your global reference coordinate system. You can also take point clouds from various sensors and transform them from the sensor’s view to the vehicle’s reference coordinate system view. You use the sensor’s extrinsic matrix, made up of a rotation matrix and translation vector, to convert your point cloud data to a WCS or global frame of reference.

Collectively, the translation vector and rotation matrix can be used to make up an extrinsic matrix, which can be used to convert data from a local coordinate system to a WCS. For example, your LiDAR extrinsic matrix may be composed as follows, where $R$ is the rotation matrix and $T$ is the translation vector:

\[
\text{LiDAR\_extrinsic} = [R\ T;0\ 0\ 0\ 1]
\]

For example, the autonomous driving KITTI dataset includes a rotation matrix and translation vector for the LiDAR extrinsic transformation matrix for each frame. The pykitti python module can be used for loading the KITTI data, and in the dataset `dataset.oxts[i].T_w_imu` gives the LiDAR extrinsic transform for the $i$th frame which can be multiplied with points in that frame to convert them to a world frame - \(\text{np.matmul(lidar\_transform\_matrix, points)}\). Multiplying a point in LiDAR frame with a LiDAR extrinsic matrix transforms it into world coordinates. Multiplying a point in the world frame with the camera extrinsic matrix gives the point coordinates in the camera’s frame of reference.

The following code example demonstrates how you can convert point cloud frames from the KITTI dataset into a WCS:

```python
import pykitti
import numpy as np

basedir = '/Users/nameofuser/kitti-data'
date = '2011_09_26'
drive = '0079'

# The 'frames' argument is optional - default: None, which loads the whole dataset.
# Calibration, timestamps, and IMU data are read automatically.
# Camera and velodyne data are available via properties that create generators
# when accessed, or through getter methods that provide random access.
data = pykitti.raw(basedir, date, drive, frames=range(0, 50, 5))

# i is frame number
i = 0

# lidar extrinsic for the ith frame
lidar_extrinsic_matrix = data.oxts[i].T_w_imu

# velodyne raw point cloud in lidar scanners own coordinate system
points = data.get_velo(i)

# transform points from lidar to global frame using lidar_extrinsic_matrix
def generate_transformed_pcd_from_point_cloud(points, lidar_extrinsic_matrix):
    tps = []
    for point in points:
        transformed_points = np.matmul(lidar_extrinsic_matrix, np.array([point[0],
                                                                        point[1],
                                                                        point[2],
                                                                        1], dtype=np.float32)).tolist()
        if len(point) > 3 and point[3] is not None:
            tps.append([transformed_points[0][0], transformed_points[1][0],
                        transformed_points[2][0], point[3]])
    return tps

def customer_transforms_points_from_lidar_to_global_frame(points, lidar_extrinsic_matrix):
    transformed_pcl = generate_transformed_pcd_from_point_cloud(points, lidar_extrinsic_matrix)
```

367
Sensor Fusion

Ground Truth supports sensor fusion of point cloud data with up to 8 video camera inputs. This feature allows human labellers to view the 3D point cloud frame side-by-side with the synchronized video frame. In addition to providing more visual context for labeling, sensor fusion allows workers to adjust annotations in the 3D scene and in 2D images and the adjustment are projected into the other view. The following video demonstrates a 3D point cloud labeling job with LiDAR and camera sensor fusion.
For best results, when using sensor fusion, your point cloud should be in a WCS. Ground Truth uses your sensor (such as LiDAR), camera, and ego vehicle pose information to compute extrinsic and intrinsic matrices for sensor fusion.

**Extrinsic Matrix**

Ground Truth uses sensor (such as LiDAR) extrinsic and camera extrinsic and intrinsic matrices to project objects to and from the point cloud data's frame of reference to the camera's frame of reference.

For example, in order to project a label from the 3D point cloud to camera image plane, Ground Truth transforms 3D points from LiDAR's own coordinate system to the camera's coordinate system. This is typically done by first transforming 3D points from LiDAR's own coordinate system to a world coordinate system (or a global reference frame) using the LiDAR extrinsic matrix. Ground Truth then uses the camera inverse extrinsic (which converts points from a global frame of reference to the camera's frame of reference) to transform the 3D points from world coordinate system obtained in previous step into the camera image plane. The LiDAR extrinsic matrix can also be used to transform 3D data into a world coordinate system. If your 3D data is already transformed into world coordinate system then the first transformation doesn't have any impact on label translation, and label translation only depends on the camera inverse extrinsic. A view matrix is used to visualize projected labels. To learn more about these transformations and the view matrix, see Ground Truth Sensor Fusion Transformations (p. 374).

Ground Truth computes these extrinsic matrices by using LiDAR and camera pose data that you provide: heading (in quaternions: qx, qy, qz, and qw) and position (x, y, z). For the vehicle, typically the heading and position are described in vehicle's reference frame in a world coordinate system and are called a ego vehicle pose. For each camera extrinsic, you can add pose information for that camera. For more information, see Pose (p. 371).

**Intrinsic Matrix**

Ground Truth use the camera extrinsic and intrinsic matrices to compute view metrics to transform labels to and from the 3D scene to camera images. Ground Truth computes the camera intrinsic matrix using camera focal length (fx, fy) and optical center coordinates (cx, cy) that you provide. For more information, see Intrinsic and Distortion (p. 374).

**Image Distortion**

Image distortion can occur for a variety of reasons. For example, images may be distorted due to barrel or fish-eye effects. Ground Truth uses intrinsic parameters along with distortion co-efficient to undistort images you provide when creating 3D point cloud labeling jobs. If a camera image is already been undistorted, all distortion coefficients should be set to 0.

For more information about the transformations Ground Truth performs to undistort images, see Camera Calibrations: Extrinsic, Intrinsic and Distortion (p. 374).

**Ego Vehicle**

To collect data for autonomous driving applications, the measurements used to generate point cloud data and are taken from sensors mounted on a vehicle, or the ego vehicle. To project label adjustments to and from the 3D scene and 2D images, Ground Truth needs your ego vehicle pose in a world coordinate system. The ego vehicle pose is comprised of position coordinates and orientation quaternion.

Ground Truth uses your ego vehicle pose to compute rotation and transformations matrices. Rotations in 3 dimensions can be represented by a sequence of 3 rotations around a sequence of axes. In theory, any three axes spanning the 3D Euclidean space are enough. In practice, the axes of rotation are chosen to be the basis vectors. The three rotations are expected to be in a global frame of reference (extrinsic). Ground Truth does not a support body centered frame of reference (intrinsic) which is attached to, and moves with, the object under rotation. To track objects, Ground Truth needs to measure from a global reference where all vehicles are moving. When using Ground Truth 3D point cloud labeling jobs, z specifies the axis of rotation (extrinsic rotation) and yaw Euler angles are in radians (rotation angle).
Pose

Ground Truth uses pose information for 3D visualizations and sensor fusion. Pose information you input through your manifest file is used to compute extrinsic matrices. If you already have an extrinsic matrix, you can use it to extract sensor and camera pose data.

For example in the autonomous driving KITTI dataset, the `pykitti` python module can be used for loading the KITTI data. In the dataset `dataset.oxts[i].T_w_imu` gives the LiDAR extrinsic transform for the $i^{th}$ frame and it can be multiplied with the points to get them in a world frame - `matmul(lidar_transform_matrix, points)`. This transform can be converted into position (translation vector) and heading (in quaternion) of LiDAR for the input manifest file JSON format. Camera extrinsic transform for `cam0` in $i^{th}$ frame can be calculated by `inv(matmul(dataset.calib.T_cam0_velo, inv(dataset.oxts[i].T_w_imu)))` and this can be converted into heading and position for `cam0`.

```python
import numpy
rotation = [[ 9.96714314e-01, -8.09890350e-02,  1.16333982e-03],
            [ 8.09967396e-02,  9.9661051e-01, -1.03090934e-02],
            [-3.24531964e-04,  1.03694477e-02,  9.99946183e-01]]
origin= [1.71104606e+00,
        5.80000039e-01,
        9.43144935e-01]
from scipy.spatial.transform import Rotation as R
# position is the origin
position = origin
r = R.from_matrix(np.asarray(rotation))
# heading in WCS using scipy
heading = r.as_quat()
print(f"pose:{position}\nheading: {heading}\n")
```

Position

In the input manifest file, `position` refers to the position of the sensor with respect to a world frame. If you are unable to put the device position in a world coordinate system, you can use LiDAR data with local coordinates. Similarly, for mounted video cameras you can specify the position and heading in a world coordinate system. For camera, if you do not have position information, please use (0, 0, 0).

The following are the fields in the position object:

1. $x$ (float) – $x$ coordinate of ego vehicle, sensor, or camera position in meters.
2. $y$ (float) – $y$ coordinate of ego vehicle, sensor, or camera position in meters.
3. $z$ (float) – $z$ coordinate of ego vehicle, sensor, or camera position in meters.

The following is an example of a `position` JSON object:

```json
{
   "position": {
      "y": -152.77584902657554,
      "x": 311.21505956090624,
      "z": -10.854137529636024
   }
}
```
**Heading**

In the input manifest file, `heading` is an object that represents the orientation of a device with respect to world frame. Heading values should be in quaternion. A quaternion is a representation of the orientation consistent with geodesic spherical properties. If you are unable to put the sensor heading in world coordinates, please use the identity quaternion \((q_x = 0, q_y = 0, q_z = 0, q_w = 1)\). Similarly, for cameras, specify the heading in quaternions. If you are unable to obtain extrinsic camera calibration parameters, please also use the identity quaternion.

Fields in `heading` object are as follows:

1. `qx` (float) - x component of ego vehicle, sensor, or camera orientation.
2. `qy` (float) - y component of ego vehicle, sensor, or camera orientation.
3. `qz` (float) - z component of ego vehicle, sensor, or camera orientation.
4. `qw` (float) - w component of ego vehicle, sensor, or camera orientation.

The following is an example of a `heading` JSON object:

```json
{
   "heading": {
      "qx": 0.034278837280808494,
      "qy": -0.7046155108831117,
      "qz": 0.7070617895701465,
      "qw": -0.04904659893885366
   }
}
```

To learn more, see [Compute Orientation Quaternions and Position (p. 372)](#).

### Compute Orientation Quaternions and Position

Ground Truth requires that all orientation, or heading, data be given in quaternions. A quaternion is a representation of the orientation consistent with geodesic spherical properties that can be used to approximate of rotation. Compared to Euler angles they are simpler to compose and avoid the problem of gimbal lock. Compared to rotation matrices they are more compact, more numerically stable, and more efficient.

You can compute quaternions from a rotation matrix or a transformation matrix.

If you have a rotation matrix (made up of the axis rotations) and translation vector (or origin) in world coordinate system instead of a single 4x4 rigid transformation matrix, then you can directly use the rotation matrix and translation vector to compute quaternions. Libraries like `scipy` and `pyquaternion` can help. The following code-block shows an example using these libraries to compute quaternion from a rotation matrix.

```python
import numpy
rotation = [[ 9.96714314e-01, -8.09980350e-02,  1.16333982e-03],
            [ 8.09967396e-02,  9.96661051e-01, -1.03090934e-02],
            [-3.24531964e-04,  1.03694477e-02,  9.99946183e-01]]
origin = [1.71104606e+00, 5.80000039e-01, 9.43144935e-01]
from scipy.spatial.transform import Rotation as R
# position is the origin
position = origin
```
A UI tool like 3D Rotation Converter can also be useful.

If you have a 4x4 extrinsic transformation matrix, note that the transformation matrix is in the form $[\mathbf{R} \; \mathbf{T} ; 0 \; 0 \; 0 \; 1]$ where $\mathbf{R}$ is the rotation matrix and $\mathbf{T}$ is the origin translation vector. That means you can extract rotation matrix and translation vector from the transformation matrix as follows.

```python
r = R.from_matrix(np.asarray(rotation))
# heading in WCS using scipy
heading = r.as_quat()
print(f"position:{position}\nheading: {heading}"
```

With your own setup, you can compute an extrinsic transformation matrix using the GPS/IMU position and orientation (latitude, longitude, altitude and roll, pitch, yaw) with respect to the LiDAR sensor on the ego vehicle. For example, you can compute pose from KITTI raw data using `pose = convertOxtsToPose(oxts)` to transform the oxts data into a local euclidean poses, specified by 4x4 rigid transformation matrices. You can then transform this pose transformation matrix to a global reference frame using the reference frames transformation matrix in the world coordinate system.

```python
struct Quaternion
{
    double w, x, y, z;
};

Quaternion ToQuaternion(double yaw, double pitch, double roll) // yaw (Z), pitch (Y), roll (X)
{
    // Abbreviations for the various angular functions
    double cy = cos(yaw * 0.5);
    double sy = sin(yaw * 0.5);
    double cp = cos(pitch * 0.5);
    double sp = sin(pitch * 0.5);
    double cr = cos(roll * 0.5);
    double sr = sin(roll * 0.5);

    Quaternion q;
    q.w = cr * cp * cy + sr * sp * sy;
    q.x = sr * cp * cy - cr * sp * sy;
    q.y = cr * sp * cy + sr * cp * sy;
    q.z = cr * cp * sy - sr * sp * cy;

    return q;
}
Ground Truth Sensor Fusion Transformations

The following sections go into greater detail about the Ground Truth sensor fusion transformations that are performed using the pose data you provide.

**LiDAR Extrinsic**

In order to project to and from a 3D LiDAR scene to a 2D camera image, Ground Truth computes the rigid transformation projection metrics using the ego vehicle pose and heading. Ground Truth computes rotation and translation of a world coordinates into the 3D plane by doing a simple sequence of rotations and translation.

Ground Truth computes rotation metrics using the heading quaternions as follows:

\[ M = \begin{pmatrix} 
1 - 2y^2 - 2z^2 & 2xy + 2zw & 2xz - 2yw \\
2xy - 2zw & 1 - 2x^2 - 2z^2 & 2yz + 2xw \\
xz + 2yw & 2yz - 2xw & 1 - 2x^2 - 2y^2 
\end{pmatrix} \]

Here, \([x, y, z, w]\) corresponds to parameters in the heading JSON object, \([qx, qy, qz, qw]\). Ground Truth computes the translation column vector as \( T = [\text{poseX}, \text{poseY}, \text{poseZ}] \). Then the extrinsic metrics is simply as follows:

\[
\text{LiDAR_extrinsic} = [R \ T; 0 \ 0 \ 0 \ 1]
\]

**Camera Calibrations: Extrinsic, Intrinsic and Distortion**

*Geometric camera calibration*, also referred to as *camera resectioning*, estimates the parameters of a lens and image sensor of an image or video camera. You can use these parameters to correct for lens distortion, measure the size of an object in world units, or determine the location of the camera in the scene. Camera parameters include intrinsics and distortion coefficients.

**Camera Extrinsic**

If the camera pose is given, then Ground Truth computes the camera extrinsic based on a rigid transformation from the 3D plane into the camera plane. The calculation is the same as the one used for the LiDAR Extrinsic (p. 374), except that Ground Truth uses camera pose (position and heading) and computes the inverse extrinsic.

\[
\text{camera_inverse_extrinsic} = \text{inv}([\text{Rc} \ Tc; 0 \ 0 \ 0 \ 1]) \ #\text{where Rc and Tc are camera pose components}
\]

**Intrinsic and Distortion**

Cameras have been around for a long-long time. However, with the introduction of the cheap *pinhole* cameras in the late 20th century, they became a common occurrence in our everyday life. Unfortunately, this cheapness comes with its price—significant distortion. Luckily, these are constants and with calibration and some remapping image distortion can be corrected. Furthermore, with calibration you can also determine the relationship between the camera’s natural units (pixels) and the real world units (for example, millimeters).

*Radial distortion* occurs when light rays bend more near the edges of a lens than they do at its optical center. The smaller the lens, the greater the distortion. The presence of the radial distortion manifests in form of the *barrel* or *fish-eye* effect and Ground Truth uses Formula 1 to undistort it.

**Formula 1:**

\[
x_{\text{corrected}} = x(1 + k_1r^2 + k_2r^4 + k_3r^6) \\
y_{\text{corrected}} = y(1 + k_1r^2 + k_2r^4 + k_3r^6)
\]
Tangential distortion occurs because the lenses used to take the images are not perfectly parallel to the imaging plane. This can be corrected with Formula 2.

**Formula 2:**

\[
\begin{align*}
    x_{corrected} &= x + [2p_1xy + p_2(r^2 + 2x^2)] \\
    y_{corrected} &= y + [p_1(r^2 + 2y^2) + 2p_2xy]
\end{align*}
\]

In the input manifest file, you can provide distortion coefficients and Ground Truth will undistort your images. All distortion coefficients are floats.

- \(k_1, k_2, k_3, k_4\) – Radial distortion coefficients. Supported for both fisheye and pinhole camera models.
- \(p_1, p_2\) – Tangential distortion coefficients. Supported for pinhole camera models.

If images are already undistorted, all distortion coefficients should be 0 in your input manifest.

In order to correctly reconstruct the corrected image, Ground Truth does a unit conversion of the images based on focal lengths. If a common focal length is used with a given aspect ratio for both axes, such as 1, in the upper formula we will have a single focal length. The matrix containing these four parameters is referred to as the **camera intrinsic calibration matrix**.

\[
\begin{pmatrix}
    x \\
    y \\
    w
\end{pmatrix} = \begin{pmatrix}
    f_x & 0 & c_x \\
    0 & f_y & c_y \\
    0 & 0 & 1
\end{pmatrix} \begin{pmatrix}
    X \\
    Y \\
    Z
\end{pmatrix}
\]

While the distortion coefficients are the same regardless of the camera resolutions used, these should be scaled with the current resolution from the calibrated resolution.

The following are float values.

- \(f_x\) - focal length in x direction.
- \(f_y\) - focal length in y direction.
- \(c_x\) - x coordinate of principal point.
- \(c_y\) - y coordinate of principal point.

Ground Truth use the camera extrinsic and camera intrinsic to compute view metrics as shown in the following code block to transform labels between the 3D scene and 2D images.

```python
def generate_view_matrix(intrinsic_matrix, extrinsic_matrix):
    intrinsic_matrix = np.c_[intrinsic_matrix, np.zeros(3)]
    view_matrix = np.matmul(intrinsic_matrix, extrinsic_matrix)
    view_matrix = np.insert(view_matrix, 2, np.array((0, 0, 0, 1)), 0)
    return view_matrix
```

### Video Frame Input Data

When you create a video frame object detection or object tracking labeling job, you can choose video files (MP4 files) or video frames for input data. All worker tasks are created using video frames, so if you choose video files, use the Ground Truth frame extraction tool to extract video frames (images) from your video files.
For both of these options, you can use the **Automated data setup** option in the Ground Truth section of the Amazon SageMaker console to set up a connection between Ground Truth and your input data in Amazon S3 so that Ground Truth knows where to look for your input data when creating your labeling tasks. This creates and stores an input manifest file in your Amazon S3 input dataset location. To learn more, see Automated Video Frame Input Data Setup (p. 378).

Alternatively, you can manually create sequence files for each sequence of video frames that you want labeled and provide the Amazon S3 location of an input manifest file that references each of these sequences files using the `source-ref` key. To learn more, see Create a Video Frame Input Manifest File (p. 379).

**Topics**

- Choose Video Files or Video Frames for Input Data (p. 376)
- Input Data Setup (p. 377)

### Choose Video Files or Video Frames for Input Data

When you create a video frame object detection or object tracking labeling job, you can provide a sequence of video frames (images) or you can use the Amazon SageMaker console to have Ground Truth automatically extract video frames from your video files. Use the following sections to learn more about these options.

#### Provide Video Frames

Video frames are sequences of images extracted from a video file. You can create a Ground Truth labeling job to have workers label multiple sequences of video frames. Each sequence is made up of images extracted from a single video.

To create a labeling job using video frame sequences, you must store each sequence using a unique key name prefix in Amazon S3. In the Amazon S3 console, key name prefixes are folders. So in the Amazon S3 console, each sequence of video frames must be located in its own folder in Amazon S3.

For example, if you have two sequences of video frames, you might use the key name prefixes `sequence1/` and `sequence2/` to identify your sequences. In this example, your sequences may be located in `s3://DOC-EXAMPLE-BUCKET/video-frames/sequence1/` and `s3://DOC-EXAMPLE-BUCKET/video-frames/sequence2/`.

If you are using the Ground Truth console to create an input manifest file, all of the sequence key name prefixes should be in the same location in Amazon S3. For example, in the Amazon S3 console, each sequence could be in a folder in `s3://DOC-EXAMPLE-BUCKET/video-frames/`. In this example, your first sequence of video frames (images) may be located in `s3://DOC-EXAMPLE-BUCKET/video-frames/sequence1/` and your second sequence may be located in `s3://DOC-EXAMPLE-BUCKET/video-frames/sequence2/`.

**Important**

Even if you only have a single sequence of video frames that you want workers to label, that sequence must have a key name prefix in Amazon S3. If you are using the Amazon S3 console, this means that your sequence is located in a folder. It cannot be located in the root of your S3 bucket.

When creating worker tasks using sequences of video frames, Ground Truth uses one sequence per task. In each task, Ground Truth orders your video frames using UTF-8 binary order.

For example, video frames might be in the following order in Amazon S3:

```
[0001.jpg, 0002.jpg, 0003.jpg, ..., 0011.jpg]
```
They are arranged in the same order in the worker's task: 0001.jpg, 0002.jpg, 0003.jpg, ..., 0011.jpg.

Frames might also be ordered using a naming convention like the following:

```
[frame1.jpg, frame2.jpg, ..., frame11.jpg]
```

In this case, frame10.jpg and frame11.jpg come before frame2.jpg in the worker task. Your worker sees your video frames in the following order: frame1.jpg, frame10.jpg, frame11.jpg, frame2.jpg, ..., frame9.jpg.

**Provide Video Files**

You can use the Ground Truth frame splitting feature when creating a new labeling job in the console to extract video frames from video files (MP4 files). A series of video frames extracted from a single video file is referred to as a *sequence of video frames*.

You can either have Ground Truth automatically extract all frames, up to 2,000, from the video, or you can specify a frequency for frame extraction. For example, you can have Ground Truth extract every 10th frame from your videos.

To use the video frame extraction tool, see Automated Video Frame Input Data Setup (p. 378).

When all of your video frames have been successfully extracted from your videos, you will see the following in your S3 input dataset location:

- A key name prefix (a folder in the Amazon S3 console) named after each video. Each of these prefixes leads to:
  - A sequence of video frames extracted from the video used to name that prefix.
  - A sequence file used to identify all of the images that make up that sequence.
  - An input manifest file with a .manifest extension. This identifies all of the sequence files that will be used to create your labeling job.

All of the frames extracted from a single video file are used for a labeling task. If you extract video frames from multiple video files, multiple tasks are created for your labeling job, one for each sequence of video frames.

Ground Truth stores each sequence of video frames that it extracts in your Amazon S3 location for input datasets using a unique key name prefix. In the Amazon S3 console, key name prefixes are folders.

**Input Data Setup**

When you create a video frame labeling job, you need to let Ground Truth know where to look for your input data. You can do this in one of two ways:

- You can store your input data in Amazon S3 and have Ground Truth automatically detect the input dataset used for your labeling job. See Automated Video Frame Input Data Setup (p. 378) to learn more about this option.
- You can create an input manifest file and sequence files and upload them to Amazon S3. See Manual Input Data Setup (p. 379) to learn more about this option.

**Topics**

- Automated Video Frame Input Data Setup (p. 378)
- Manual Input Data Setup (p. 379)
Automated Video Frame Input Data Setup

You can use the Ground Truth automated data setup to automatically detect video files in your Amazon S3 bucket and extract video frames from those files. To learn how, see Provide Video Files (p. 377).

If you already have video frames in Amazon S3, you can use the automated data setup to use these video frames in your labeling job. For this option, all video frames from a single video must be stored using a unique prefix. To learn about the requirements to use this option, see Provide Video Frames (p. 376).

Select one of the following sections to learn how to set up your automatic input dataset connection with Ground Truth.

Provide Video Files and Extract Frames

Use the following procedure to connect your video files with Ground Truth and automatically extract video frames from those files for video frame object detection and object tracking labeling jobs.

Make sure your video files are stored in an Amazon S3 bucket in the same AWS Region that you perform the automated data setup in.

Automatically connect your video files in Amazon S3 with Ground Truth and extract video frames:


   Your input and output S3 buckets must be located in the same AWS Region that you create your labeling job in. This link puts you in the North Virginia (us-east-1) AWS Region. If your input data is in an Amazon S3 bucket in another Region, switch to that Region. To change your AWS Region, on the navigation bar, choose the name of the currently displayed Region.

2. Select Create labeling job.

3. Enter a Job name.

4. In the section Input data setup, select Automated data setup.

5. Enter an Amazon S3 URI for S3 location for input datasets. An S3 URI looks like the following: s3://DOC-EXAMPLE-BUCKET/path-to-files/. This URI should point to the Amazon S3 location where your video files are stored.

6. Specify your S3 location for output datasets. This is where your output data is stored. You can choose to store your output data in the Same location as input dataset or Specify a new location and entering the S3 URI of the location that you want to store your output data.

7. Choose Video Files for your Data type using the dropdown list.

8. Choose Yes, extract frames for object tracking and detection tasks.


   - When you choose Use all frames extracted from the video to create a labeling task, Ground Truth extracts all frames from each video in your S3 location for input datasets, up to 2,000 frames. If a video in your input dataset contains more than 2,000 frames, the first 2,000 are extracted and used for that labeling task.
   - When you choose Use every $x$ frame from a video to create a labeling task, Ground Truth extracts every $x^{th}$ frame from each video in your S3 location for input datasets.

   For example, if your video is 2 seconds long, and has a frame rate of 30 frames per second, there are 60 frames in your video. If you specify 10 here, Ground Truth extracts every $10^{th}$ frame from your video. This means the $1^{st}$, $10^{th}$, $20^{th}$, $30^{th}$, $40^{th}$, $50^{th}$, and $60^{th}$ frames are extracted.

10. Choose or create an IAM execution role. Make sure that this role has permission to access your Amazon S3 locations for input and output data specified in steps 5 and 6.
11. Select **Complete data setup**.

### Provide Video Frames

Use the following procedure to connect your sequences of video frames with Ground Truth for video frame object detection and object tracking labeling jobs.

Make sure your video frames are stored in an Amazon S3 bucket in the same AWS Region that you perform the automated data setup in. Each sequence of video frames should have a unique prefix. For example, if you have two sequences stored in `s3://DOCD-EXAMPLE-BUCKET/video-frames/sequences/`, each should have a unique prefix like `sequence1` and `sequence2` and should both be located directly under the `/sequences/` prefix. In the example above, the locations of these two sequences is: `s3://DOCD-EXAMPLE-BUCKET/video-frames/sequences/sequence1/` and `s3://DOCD-EXAMPLE-BUCKET/video-frames/sequences/sequence2/`.

**Automatically connect your video frame in Amazon S3 with Ground Truth:**

1. Navigate to the **Create labeling job** page in the Amazon SageMaker console: [https://console.aws.amazon.com/sagemaker/groundtruth](https://console.aws.amazon.com/sagemaker/groundtruth).

   Your input and output S3 buckets must be located in the same AWS Region that you create your labeling job in. This link puts you in the North Virginia (us-east-1) AWS Region. If your input data is in an Amazon S3 bucket in another Region, switch to that Region. To change your AWS Region, on the navigation bar, choose the name of the currently displayed Region.

2. Select **Create labeling job**.

3. Enter a **Job name**.

4. In the section **Input data setup**, select **Automated data setup**.

5. Enter an Amazon S3 URI for **S3 location for input datasets**.

   This should be the Amazon S3 location where your sequences are stored. For example, if you have two sequences stored in `s3://DOCD-EXAMPLE-BUCKET/video-frames/sequences/sequence1/`, `s3://DOCD-EXAMPLE-BUCKET/video-frames/sequences/sequence2/`, enter `s3://DOCD-EXAMPLE-BUCKET/video-frames/sequences/` here.

6. Specify your **S3 location for output datasets**. This is where your output data is stored. You can choose to store your output data in the **Same location as input dataset** or **Specify a new location** and entering the S3 URI of the location that you want to store your output data.

7. Choose **Video frames** for your **Data type** using the dropdown list.

8. Choose or create an IAM execution role. Make sure that this role has permission to access your Amazon S3 locations for input and output data specified in steps 5 and 6.

9. Select **Complete data setup**.

These procedures will create an input manifest in the Amazon S3 location for input datasets that you specified in step 5. If you are creating a labeling job using the SageMaker API or, AWS CLI, or an AWS SDK, use the Amazon S3 URI for this input manifest file as input to the parameter `ManifestS3Uri`.

### Manual Input Data Setup

Choose the manual data setup option if you have created sequence files for each of your video frame sequences, and a manifest file listing references to those sequences files.

**Create a Video Frame Input Manifest File**

Ground Truth uses the input manifest file to identify the location of your input dataset when creating labeling tasks. For video frame object detection and object tracking labeling jobs, each line in the input manifest file identifies the location of a video frame sequence file. Each sequence file identifies the images included in a single sequence of video frames.
Use this page to learn how to create a video frame sequence file and an input manifest file for video frame object tracking and object detection labeling jobs.

If you want Ground Truth to automatically generate your sequence files and input manifest file, see Automated Video Frame Input Data Setup (p. 378).

Create a Video Frame Sequence Input Manifest

In the video frame sequence input manifest file, each line in the manifest is a JSON object, with a "source-ref" key that references a sequence file. Each sequence file identifies the location of a sequence of video frames. This is the manifest file formatting required for all video frame labeling jobs.

The following example demonstrates the syntax used for an input manifest file:

```json
{"source-ref": "s3://DOC-EXAMPLE-BUCKET/example-folder/seq1.json"}
{"source-ref": "s3://DOC-EXAMPLE-BUCKET/example-folder/seq2.json"}
```

Create a Video Frame Sequence File

The data for each sequence of video frames needs to be stored in a JSON data object. The following is an example of the format you use for a sequence file. Information about each frame is included as a JSON object and is listed in the frames list. The following JSON has been expanded for readability.

```json
{
  "seq-no": 1,
  "prefix": "s3://mybucket/prefix/video1/",
  "number-of-frames": 3,
  "frames": [
    {
      "frame-no": 1, "unix-timestamp": 1566861644, "frame": "frame0001.jpg" },
    {
      "frame-no": 2, "unix-timestamp": 1566861644, "frame": "frame0002.jpg" },
    {
      "frame-no": 3, "unix-timestamp": 1566861644, "frame": "frame0003.jpg" }
  ]
}
```

The following table provides details about the parameters shown in the this code example.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Required</th>
<th>Accepted Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>seq-no</td>
<td>Yes</td>
<td>Integer</td>
<td>The ordered number of the sequence.</td>
</tr>
<tr>
<td>prefix</td>
<td>Yes</td>
<td>String</td>
<td>The Amazon S3 location where the sequence files are located.</td>
</tr>
</tbody>
</table>
|                       |          | Accepted Values | The prefix must end with a forward slash: /.
<p>|                       |          | s3://&lt;bucket-name&gt;/&lt;prefix&gt;/ |                                                                         |
| number-of-frames      | Yes      | Integer         | The total number of frames included in the sequence file. This number must match the total number of frames listed in the frames parameter in the next row. |</p>
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Required</th>
<th>Accepted Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>frames</td>
<td>Yes</td>
<td>List of JSON objects</td>
<td>A list of frame data. The length of the list must equal number-of-frames. In the worker UI, frames in a sequence are ordered in UTF-8 binary order. To learn more about this ordering, see Provide Video Frames (p. 376).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>frame-no, frame</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>unix-timestamp</td>
<td></td>
</tr>
<tr>
<td>frame-no</td>
<td>Yes</td>
<td>String</td>
<td>The frame order number. This will determine the order of a frame in the sequence.</td>
</tr>
<tr>
<td>unix-timestamp</td>
<td>No</td>
<td>Integer</td>
<td>The unix timestamp of a frame. The number of seconds since January 1st, 1970 until the UTC time when the frame was captured.</td>
</tr>
<tr>
<td>frame</td>
<td>Yes</td>
<td>String</td>
<td>The name of a video frame image file.</td>
</tr>
</tbody>
</table>

**Output Data**

The output from a labeling job is placed in the location that you specified in the console or in the call to the CreateLabelingJob operation.

Each line in the output data file is identical to the manifest file with the addition of an attribute and value for the label assigned to the input object. The attribute name for the value is defined in the console or in the call to the CreateLabelingJob operation. You can't use `metadata` in the label attribute name. If you are running an image semantic segmentation, 3D point cloud semantic segmentation, or 3D point cloud object tracking job, the label attribute must end with `-ref`. For any other type of job, the attribute name can't end with `-ref`.

The output of the labeling job is the value of the key-value pair with the label. The label and the value overwrites any existing JSON data in the input file with the new value.

For example, the following is the output from an image classification labeling job where the input data files were stored in an Amazon S3 `AWSDOC-EXAMPLE-BUCKET` and the label attribute name was defined as `sport`. In this example the JSON object is formatted for readability, in the actual output file the JSON object is on a single line. For more information about the data format, see JSON Lines.

```json
{
    "source-ref": "s3://AWSDOC-EXAMPLE-BUCKET/image_example.png",
    "sport": 0,
    "sport-metadata":
    {
        "class-name": "football",
        "confidence": 0.00,
        "type": "groundtruth/image-classification",
        "job-name": "identify-sport",
```
The value of the label can be any valid JSON. In this case the label's value is the index of the class in the classification list. Other job types, such as bounding box, have more complex values.

Any key-value pair in the input manifest file other than the label attribute is unchanged in the output file. You can use this to pass data to your application.

The output from a labeling job can be used as the input to another labeling job. You can use this when you are chaining together labeling jobs. For example, you can send one labeling job to determine the sport that is being played. Then you send another using the same data to determine if the sport is being played indoors or outdoors. By using the output data from the first job as the manifest for the second job, you can consolidate the results of the two jobs into one output file for easier processing by your applications.

The output data file is written to the output location periodically while the job is in progress. These intermediate files contain one line for each line in the manifest file. If an object is labeled, the label is included. If the object hasn't been labeled, it is written to the intermediate output file identically to the manifest file.

Output Directories

Ground Truth creates several directories in your Amazon S3 output path. These directories contain the results of your labeling job and other artifacts of the job. The top-level directory for a labeling job is given the same name as your labeling job; the output directories are placed beneath it. For example, if you named your labeling job `find-people`, your output would be in the following directories:

- `s3://AWSDOC-EXAMPLE-BUCKET/find-people/activelearning`
- `s3://AWSDOC-EXAMPLE-BUCKET/find-people/annotations`
- `s3://AWSDOC-EXAMPLE-BUCKET/find-people/inference`
- `s3://AWSDOC-EXAMPLE-BUCKET/find-people/manifests`
- `s3://AWSDOC-EXAMPLE-BUCKET/find-people/training`

Each directory contains the following output:

**Active Learning Directory**

The `activelearning` directory is only present when you are using automated data labeling. It contains the input and output validation set for automated data labeling, and the input and output folder for automatically labeled data.

**Annotations Directory**

The `annotations` directory contains all of the annotations made by the workforce. These are the responses from individual workers that have not been consolidated into a single label for the data object. There are three subdirectories in the `annotations` directory.

The first, `worker-response`, contains the responses from individual workers. This contains a subdirectory for each iteration, which in turn contains a subdirectory for each data object in that iteration. The annotation for each data object is stored in a timestamped JSON file that contains the annotation made by a single worker, and if you use a private workforce, metadata about that worker. To learn more about this metadata, see Worker Metadata (p. 383). There may be more than one annotation for each data object in this directory, depending on how many workers you want to annotate each object.
The second, consolidated-annotation, contains information required to consolidate the annotations in the current batch into labels for your data objects.

The third, intermediate, contains the output manifest for the current batch with any completed labels. This file is updated as the label for each data object is completed.

**Inference Directory**

The inference directory is only present when you are using automated data labeling. This directory contains the input and output files for the SageMaker batch transform used while labeling data objects.

**Manifest Directory**

The manifest directory contains the output manifest from your labeling job. There is one subdirectory in the manifest directory, output. The output directory contains the output manifest file for your labeling job. The file is named `output.manifest`.

**Training Directory**

The training directory is only present when you are using automated data labeling. This directory contains the input and output files used to train the automated data labeling model.

**Confidence Score**

When you have more than one worker annotate a single task, your label results from annotation consolidation. Ground Truth calculates a confidence score for each label. A confidence score is a number between 0 and 1 that indicates how confident Ground Truth is in the label. You can use the confidence score to compare labeled data objects to each other, and to identify the least or most confident labels.

You should not interpret the value of a confidence score as an absolute value, or compare confidence scores across labeling jobs. For example, if all of the confidence scores are between 0.98 and 0.998, you should only compare the data objects with each other and not rely on the high confidence scores.

You should not compare the confidence scores of human-labeled data objects and auto-labeled data objects. The confidence scores for humans are calculated using the annotation consolidation function for the task, while the confidence scores for automated labeling are calculated using a model that incorporates object features. The two models generally have different scales and average confidence.

For a bounding box labeling job, Ground Truth calculates a confidence score per box. You can compare confidence scores within one image or across images for the same labeling type (human or auto). You can't compare confidence scores across labeling jobs.

If a single worker annotates a task (NumberOfHumanWorkersPerDataObject is set to 1 or in the console, you enter 1 for **Number of workers per dataset object**), the confidence score is set to 0.00.

**Worker Metadata**

Ground Truth provides information that you can use to track individual workers in task output data. The following data is located in the directories under the **worker-response** located in the Annotations Directory (p. 382):

- The acceptanceTime is the time that the worker accepted the task. The format of this date and time stamp is `YYYY-MM-DDTHH:MM:SS.mmmZ` for the year (YYYY), month (MM), day (DD), hour (HH), minute (MM), second (SS) and millisecond (mmm). The date and time are separated by a T.
- The submissionTime is the time that the worker submitted their annotations using the Submit button. The format of this date and time stamp is `YYYY-MM-DDTHH:MM:SS.mmmZ` for the year (YYYY), month (MM), day (DD), hour (HH), minute (MM), second (SS) and millisecond (mmm). The date and time are separated by a T.
- timeSpentInSeconds reports the total time, in seconds, that a worker worked on that task. This is the submissionTime subtracted from the acceptanceTime.
The workerId is unique to each worker.

If you use a private workforce, in workerMetadata, you see the following.

- The identityProviderType is the service used to manage the private workforce.
- The issuer is the Cognito user pool or OIDC Identity Provider (IdP) issuer associated with the work team assigned to this human review task.
- A unique sub identifier refers to the worker. If you create a workforce using Amazon Cognito, you can retrieve details about this worker (such as the name or user name) using this ID using Amazon Cognito. To learn how, see Managing and Searching for User Accounts in Amazon Cognito Developer Guide.

The following is an example of the output you may see if you use Amazon Cognito to create a private workforce. This is identified in the identityProviderType.

```
"submissionTime": "2020-12-28T18:59:58Z",
"acceptanceTime": "2020-12-28T18:59:15Z",
"timeSpentInSeconds": 43,
"workerId": "a12b3cdefg4h5i67",
"workerMetadata": {
  "identityData": {
    "identityProviderType": "Cognito",
    "sub": "aaaaaaaa-bbbb-cccc-dddd-eeeeeeeeeee"
  }
}
```

The following is an example of the workerMetadata you may see if you use your own OIDC IdP to create a private workforce:

```
"workerId": "a12b3cdefg4h5i67",
"workerMetadata": {
  "identityData": {
    "identityProviderType": "Oidc",
    "issuer": "https://example-oidc-ipd.com/adfs",
    "sub": "aaaaaaaa-bbbb-cccc-dddd-eeeeeeeeeee"
  }
}
```

To learn more about using private workforces, see Use a Private Workforce (p. 430).

Output Metadata

The output from each job contains metadata about the label assigned to data objects. These elements are the same for all jobs with minor variations. The following example shows the metadata elements:

```
"confidence": 0.00,
"type": "groundtruth/image-classification",
"job-name": "identify-animal-species",
"human-annotated": "yes",
"creation-date": "2020-10-18T22:18:13.527256"
```

The elements have the following meaning:

- **confidence** – The confidence that Ground Truth has that the label is correct. For more information, see Confidence Score (p. 383).
- **type** – The type of classification job. For job types, see Built-in Task Types (p. 312).
- **job-name** – The name assigned to the job when it was created.
• **human-annotated** – Whether the data object was labeled by a human or by automated data labeling. For more information, see Automate Data Labeling (p. 406).

• **creation-date** – The date and time that the label was created.

## Classification Job Output

The following are sample outputs (output manifest files) from an image classification job and a text classification job. They include the label that Ground Truth assigned to the data object, the value for the label, and metadata that describes the label.

In addition to the standard metadata elements, the metadata for a classification job includes the text value of the label's class. For more information, see Image Classification Algorithm (p. 690).

The red, italicized text in the examples below depends on labeling job specifications and output data.

```json
{
  "source-ref": "s3://AWSDOC-EXAMPLE-BUCKET/example_image.jpg",
  "species": "0",
  "species-metadata":
  {
    "class-name": "dog",
    "confidence": 0.00,
    "type": "groundtruth/image-classification",
    "job-name": "identify-animal-species",
    "human-annotated": "yes",
    "creation-date": "2018-10-18T22:18:13.527256"
  }
}
```

```json
{
  "source": "The food was delicious",
  "mood": "1",
  "mood-metadata":
  {
    "class-name": "positive",
    "confidence": 0.8,
    "type": "groundtruth/text-classification",
    "job-name": "label-sentiment",
    "human-annotated": "yes",
    "creation-date": "2020-10-18T22:18:13.527256"
  }
}
```

## Multi-label Classification Job Output

The following are example output manifest files from a multi-label image classification job and a multi-label text classification job. They include the labels that Ground Truth assigned to the data object (for example, the image or piece of text) and metadata that describes the labels the worker saw when completing the labeling task.

The label attribute name parameter (for example, image-label-attribute-name) contains an array of all of the labels selected by at least one of the workers who completed this task. This array contains integer keys (for example, [1, 0, 8]) that correspond to the labels found in class-map. In the multi-label image classification example, bicycle, person, and clothing were selected by at least one of the workers who completed the labeling task for the image, exampleimage.jpg.

The confidence-map shows the confidence score that Ground Truth assigned to each label selected by a worker. To learn more about Ground Truth confidence scores, see Confidence Score (p. 383).
The red, italicized text in the examples below depends on labeling job specifications and output data.

The following is an example of a multi-label image classification output manifest file.

```
{
    "source-ref": "s3://AWSDOC-EXAMPLE-BUCKET/example_image.jpg",
    "image-label-attribute-name": [1, 0, 8],
    "image-label-attribute-name-metadata":
    { 
        "job-name": "labeling-job/image-label-attribute-name",
        "class-map":
        { 
            "1": "bicycle",
            "0": "person",
            "8": "clothing"
        },
        "human-annotated": "yes",
        "creation-date": "2020-02-27T21:36:25.000201",
        "confidence-map":
        { 
            "1": 0.95,
            "0": 0.77,
            "8": 0.2
        },
        "type": "groundtruth/image-classification-multilabel"
    }
}
```

The following is an example of a multi-label text classification output manifest file. In this example, approving, sad and critical were selected by at least one of the workers who completed the labeling task for the object exampletext.txt found in AWSDOC-EXAMPLE-BUCKET.

```
{
    "source-ref": "AWSDOC-EXAMPLE-BUCKET/text_file.txt",
    "text-label-attribute-name": [1, 0, 4],
    "text-label-attribute-name-metadata":
    { 
        "job-name": "labeling-job/text-label-attribute-name",
        "class-map":
        { 
            "1": "approving",
            "0": "sad",
            "4": "critical"
        },
        "human-annotated": "yes",
        "creation-date": "2020-02-20T21:36:25.000201",
        "confidence-map":
        { 
            "1": 0.95,
            "0": 0.77,
            "4": 0.2
        },
        "type": "groundtruth/text-classification-multilabel"
    }
}
```

**Bounding Box Job Output**

The following is sample output (output manifest file) from a bounding box job. For this task, three bounding boxes are returned. The label value contains information about the size of the image, and the location of the bounding boxes.

The `class_id` element is the index of the box's class in the list of available classes for the task. The `class-map` metadata element contains the text of the class.

The metadata has a separate confidence score for each bounding box. The metadata also includes the `class-map` element that maps the `class_id` to the text value of the class. For more information, see [Object Detection Algorithm (p. 770)](#).

The red, italicized text in the examples below depends on labeling job specifications and output data.
The output of a bounding box adjustment job looks like the following JSON. Note that the original JSON is kept intact and two new jobs are listed, each with “adjust-” prepended to the original attribute's name.

```json
{
    "source-ref": "s3://AWSDOC-EXAMPLE-BUCKET/example_image.png",
    "bounding-box": {
        "image_size": [{ "width": 500, "height": 400, "depth": 3 }],
        "annotations": [
            {"class_id": 0, "left": 111, "top": 134,
             "width": 61, "height": 128},
            {"class_id": 5, "left": 161, "top": 250,
             "width": 30, "height": 30},
            {"class_id": 5, "left": 20, "top": 20,
             "width": 30, "height": 30}
        ]
    },
    "bounding-box-metadata": {
        "objects": [
            {"confidence": 0.8},
            {"confidence": 0.9},
            {"confidence": 0.9}
        ],
        "class-map": {
            "0": "dog",
            "5": "bone"
        },
        "type": "groundtruth/object-detection",
        "human-annotated": "yes",
        "creation-date": "2018-10-18T22:18:13.527256",
        "job-name": "identify-dogs-and-toys"
    }
}
```
In this output, the job's type doesn't change, but an adjustment-status field is added. This field has the value of adjusted or unadjusted. If multiple workers have reviewed the object and at least one adjusted the label, the status is adjusted.

Named Entity Recognition

The following is an example output manifest file from a named entity recognition (NER) labeling task. For this task, seven entities are returned.

In the output manifest, the JSON object, annotations, includes a list of the labels (label categories) that you provided.

Worker responses are in a list named entities. Each entity in this list is a JSON object that contains a label value that matches one in the labels list, an integer startOffset value for labeled span's starting Unicode offset, and an integer endOffset value for the ending Unicode offset.

The metadata has a separate confidence score for each entity. If a single worker labeled each data object, the confidence value for each entity will be zero.

The red, italicized text in the examples below depends on labeling job inputs and worker responses.
Amazon SageMaker Developer Guide
Use Input and Output Data

"source": "Amazon SageMaker is a cloud machine-learning platform that was launched in November 2017. SageMaker enables developers to create, train, and deploy machine-learning (ML) models in the cloud. SageMaker also enables developers to deploy ML models on embedded systems and edge-devices",

"example-ner-labeling-job": {
  "annotations": {
    "labels": [
      {
        "label": "Date",
        "shortDisplayName": "dt"
      },
      {
        "label": "Verb",
        "shortDisplayName": "vb"
      },
      {
        "label": "Thing",
        "shortDisplayName": "tng"
      },
      {
        "label": "People",
        "shortDisplayName": "ppl"
      }
    ],
    "entities": [
      {
        "label": "Thing",
        "startOffset": 22,
        "endOffset": 53
      },
      {
        "label": "Thing",
        "startOffset": 269,
        "endOffset": 281
      },
      {
        "label": "Verb",
        "startOffset": 63,
        "endOffset": 71
      },
      {
        "label": "Verb",
        "startOffset": 228,
        "endOffset": 234
      },
      {
        "label": "Date",
        "startOffset": 75,
        "endOffset": 88
      },
      {
        "label": "People",
        "startOffset": 108,
        "endOffset": 118
      },
      {
        "label": "People",
        "startOffset": 214,
        "endOffset": 224
      }
    ]
  }
},

"example-ner-labeling-job-metadata": {
  "job-name": "labeling-job/example-ner-labeling-job",
  "type": "groundtruth/text-span"
Label Verification Job Output

The output (output manifest file) of a bounding box verification job looks different than the output of a bounding box annotation job. That's because the workers have a different type of task. They're not labeling objects, but evaluating the accuracy of prior labeling, making a judgment, and then providing that judgment and perhaps some comments.

If human workers are verifying or adjusting prior bounding box labels, the output of a verification job would look like the following JSON. The red, italicized text in the examples below depends on labeling job specifications and output data.

```json
{
  "source-ref": "s3://AWSDOC-EXAMPLE-BUCKET/image_example.png",
  "bounding-box":
  {
    "image_size": [{ "width": 500, "height": 400, "depth": 3} ],
    "annotations": [
      {"class_id": 0, "left": 111, "top": 134, "width": 61, "height": 128},
      {"class_id": 5, "left": 161, "top": 250, "width": 30, "height": 30},
      {"class_id": 5, "left": 20, "top": 20, "width": 30, "height": 30}
    ]
  },
  "bounding-box-metadata":
  {
    "objects": [
      {"confidence": 0.8},
      {"confidence": 0.9},
      {"confidence": 0.9}
    ],
    "class-map":
  }
}
```
Although the type on the original bounding box output was `groundtruth/object-detection`, the new type is `groundtruth/label-verification`. Also note that the `worker-feedback` array provides worker comments. If the worker doesn't provide comments, the empty fields are excluded during consolidation.

**Semantic Segmentation Job Output**

The following is the output manifest file from a semantic segmentation labeling job. The value of the label for this job is a reference to a PNG file in an Amazon S3 bucket.

In addition to the standard elements, the metadata for the label includes a color map that defines which color is used to label the image, the class name associated with the color, and the confidence score for each color. For more information, see [Semantic Segmentation Algorithm](p. 792).

The red, italicized text in the examples below depends on labeling job specifications and output data.
Confidence is scored on a per-image basis. Confidence scores are the same across all classes within an image.

The output of a semantic segmentation adjustment job looks similar to the following JSON.

```json
{
  "source-ref": "s3://AWSDOC-EXAMPLE-BUCKET/example_city_image.png",
  "city-streets-ref": "S3 bucket location",
  "city-streets-ref-metadata": {
    "internal-color-map": {
      "0": {
        "class-name": "BACKGROUND",
        "confidence": 0.9,
        "hex-color": "#ffffff"
      },
      "1": {
        "class-name": "buildings",
        "confidence": 0.9,
        "hex-color": "#2acf59"
      },
      "2": {
        "class-name": "road",
        "confidence": 0.9,
        "hex-color": "#f28333"
      }
    },
    "type": "groundtruth/semantic-segmentation",
    "human-annotated": "yes",
    "creation-date": "2018-10-18T22:18:13.527256",
    "job-name": "label-city-streets",
  },
  "adjusted-city-streets-ref": "s3://AWSDOC-EXAMPLE-BUCKET/example_city_image.png",
  "adjusted-city-streets-ref-metadata": {
    "internal-color-map": {
      "0": {
        "class-name": "BACKGROUND",
        "confidence": 0.9,
        "hex-color": "#ffffff"
      },
      "1": {
        "class-name": "buildings",
        "confidence": 0.9,
        "hex-color": "#2acf59"
      }
    },
    "type": "groundtruth/semantic-segmentation",
    "human-annotated": "yes",
    "creation-date": "2018-10-18T22:18:13.527256",
    "job-name": "label-city-streets",
  }
}
```
Video Frame Object Detection Output

The following is the output manifest file from a video frame object detection labeling job. The red, italicized text in the examples below depends on labeling job specifications and output data.

In addition to the standard elements, the metadata includes a class map that lists each class that has at least one label in the sequence. The metadata also includes job-name which is the name you assigned to the labeling job. For adjustment tasks, if one or more bounding boxes were modified, there is an adjustment-status parameter in the metadata for audit workflows that is set to adjusted.

```
{
  "source-ref": "s3://DOC-EXAMPLE-BUCKET/example-path/input-manifest.json",
  "CarObjectDetection-ref": "s3://AWSDOC-EXAMPLE-BUCKET/output/labeling-job-name/annotations/consolidated-annotation/output/0/SeqLabel.json",
  "CarObjectDetection-ref-metadata": {
    "class-map": {
      "0": "car",
      "1": "bus"
    },
    "job-name": "labeling-job/labeling-job-name",
    "human-annotated": "yes",
    "creation-date": "2020-05-15T08:01:16+0000",
    "type": "groundtruth/video-object-detection"
  }
}
```

Ground Truth creates one output sequence file for each sequence of video frames that was labeled. Each output sequence file contains the following:

- All annotations for all frames in a sequence in the detection-annotations list of JSON objects.
- For each frame that was annotated by a worker, the frame file name (frame), number (frame-no), a list of JSON objects containing annotations (annotations), and if applicable, frame-attributes. The name of this list is defined by the task type you use: polylines, polygons, keypoints, and for bounding boxes, annotations.

Each JSON object contains information about a single annotation and associated label. The following table outlines the parameters you'll see for each video frame task type.

<table>
<thead>
<tr>
<th>Task Type</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bounding Box</td>
<td>Box dimensions: height and width</td>
</tr>
<tr>
<td></td>
<td>Box top, left corner pixel location: top and left</td>
</tr>
</tbody>
</table>

393
### Task Type Parameters

<table>
<thead>
<tr>
<th>Task Type</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Keypoint</td>
<td>Keypoint vertices: { &quot;x&quot;: int, &quot;y&quot;: int }</td>
</tr>
<tr>
<td>Polygon</td>
<td>A list of polygon vertices: vertices</td>
</tr>
<tr>
<td></td>
<td>Polygon vertices: { &quot;x&quot;: int, &quot;y&quot;: int }</td>
</tr>
<tr>
<td></td>
<td>A polygon is a closed shape and so the first</td>
</tr>
<tr>
<td></td>
<td>point will also represent the last point.</td>
</tr>
<tr>
<td>Polyline</td>
<td>A list of polyline vertices: vertices</td>
</tr>
<tr>
<td></td>
<td>Polyline vertices: { &quot;x&quot;: int, &quot;y&quot;: int }</td>
</tr>
</tbody>
</table>

In addition to task type specific values, you will see the following in each JSON object:

- Values of any label-category-attributes that were specified for that label.
- The class-id of the box. Use the class-map in the output manifest file to see which label category this ID maps to.

The following is an example of a `SeqLabel.json` file from a bounding box video frame object detection labeling job. This file will be located under `s3://your-output-bucket/output-prefix/annotations/consolidated-annotation/output/annotation-number/`

```json
{
    "detection-annotations": [ {
        "annotations": [ {
            "height": 41,
            "width": 53,
            "top": 152,
            "left": 339,
            "class-id": "1",
            "label-category-attributes": { "occluded": "no", "size": "medium" }
        } ],
        "height": 24,
        "width": 37,
        "top": 148,
        "left": 183,
        "class-id": "0",
        "label-category-attributes": { "occluded": "no", "size": "medium" }
    } ],
    "frame-no": "0",
    "frame": "frame_0000.jpeg",
    "frame-attributes": { name: value, name: value }
},
{
    "annotations": [ {
        "height": 41,
        "width": 53,
        "top": 152,
        "left": 341,
        "class-id": "0",
```
Video Frame Object Tracking Output

The following is the output manifest file from a video frame object tracking labeling job. The red, italicized text in the examples below depends on labeling job specifications and output data.

In addition to the standard elements, the metadata includes a class map that lists each class that has at least one label in the sequence of frames. The metadata also includes job-name which is the name you assigned to the labeling job. For adjustment tasks, if one or more bounding boxes were modified, there is an adjustment-status parameter in the metadata for audit workflows that is set to adjusted.

Ground Truth creates one output sequence file for each sequence of video frames that was labeled. Each output sequence file contains the following:

- All annotations for all frames in a sequence in the tracking-annotations list of JSON objects.
- For each frame that was annotated by a worker, the frame (frame), number (frame-no), a list of JSON objects containing annotations (annotations), and if applicable, frame attributes (frame-attributes). The name of this list is defined by the task type you use: polylines, polygons, keypoints, and for bounding boxes, annotations.

Each JSON object contains information about a single annotation and associated label. The following table outlines the parameters you’ll see for each video frame task type.
Task Type | Parameters
---|---
Bounding Box | Box dimensions: **height and width**
| Box top, left corner pixel location: **top** and **left**
Keypoint | Keypoint vertices: `{ "x": int, "y": int }`
Polygon | A list of polygon vertices: **vertices**
| Polygon vertices: `{ "x": int, "y": int }`
| A polygon is a closed shape and so the first point will also represent the last point.
Polyline | A list of polyline vertices: **vertices**
| Polyline vertices: `{ "x": int, "y": int }`

In addition to task type specific values, you will see the following in each JSON object:

- **Values of any label-category-attributes** that were specified for that label.
- The **class-id** of the box. Use the **class-map** in the output manifest file to see which label category this ID maps to.
- An **object-id** which identifies an instance of a label. This ID will be the same across frames if a worker identified the same instance of an object in multiple frames. For example, if a car appeared in multiple frames, all bounding boxes uses to identify that car would have the same **object-id**.
- The **object-name** which is the instance ID of that annotation.

The following is an example of a `SeqLabel.json` file from a bounding box video frame object tracking labeling job. This file will be located under `s3://your-output-bucket/output-prefix/annotations/consolidated-annotation/output/annotation-number/`:

```json
{
  "tracking-annotations": [
    {
      "annotations": [
        {
          "height": 36,
          "width": 46,
          "top": 178,
          "left": 315,
          "class-id": "0",
          "label-category-attributes": {
            "occluded": "no"
          },
          "object-id": "480d450-c0ca-11ea-961f-a9b1c5c97972",
          "object-name": "car:1"
        }
      ],
      "frame-no": "0",
      "frame": "frame_0001.jpeg",
      "frame-attributes": {}
    },
    {
      "annotations": [
        {
          "height": 30,
          "width": 47,
          "top": 163,
```
3D Point Cloud Semantic Segmentation Output

The following is the output manifest file from a 3D point cloud semantic segmentation labeling job.

In addition to the standard elements, the metadata for the label includes a color map that defines which color is used to label the image, the class name associated with the color, and the confidence score for each color. Additionally, there is an `adjustment-status` parameter in the metadata for audit workflows that is set to `adjusted` if the color mask is modified. If you added one or more `frameAttributes` to your label category configuration file, worker responses for frame attributes are in the JSON object, `dataset-object-attributes`.

The `your-label-attribute-ref` parameter contains the location of a compressed file with a `.zlib` extension. When you uncompress this file, it contains an array. Each index in the array corresponds to the index of an annotated point in the input point cloud. The value of the array at a given index gives the class of the point at the same index in the point cloud, based on the semantic color map found in the `color-map` parameter of the `metadata`.

You can use Python code similar to the following to decompress a `.zlib` file:

```python
import zlib
from array import array

# read the label file
compressed_binary_file = open(zlib_file_path/file.zlib, 'rb').read()

# uncompress the label file
binary_content = zlib.decompress(compressed_binary_file)

# load labels to an array
my_int_array_data = array('B', binary_content);

print(my_int_array_data)
```
The code block above will produce an output similar to the following. Each element of the printed array contains the class of a point at the that index in the point cloud. For example, `my_int_array_data[0] = 1` means `point[0]` in the input point cloud has a class 1. In the following output manifest file example, class 0 corresponds with “Background”, 1 with Car, and 2 with Pedestrian.

```
>> array('B', [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2])
```

The following is an example of a semantic segmentation 3D point cloud labeling job output manifest file. The red, italicized text in the examples below depends on labeling job specifications and output data.

```
{
  "source-ref": "s3://AWSDOC-EXAMPLE-BUCKET/examplefolder/frame1.bin",
  "source-ref-metadata":{
    "format": "binary/xyz1",
    "unix-timestamp": 1566861644.759115,
    "ego-vehicle-pose":{}
  },
  "prefix": "s3://AWSDOC-EXAMPLE-BUCKET/lidar_singleframe_dataset/prefix",
  "images": [{},
    "lidar-ss-label-attribute-ref": "s3://your-output-bucket/labeling-job-name/annotations/consolidated-annotation/output/dataset-object-id/filename.zlib",
    "lidar-ss-label-attribute-ref-metadata": {
      'color-map': {
        "0": {"class-name": "Background",
               "hex-color": "#ffffff",
               "confidence": 0.00
             },
        "1": {"class-name": "Car",
               "hex-color": "#2ca02c",
               "confidence": 0.00
             },
        "2": {"class-name": "Pedestrian",
               "hex-color": "#1f77b4",
               "confidence": 0.00
             },
        "3": {"class-name": "Tree",
               "hex-color": "#ff7f0e",
               "confidence": 0.00
             }
      },
      'type': 'groundtruth/point_cloud_single_frame_semantic_segmentation',
      'human-annotated': 'yes',
      'creation-date': '2019-11-12T01:18:14.271944',
      'job-name': 'labeling-job-name',
      //only present for adjustment audit workflow
      "adjustment-status": "adjusted", // "adjusted" means the label was adjusted
      "dataset-object-attributes": {"name": value, name: value}
    }
  }
}
```

### 3D Point Cloud Object Detection Output

The following is sample output from a 3D point cloud objected detection job. For this task type, the data about 3D cuboids is returned in the `3d-bounding-box` parameter, in a list named `annotations`. In this list, each 3D cuboid is described using the following information.
• Each class, or label category, that you specify in your input manifest is associated with a class-id. Use the class-map to identify the class associated with each class ID.

• These classes are used to give each 3D cuboid an object-name in the format `<class>:<integer>` where integer is a unique number to identify that cuboid in the frame.

• center-x, center-y, and center-z are the coordinates of the center of the cuboid, in the same coordinate system as the 3D point cloud input data used in your labeling job.

• length, width, and height describe the dimensions of the cuboid.

• yaw is used to describe the orientation (heading) of the cuboid in radians.

The yaw measurement in the output data is 180 degrees, or pi in radians, minus yaw in the right handed world coordinate system when looking down at the cuboid. In other words, when looking at a cuboid from the top-down, yaw_in_output_data is clockwise-positive (in contrast to the right handed world coordinate system, in which the top-down view is associated with counter-clockwise positive rotation). When looking up from the cuboid, yaw_in_output_data is counterclockwise-positive.

To convert yaw_in_output_data to the more common orientation of the right handed world coordinate system, use the following (all units are in radians):

\[
yaw_{\text{right-handed cartesian system}} = \pi - yaw_{\text{in output data}}
\]

• If you created a 3D point cloud adjustment labeling job and included pitch and roll in the input manifest file, the same pitch and roll measurements will appear in the output manifest file. Otherwise, pitch and role will always be 0.

• If you included label attributes in your input manifest file for a given class, a label-category-attributes parameter is included for all cuboids for which workers selected label attributes.

If one or more cuboids were modified, there is an adjustment-status parameter in the metadata for audit workflows that is set to adjusted. If you added one or more frameAttributes to your label category configuration file, worker responses for frame attributes are in the JSON object, dataset-object-attributes.

The red, italicized text in the examples below depends on labeling job specifications and output data. The ellipses (...) denote a continuation of that list, where additional objects with the same format as the proceeding object can appear.

```json
{
    "source-ref": "s3://AWSDOC-EXAMPLE-BUCKET/examplefolder/frame1.txt",
    "source-ref-metadata": {
        "format": "text/xyzi",
        "unix-timestamp": 1566861644.759115,
        "prefix": "s3://AWSDOC-EXAMPLE-BUCKET/lidar_singleframe_dataset/prefix",
        "ego-vehicle-pose": {
            "heading": {
                "qx": -0.02111296123795955,
                "qy": -0.006495469416730261,
                "qz": -0.008024565904865688,
                "qw": 0.9997181192298087
            },
            "position": {
                "x": -2.7161461413869947,
                "y": 116.25822288149078,
                "z": 1.8348751887989475
            }
        },
        "images": [
            {"fx": 847.7962624528487,
```

399
"fy": 850.0340893791985,
"cx": 576.2129134707038,
"cy": 317.2423573573745,
"k1": 0,
"k2": 0,
"k3": 0,
"k4": 0,
"p1": 0,
"p2": 0,
"skew": 0,
"unix-timestamp": 1566861644.759115,
"image-path": "images/frame_0_camera_0.jpg",
"position": {
  "x": -2.2722515189268138,
  "y": 116.86003310568965,
  "z": 1.454614668542299
},
"heading": {
  "qx": 0.7594754093069037,
  "qy": 0.02181790885672969,
  "qz": -0.02461725233103356,
  "qw": -0.6496916273040025
},
"camera_model": "pinhole"
],
"3d-bounding-box": {
  "annotations": [
    {
      "label-category-attributes": {
        "Occlusion": "Partial",
        "Type": "Sedan"
      },
      "object-name": "Car:1",
      "class-id": 0,
      "center-x": -2.616382013657516,
      "center-y": 125.04149850484193,
      "center-z": 0.311272296465834,
      "length": 2.993000265181146,
      "width": 1.8355260519692056,
      "height": 1.3233490884304047,
      "roll": 0,
      "pitch": 0,
      "yaw": 1.6479308313703527
    },
    {
      "label-category-attributes": {
        "Occlusion": "Partial",
        "Type": "Sedan"
      },
      "object-name": "Car:2",
      "class-id": 0,
      "center-x": -5.188984560617168,
      "center-y": 99.7954483288783,
      "center-z": 0.222643556745657,
      "length": 4,
      "width": 2,
      "height": 2,
      "roll": 0,
      "pitch": 0,
      "yaw": 1.62433170732068055
    }
  ]
}
"3d-bounding-box-metadata":
{
  "objects": [],
  "class_map": {
    "0": "Car",
  },
  "type": "groundtruth/point_cloud_object_detection",
  "human-annotated": "yes",
  "creation-date": "2018-10-18T22:18:13.527256",
  "job-name": "identify-3d-objects",
  "adjustment-status": "adjusted",
  "dataset-object-attributes": {
    name: value, name: value
  }
}

3D Point Cloud Object Tracking Output

The following is an example of an output manifest file from a 3D point cloud object tracking labeling job. The red, italicized text in the examples below depends on labeling job specifications and output data. The ellipses (...) denote a continuation of that list, where additional objects with the same format as the proceeding object can appear.

In addition to the standard elements, the metadata includes a class map that lists each class that has at least one label in the sequence. If one or more cuboids were modified, there is an adjustment-status parameter in the metadata for audit workflows that is set to adjusted.

```
{
  "source-ref": "s3://AWSDOC-EXAMPLE-BUCKET/myfolder/seq1.json",
  "lidar-label-attribute-ref": "s3://<CustomerOutputLocation>/<labelingJobName>/annotations/consolidated-annotation/output/<datasetObjectId>/SeqLabel.json",
  "lidar-label-attribute-ref-metadata": {
    "objects": [
      { "frame-no": 300, "confidence": [] },
      { "frame-no": 301, "confidence": [] },
      ...
    ],
    'class-map': {'0': 'Car', '1': 'Person'},
    'type': 'groundtruth/point_cloud_object_tracking',
    'human-annotated': 'yes',
    'creation-date': '2019-11-12T18:18:14.271944',
    'job-name': 'identify-3d-objects',
    "adjustment-status": "adjusted"
  }
}
```

In the above example, the cuboid data for each frame in seq1.json is in SeqLabel.json in the Amazon S3 location, s3://<customerOutputLocation>/<labelingJobName>/annotations/consolidated-annotation/output/<datasetObjectId>/SeqLabel.json. The following is an example of this label sequence file.

For each frame in the sequence, you see the frame-number, frame-name, if applicable, frame-attributes, and a list of annotations. This list contains 3D cubiods that were drawn for that frame. Each annotation includes the following information:
• An object-name in the format `<class>:<integer>` where class identifies the label category and integer is a unique ID across the dataset.

• When workers draw a cuboid, it is associated with a unique object-id which is associated with all cuboids that identify the same object across multiple frames.

• Each class, or label category, that you specified in your input manifest is associated with a class-id. Use the class-map to identify the class associated with each class ID.

• center-x, center-y, and center-z are the coordinates of the center of the cuboid, in the same coordinate system as the 3D point cloud input data used in your labeling job.

• length, width, and height describe the dimensions of the cuboid.

• yaw is used to describe the orientation (heading) of the cuboid in radians.

The yaw measurement in the output data is 180 degrees, or π in radians, minus yaw in the right handed world coordinate system when looking down at the cuboid. In other words, when looking at a cuboid from the top-down, yaw_in_output_data is clockwise-positive (in contrast to the right handed world coordinate system, in which the top-down view is associated with counter-clockwise positive rotation). When looking up from the cuboid, yaw_in_output_data is counterclockwise-positive.

To convert yaw_in_output_data to the more common orientation of the right handed world coordinate system, use the following (all units are in radians):

\[
yaw_{right\_handed\_cartesian\_system} = \pi - \text{yaw}_{in\_output\_data}
\]

• If you created a 3D point cloud adjustment labeling job and included pitch and roll in the input manifest file, the same pitch and roll measurements will appear in the output manifest file. Otherwise, pitch and role will always be 0.

• If you included label attributes in your input manifest file for a given class, a label-category-attributes parameter is included for all cuboids for which workers selected label attributes.

```json
{
  "tracking-annotations": [
    {
      "frame-number": 0,
      "frame-name": "0.txt.pcd",
      "frame-attributes": {"name": value, name: value},
      "annotations": [
        {
          "label-category-attributes": {},
          "object-name": "Car:4",
          "class-id": 0,
          "center-x": -2.2906369208300674,
          "center-y": 103.73924823843463,
          "center-z": 0.37634114027023313,
          "length": 4,
          "width": 2,
          "height": 2,
          "roll": 0,
          "pitch": 0,
          "yaw": 1.582722214406014,
          "object-id": "ae5dc770-a782-11ea-b57d-67c51a0561a1"
        },
        {
          "label-category-attributes": {
            "Occlusion": "Partial",
            "Type": "Sedan"
          },
          "object-name": "Car:1",
          "class-id": 0,
          "center-x": -2.2906369208300674,
          "center-y": 103.73924823843463,
          "center-z": 0.37634114027023313,
          "length": 4,
          "width": 2,
          "height": 2,
          "roll": 0,
          "pitch": 0,
          "yaw": 1.582722214406014,
          "object-id": "ae5dc770-a782-11ea-b57d-67c51a0561a1"
        }
      ]
    }
  ]
}
```
"center-x": -2.6451293634707413,
"center-y": 124.95344555706848,
"center-z": 0.5020834081743839,
"length": 4,
"width": 2,
"height": 2.080488827301309,
"roll": 0,
"pitch": 0,
"yaw": -1.5963335581398077,
"object-id": "06efb020-a782-11ea-b57d-67c51a0561a1"
},
{
"label-category-attributes": {
"Occlusion": "Partial",
"Type": "Sedan"
},
"object-name": "Car:2",
"class-id": 0,
"center-x": -5.205611313118477,
"center-y": 99.91731932137061,
"center-z": 0.22917217081212138,
"length": 3.8747142207671956,
"width": 1.9999999999999918,
"height": 2,
"roll": 0,
"pitch": 0,
"yaw": 1.567228760316775,
"object-id": "26fad020-a782-11ea-b57d-67c51a0561a1"
}
}
{
"frame-number": 1,
"frame-name": "1.txt.pcd",
"frame-attributes": {},
"annotations": [
{
"label-category-attributes": {},
"object-name": "Car:4",
"class-id": 0,
"center-x": -2.2906369208300674,
"center-y": 103.73924823843463,
"center-z": 0.37634114027023313,
"length": 4,
"width": 2,
"height": 2,
"roll": 0,
"pitch": 0,
"yaw": 1.5827222214406014,
"object-id": "ae5dc770-a782-11ea-b57d-67c51a0561a1"
}
],
"label-category-attributes": {
"Occlusion": "Partial",
"Type": "Sedan"
},
"object-name": "Car:1",
"class-id": 0,
"center-x": -2.6451293634707413,
"center-y": 124.95344555706848,
"center-z": 0.5020834081743839,
"length": 4,
"width": 2,
"height": 2.080488827301309,
"roll": 0,
"pitch": 0,
Enhanced Data Labeling

Amazon SageMaker Ground Truth manages sending your data objects to workers to be labeled. Labeling each data object is a task. Workers complete each task until the entire labeling job is complete. Ground Truth divides the total number of tasks into smaller batches that are sent to workers. A new batch is sent to workers when the previous one is finished.

Ground Truth provides two features that help improve the accuracy of your data labels and reduce the total cost of labeling your data:

- **Annotation consolidation** helps to improve the accuracy of your data object labels. It combines the results of multiple workers' annotation tasks into one high-fidelity label.
- **Automated data labeling** uses machine learning to label portions of your data automatically without having to send them to human workers.

**Topics**

- Batches for Labeling Tasks (p. 404)
- Consolidate Annotations (p. 405)
- Automate Data Labeling (p. 406)
- Chaining Labeling Jobs (p. 412)

**Batches for Labeling Tasks**

Amazon SageMaker Ground Truth sends data objects to your workers in batches. There are one or more tasks for each data object. For each task, a worker annotates one of your data objects. A batch does the following:

- Sets the number of data objects that are available to workers. After the objects are annotated, another batch is sent.
- Breaks the work into smaller chunks to avoid overloading your workforce.
• Provides chunks of data for the iterative training of automated data labeling models.

Ground Truth first sends a batch of 10 tasks to your workers. It uses this small batch to set up the labeling job and to make sure that the job is correctly configured.

Ground Truth then sends larger batches to your workers.

You configure batch size when you create the job using the `CreateLabelingJob` operation. If you use the Amazon SageMaker console to create the labeling job, Ground Truth automatically configures your job to use 1,000 tasks in each batch.

### Consolidate Annotations

An *annotation* is the result of a single worker's labeling task. *Annotation consolidation* combines the annotations of two or more workers into a single label for your data objects. A label, which is assigned to each object in the dataset, is a probabilistic estimate of what the true label should be. Each object in the dataset typically has multiple annotations, but only one label or set of labels.

You decide how many workers annotate each object in your dataset. Using more workers can increase the accuracy of your labels, but also increases the cost of labeling. To learn more about Ground Truth pricing, see [Amazon SageMaker Ground Truth pricing](#).

If you use the Amazon SageMaker console to create a labeling job, the following are the defaults for the number of workers who can annotate objects:

- Text classification—3 workers
- Image classification—3 workers
- Bounding boxes—5 workers
- Semantic segmentation—3 workers
- Named entity recognition—3 workers

When you use the `CreateLabelingJob` operation, you set the number of workers to annotate each data object with the `NumberOfHumanWorkersPerDataObject` parameter. You can override the default number of workers that annotate a data object using the console or the `CreateLabelingJob` operation.

Ground Truth provides an annotation consolidation function for each of its predefined labeling tasks: bounding box, image classification, name entity recognition, semantic segmentation, and text classification. These are the functions:

- Multi-class annotation consolidation for image and text classification uses a variant of the Expectation Maximization approach to annotations. It estimates parameters for each worker and uses Bayesian inference to estimate the true class based on the class annotations from individual workers.
- Bounding box annotation consolidates bounding boxes from multiple workers. This function finds the most similar boxes from different workers based on the Jaccard index, or intersection over union, of the boxes and averages them.
- Semantic segmentation annotation consolidation treats each pixel in a single image as a multi-class classification. This function treats the pixel annotations from workers as “votes,” with more information from surrounding pixels incorporated by applying a smoothing function to the image.
- Named entity recognition clusters text selections by Jaccard similarity and calculates selection boundaries based on the mode, or the median if the mode isn't clear. The label resolves to the most assigned entity label in the cluster, breaking ties by random selection.

You can use other algorithms to consolidate annotations. For information, see [Create Your Own Annotation Consolidation Function](#).
Create Your Own Annotation Consolidation Function

You can choose to use your own annotation consolidation function to determine the final labels for your labeled objects. There are many possible approaches for writing a function and the approach that you take depends on the nature of the annotations to consolidate. Broadly, consolidation functions look at the annotations from workers, measure the similarity between them, and then use some form of probabilistic judgment to determine what the most probable label should be.

If you want to use other algorithms to create annotation consolidations functions, you can find the worker responses in the [project-name]/annotations/worker-response folder of the Amazon S3 bucket where you direct the job output.

Assess Similarity

To assess the similarity between labels, you can use one of the following strategies, or you can use one that meets your data labeling needs:

- For label spaces that consist of discrete, mutually exclusive categories, such as multi-class classification, assessing similarity can be straightforward. Discrete labels either match or do not match.
- For label spaces that don't have discrete values, such as bounding box annotations, find a broad measure of similarity. For bounding boxes, one such measure is the Jaccard index. This measures the ratio of the intersection of two boxes with the union of the boxes to assess how similar they are. For example, if there are three annotations, then there can be a function that determines which annotations represent the same object and should be consolidated.

Assess the Most Probable Label

With one of the strategies detailed in the previous sections in mind, make some sort of probabilistic judgment on what the consolidated label should be. In the case of discrete, mutually exclusive categories, this can be straightforward. One of the most common ways to do this is to take the results of a majority vote between the annotations. This weights the annotations equally.

Some approaches attempt to estimate the accuracy of different annotators and weight their annotations in proportion to the probability of correctness. An example of this is the Expectation Maximization method, which is used in the default Ground Truth consolidation function for multi-class annotations.

For more information about creating an annotation consolidation function, see Step 3: Processing with AWS Lambda (p. 308).

Automate Data Labeling

If you choose, Amazon SageMaker Ground Truth can use active learning to automate the labeling of your input data for certain built-in task types. Active learning is a machine learning technique that identifies data that should be labeled by your workers. In Ground Truth, this functionality is called automated data labeling. Automated data labeling helps to reduce the cost and time that it takes to label your dataset compared to using only humans. When you use automated labeling, you incur SageMaker training and inference costs.

We recommend using automated data labeling on large datasets because the neural networks used with active learning require a significant amount of data for every new dataset. Typically, as you provide more data, the potential for high accuracy predictions goes up. Data will only be auto-labeled if the neural network used in the auto-labeling model can achieve an acceptably high level of accuracy. Therefore, with larger datasets, there is more potential to automatically label the data because the neural network can achieve high enough accuracy for auto-labeling. Automated data labeling is most appropriate when you have thousands of data objects. The minimum number of objects allowed for automated data labeling is 1,250, but we strongly suggest providing a minimum of 5,000 objects.

Automated data labeling is available only for the following Ground Truth built-in task types:
To learn how to create a custom active learning workflow using your own model, see Set up an active learning workflow with your own model (p. 412).

Note
Before you use an automated-labeling model in production, you need to fine-tune or test it, or both. You might fine-tune the model (or create and tune another supervised model of your choice) on the dataset produced by your labeling job to optimize the model's architecture and hyperparameters. If you decide to use the model for inference without fine-tuning it, we strongly recommend making sure that you evaluate its accuracy on a representative (for example, randomly selected) subset of the dataset labeled with Ground Truth and that it matches your expectations.

How it Works

You enable automated data labeling when you create a labeling job. This is how it works:

1. When Ground Truth starts an automated data labeling job, it selects a random sample of input data (objects) and sends it to human workers.

2. When the labeled data is returned, it is used to create a training set and a validation set. Ground Truth uses these datasets to train and validate the model used for auto-labeling.

3. Ground Truth runs a batch transform job, using the validated model for inference on the validation data. Batch inference produces a confidence score and quality metric for each object in the validation data.

4. The auto labeling component will use these quality metrics and confidence scores to create a confidence score threshold that ensures quality labels.

5. Ground Truth runs a batch transform job on the unlabeled data in the dataset, using the same validated model for inference. This produces a confidence score for each object.

6. The Ground Truth auto labeling component determines if the confidence score produced in step 5 for each object meets the required threshold determined in step 4. If the confidence score meets the threshold, the expected quality of automatically labeling exceeds the requested level of accuracy and that object is considered auto-labeled.

7. Step 6 produces a dataset of unlabeled data with confidence scores. Ground Truth selects data points with low confidence scores from this dataset and sends them to human workers.

8. Ground Truth uses the existing human-labeled data and this additional labeled data from human workers to update the model.

9. The process is repeated until the dataset is fully labeled or until another stopping condition is met. For example, auto-labeling stops if your human annotation budget is reached.

The preceding steps happen in iterations. Select each tab in the following table to see an example of the processes that happen in each iteration for an object detection automated labeling job. The number of data objects used in a given step in these images (for example, 200) is specific to this example. If there are fewer than 5,000 objects to label, the validation set size is 20% of the whole dataset. If there are more than 5,000 objects in your input dataset, the validation set size is 10% of the whole dataset. You can control the number of human labels collected per active learning iteration by changing the value for MaxConcurrentTaskCount when using the API operation CreateLabelingJob. This value is set to
1,000 when you create a labeling job using the console. In the active learning flow illustrated under the `Active Learning` tab, this value is set to 200.

**Model Training**

![Diagram showing active learning process]

- **Iterations 1, 2, and 3:** Model training and validation.
- **Unlabeled images:** Sent to annotators.
- **Send randomly selected images to annotators:** Iteration 3: Validation.
Automated Labeling
Active Learning

Accuracy of Automated Labels

The definition of accuracy depends on the built-in task type that you use with automated labeling. For all task types, these accuracy requirements are pre-determined by Ground Truth and cannot be manually configured.

- For image classification and text classification, Ground Truth uses logic to find a label-prediction confidence level that corresponds to at least 95% label accuracy. This means Ground Truth expects the accuracy of the automated labels to be at least 95% when compared to the labels that human labelers would provide for those examples.
- For bounding boxes, the expected mean Intersection Over Union (IoU) of the auto-labeled images is 0.6. To find the mean IoU, Ground Truth calculates the mean IoU of all the predicted and missed boxes on the image for every class, and then averages these values across classes.
- For semantic segmentation, the expected mean IoU of the auto-labeled images is 0.7. To find the mean IoU, Ground Truth takes the mean of the IoU values of all the classes in the image (excluding the background).

At every iteration of Active Learning (steps 3-6 in the list above), the confidence threshold is found using the human-annotated validation set so that the expected accuracy of the auto-labeled objects satisfies certain predefined accuracy requirements.

Create an Automated Data Labeling Job (Console)

To create a labeling job that uses automated labeling in the SageMaker console, use the following procedure.

To create an automated data labeling job (console)

2. Using Create a Labeling Job (Console) (p. 314) as a guide, complete the Job overview and Task type sections. Note that auto labeling is not supported for custom task types.
4. In the same section, choose **Enable automated data labeling**.
5. Using Step 4: Configure the Bounding Box Tool (p. 165) as a guide, create worker instructions in the section **Task Type labeling tool**. For example, if you chose **Semantic segmentation** as your labeling job type, this section is called **Semantic segmentation labeling tool**.
6. To preview your worker instructions and dashboard, choose **Preview**.
7. Choose **Create**. This creates and starts your labeling job and the auto labeling process.

You can see your labeling job appear in the **Labeling jobs** section of the SageMaker console. Your output data appears in the Amazon S3 bucket that you specified when creating the labeling job. For more information about the format and file structure of your labeling job output data, see **Output Data** (p. 381).

**Create an Automated Data Labeling Job (API)**

To create an automated data labeling job using the the SageMaker API, use the **LabelingJobAlgorithmsConfig** parameter of the **CreateLabelingJob** operation. To learn how to start a labeling job using the **CreateLabelingJob** operation, see **Create a Labeling Job (API)** (p. 317).

Specify the Amazon Resource Name (ARN) of the algorithm that you are using for automated data labeling in the **LabelingJobAlgorithmSpecificationArn** parameter. Choose from one of the four Ground Truth built-in algorithms that are supported with automated labeling:

- **Image Classification (Single Label)** (p. 178)
- **Image Semantic Segmentation** (p. 171)
- **Object detection (Bounding Box)** (p. 166)
- **Text Classification (Single Label)** (p. 188)

When an automated data labeling job finishes, Ground Truth returns the ARN of the model it used for the automated data labeling job. Use this model as the starting model for similar auto-labeling job types by providing the ARN, in string format, in the **InitialActiveLearningModelArn** parameter. To retrieve the model's ARN, use an AWS Command Line Interface (AWS CLI) command similar to the following.

```shell
# Fetch the mARN of the model trained in the final iteration of the previous labeling job.Ground Truth
pretrained_model_arn = sagemaker_client.describe_labeling_job(LabelingJobName=job_name)['LabelingJobOutput']['FinalActiveLearningModelArn']
```

To encrypt data on the storage volume attached to the ML compute instance(s) that are used in automated labeling, include an AWS Key Management Service (AWS KMS) key in the **VolumeKmsKeyId** parameter. For information about AWS KMS keys, see **What is AWS Key Management Service?** in the **AWS Key Management Service Developer Guide**.

For an example that uses the **CreateLabelingJob** operation to create an automated data labeling job, see the **object_detection_tutorial** example in the **SageMaker Examples, Ground Truth Labeling Jobs** section of a SageMaker notebook instance. To learn how to create and open a notebook instance, see **Create a Notebook Instance** (p. 124). To learn how to access SageMaker example notebooks, see **Example Notebooks** (p. 134).

**Amazon EC2 Instances Required for Automated Data Labeling**

The following table lists the Amazon Elastic Compute Cloud (Amazon EC2) instances that you need to run automated data labeling for training and batch inference jobs.
Automated Data Labeling Job Type | Training Instance Type | Inference Instance Type
--- | --- | ---
Image classification | ml.p3.2xlarge* | ml.c5.xlarge
Object detection (bounding box) | ml.p3.2xlarge* | ml.c5.4.large
Text classification | ml.c5.2xlarge | ml.m4.xlarge
Semantic segmentation | ml.p3.2xlarge* | ml.p3.2xlarge*

* In the Asia Pacific (Mumbai) Region (ap-south-1) use ml.p2.8xlarge instead.

Automated data labeling incurs two separate charges: the per-item charge (see pricing), and the charge for the Amazon EC2 instance required to run the model (see Amazon EC2 pricing).

Ground Truth manages the instances that you use for automated data labeling jobs. It creates, configures, and terminates the instances as needed to perform your job. These instances don’t appear in your Amazon EC2 instance dashboard.

**Set up an active learning workflow with your own model**

You can create an active learning workflow with your own algorithm to run training and inferences in that workflow to auto-label your data. The notebook `bring_your_own_model_for_sagemaker_labeling_workflows_with_active_learning.ipynb` demonstrates this using the SageMaker built-in algorithm, BlazingText. This notebook provides an AWS CloudFormation stack that you can use to execute this workflow using AWS Step Functions. You can find the notebook and supporting files in this GitHub repository.

You can also find this notebook in the SageMaker Examples repo. See Use Example Notebooks to learn how to find an Amazon SageMaker example notebook.

**Chaining Labeling Jobs**

Amazon SageMaker Ground Truth can reuse datasets from prior jobs in two ways: cloning and chaining.

*Cloning* copies the setup of a prior labeling job and allows you to make additional changes before setting it to run.

*Chaining* uses not only the setup of the prior job, but also the results. This allows you to continue an incomplete job and add labels or data objects to a completed job. Chaining is a more complex operation.

For data processing:

- Cloning uses the prior job’s *input* manifest, with optional modifications, as the new job’s *input* manifest.
- Chaining uses the prior job’s *output* manifest as the new job’s *input* manifest.

Chaining is useful when you need to:

- Continue a labeling job that was manually stopped.
- Continue a labeling job that failed mid-job, after fixing issues.
- Switch to automated data labeling after manually labeling part of a job (or the other way around).
- Add more data objects to a completed job and start the job from there.
• Add another annotation to a completed job. For example, you have a collection of phrases labeled for topic, then want to run the set again, categorizing them by the topic’s implied audience.

In Amazon SageMaker Ground Truth you can configure a chained labeling job with either the console or the API.

**Key Term: Label Attribute Name**

The *label attribute name* (LabelAttributeName in the API) is a string used as the key for the key-value pair formed with the label that a worker assigns to the data object.

The following rules apply for the label attribute name:

- It can’t end with `-metadata`.
- The names `source` and `source-ref` are reserved and can’t be used.
- For semantic segmentation labeling jobs, it must end with `-ref`. For all other labeling jobs, it can’t end with `-ref`. If you use the console to create the job, Amazon SageMaker Ground Truth automatically appends `-ref` to all label attribute names except for semantic segmentation jobs.
- For a chained labeling job, if you’re using the same label attribute name from the originating job and you configure the chained job to use auto-labeling, then if it had been in auto-labeling mode at any point, Ground Truth uses the model from the originating job.

In an output manifest, the label attribute name appears similar to the following.

```json
"source-ref": "<S3 URI>",
"<label attribute name>": {
  "annotations": [{
    "class_id": 0,
    "width": 99,
    "top": 87,
    "height": 62,
    "left": 175
  }],
  "image_size": [{
    "width": 344,
    "depth": 3,
    "height": 234
  }]
},
"<label attribute name>-metadata": {
  "job-name": "<job name>",
  "class-map": {
    "0": "<label attribute name>",
  },
  "human-annotated": "yes",
  "objects": [{
    "confidence": 0.09
  }],
  "creation-date": "<timestamp>",
  "type": "groundtruth/object-detection"
}
```

If you’re creating a job in the console and don’t explicitly set the label attribute name value, Ground Truth uses the job name as the label attribute name for the job.

**Start a Chained Job (Console)**

Choose a stopped, failed, or completed labeling job from the list of your existing jobs. This enables the *Actions* menu.
From the **Actions** menu, choose **Chain**.

**Job Overview Panel**

In the **Job overview** panel, a new **Job name** is set based on the title of the job from which you are chaining this one. You can change it.

You may also specify a label attribute name different from the labeling job name.

If you're chaining from a completed job, the label attribute name uses the name of the new job you're configuring. To change the name, select the check box.

If you're chaining from a stopped or failed job, the label attribute name uses the name of the job from which you're chaining. It's easy to see and edit the value because the name check box is checked.

**Attribute label naming considerations**

- **The default** uses the label attribute name Ground Truth has selected. All data objects without data connected to that label attribute name are labeled.
- **Using a label attribute name** not present in the manifest causes the job to process *all* the objects in the dataset.

The **input dataset location** in this case is automatically selected as the output manifest of the chained job. The input field is not available, so you cannot change it.

**Adding data objects to a labeling job**

You cannot specify an alternate manifest file. Manually edit the output manifest from the previous job to add new items before starting a chained job. The Amazon S3 URI helps you locate where you are storing the manifest in your Amazon S3 bucket. Download the manifest file from there, edit it locally on your computer, and then upload the new version to replace it. Make sure you are not introducing errors during editing. We recommend you use JSON linter to check your JSON. Many popular text editors and IDEs have linter plugins available.

**Start a Chained Job (API)**

The procedure is almost the same as setting up a new labeling job with **CreateLabelingJob**, except for two primary differences:

- **Manifest location**: Rather than use your original manifest from the prior job, the value for the `ManifestS3Uri` in the `DataSource` should point to the Amazon S3 URI of the *output manifest* from the prior labeling job.
- **Label attribute name**: Setting the correct `LabelAttributeName` value is important here. This is the key portion of a key-value pair where labeling data is the value. Sample use cases include:
  - **Adding new or more specific labels to a completed job** — Set a new label attribute name.
  - **Labeling the unlabeled items from a prior job** — Use the label attribute name from the prior job.

**Use a Partially Labeled Dataset**

You can get some chaining benefits if you use an augmented manifest that has already been partially labeled. Check the **Label attribute name** check box and set the name so that it matches the name in your manifest.

If you're using the API, the instructions are the same as those for starting a chained job. However, be sure to upload your manifest to an Amazon S3 bucket and use it instead of using the output manifest from a prior job.

The **Label attribute name** value in the manifest has to conform to the naming considerations discussed earlier.
Ground Truth Security and Permissions

Use the topics on this page to learn about Ground Truth security features, and how to configure AWS Identity and Access Management (IAM) permissions to allow an IAM user or role create a labeling job. Additionally, learn how to create an execution role. An execution role is the role that you specify when you create a labeling job and it is used to execute your labeling job.

If you are a new user and want to get started quickly, or if you do not require granular permissions, see Grant General Permissions To Get Started Using Ground Truth (p. 417).

**Important**
When you create your labeling job, if you set the Task time limit (TaskTimeLimitInSeconds when using the Amazon SageMaker API) to be greater than one hour (3,600 seconds), you must increases the max session duration of your execution role to be greater than or equal to the task timeout.

You can modify the max session duration of your execution role using the IAM console, AWS CLI, and IAM API. To modify your execution role, go to Modifying a Role in the IAM User Guide, select your preferred method (console, CLI, or API) to modify the role from the Topics list, and then select Modifying a Role Maximum Session Duration to view the instructions. For 3D point cloud task types, refer to Increase MaxSessionDuration for Execution Role (p. 257).

For more information about IAM users and roles, see Identities (Users, Groups, and Roles) in the IAM User Guide.

To learn more about using IAM with SageMaker, see Identity and Access Management for Amazon SageMaker (p. 1618).

**Topics**
- CORS Permission Requirement (p. 415)
- Assign IAM Permissions to Use Ground Truth (p. 416)
- Data and Storage Volume Encryption (p. 423)
- Workforce Authentication and Restrictions (p. 423)

**CORS Permission Requirement**

Earlier in 2020, widely used browsers like Chrome and Firefox changed their default behavior for rotating images based on image metadata, referred to as EXIF data. Previously, images would always display in browsers exactly how they are stored on disk, which is typically unrotated. After the change, images now rotate according to a piece of image metadata called orientation value. This has important implications for the entire machine learning (ML) community. For example, if the EXIF orientation is not considered, applications that are used to annotate images may display images in unexpected orientations and result in incorrect labels.

It is estimated that, starting with Chrome 88 on January 19th, 2021, AWS can no longer automatically prevent the rotation of images because the web standards group W3C has decided that the ability to control rotation of images violates the web’s Same Origin Policy. Therefore, to ensure human workers annotate your input images in a predictable orientation when you submit requests to create a labeling job, you must add a CORS header policy to the S3 buckets that contain your input images by January 12th, 2021.

**Important**
If you do not add a CORS configuration to the S3 buckets that contains your input data by January 12th, 2021, labeling tasks for those input data objects will fail.

If you create your job through the Ground Truth console, under Enable enhanced image access, a check box is select to enable CORS configuration on the S3 bucket that contains your input manifest file.
Keep this check box selected. If all of your input data is not located in the same S3 bucket as your input manifest file, you must add a CORS configuration to all S3 buckets that contain input data using the following instructions.

Enable enhanced image access

<table>
<thead>
<tr>
<th>Enable Cross-Origin Resource Sharing (CORS) configuration</th>
<th>Info</th>
</tr>
</thead>
<tbody>
<tr>
<td>CORS is a browser security policy for sharing resources. Without proper CORS settings in your S3 bucket, a worker might not be able to complete your image annotation tasks.</td>
<td></td>
</tr>
</tbody>
</table>

If you are using the CreateLabelingJob API to create a Ground Truth labeling job, you can add a CORS policy to an S3 bucket that contains input data in the S3 console. To set the required CORS headers on the S3 bucket that contain your input images in the S3 console, follow the directions detailed in How do I add cross-domain resource sharing with CORS?. Use the following CORS configuration code for the buckets that host your images (Choose JSON or XML based on your preference; they both accomplish the same configuration).

**JSON**

```json
[{  "AllowedHeaders": [],  "AllowedMethods": ["GET"],  "AllowedOrigins": ["*"],  "ExposeHeaders": []}
```

**XML**

```xml
<CORSConfiguration>  
    <CORSRule>  
        <AllowedOrigin>*</AllowedOrigin>  
        <AllowedMethod>GET</AllowedMethod>  
    </CORSRule>  
</CORSConfiguration>
```

**Assign IAM Permissions to Use Ground Truth**

Use the topics on this page to learn how to use AWS Identity and Access Management (IAM) managed and custom policies to manage access to Ground Truth and associated resources.

You can use the sections on this page to learn the following:

- How to create IAM policies that will grant an IAM user or role permission to create a labeling job. Administrators can use IAM policies to restrict access to Amazon SageMaker and other AWS services that are specific to Ground Truth.
- How to create an execution role for labeling jobs. An execution role is the role that you specify when you create a labeling job and it is used to execute your labeling job.

The following is an overview of the topics you’ll find on this page:

- If you are getting started using Ground Truth, or you do not require granular permissions for your use case, see Grant General Permissions To Get Started Using Ground Truth (p. 417).
- Use the policy in Permissions Required to Use the Amazon SageMaker Ground Truth Console (p. 417) to grant access to the Ground Truth area of the SageMaker console. This policy includes permissions to create and modify private work teams. To learn more about these permissions, see Grant Permissions for Private Workforce Creation (p. 419).
• When you create a labeling job, you must provide an execution role. Use Create an Execution Role to Start a Labeling Job (p. 420) to learn about the permissions required for this role.

Grant General Permissions To Get Started Using Ground Truth

If you are getting started using Ground Truth and you do not require granular permissions for your use case, you can attach the managed policy AmazonSageMakerFullAccess to an IAM user or role to give that entity permission to create a labeling job. This is a broad policy that grants an IAM entity permission to use SageMaker features, as well as features of related AWS services through the console and API. This policy will give the IAM entity permission to create a labeling job and to create and manage workforces using Amazon Cognito. To learn more, see AmazonSageMakerFullAccess Policy.

To create an execution role, you can attach the policy AmazonSageMakerGroundTruthExecution to an IAM role. An execution role is the role that you specify when you create a labeling job and it is used to execute your labeling job. AmazonSageMakerGroundTruthExecution grants all required permissions except for data and storage volume encryption if your S3 buckets, objects, and Lambda functions do not meet the conditions specified in the policy. Additionally, be aware of the following Amazon S3 and AWS Lambda permissions included in this policy:

- **AWS Lambda permissions** – When you create a custom labeling workflow, this execution role is restricted to invoking AWS Lambda functions with one of the following strings as part of the function name: GtRecipe, SageMaker, Sagemaker, sagemaker, or LabelingFunction. This applies to both your pre-annotation and post-annotation Lambda functions. If you choose to use names without those strings, you must explicitly provide lambda:InvokeFunction permission to the execution role used to create the labeling job.

- **Amazon S3 permissions** – This policy grants an execution role permission to access Amazon S3 buckets with the following strings in the name: GroundTruth, Groundtruth, groundtruth, SageMaker, Sagemaker, and sagemaker. Make sure your input and output bucket names include these strings, or add additional permissions to your execution role to grant it permission to access your S3 buckets.

**Important**

When you create your labeling job, if you set the Task time limit (TaskTimeLimitInSeconds when using the API) to be greater than one hour (3,600 seconds), you must increases the max session duration of your execution role to be greater than or equal to the task timeout. You can modify the max session duration of your execution role using the IAM console, AWS CLI, and IAM API. To modify your execution role, go to Modifying a Role in the IAM User Guide, select your preferred method (console, CLI, or API) to modify the role from the Topics list, and then select Modifying a Role Maximum Session Duration to view the instructions. For 3D point cloud task types, refer to Increase MaxSessionDuration for Execution Role (p. 257).

Permissions Required to Use the Amazon SageMaker Ground Truth Console

To access the SageMaker console, you must have a minimum set of permissions. To use the Ground Truth console, you need to grant permissions for additional resources. Specifically, the console needs permissions for the AWS Marketplace to view subscriptions, Amazon S3 actions for access to your input and output files. Amazon Cognito permission is required for initial work team setup.

To grant permission to an IAM user or role to use the Ground Truth area of the SageMaker console to create a labeling job, attach the following policy to the user or role.

```json
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "SageMakerApis",
```
"Effect": "Allow",
"Action": [ 
  "sagemaker:*"
],
"Resource": "*"
},
{
"Sid": "KmsKeysForCreateForms",
"Effect": "Allow",
"Action": [ 
  "kms:DescribeKey",
  "kms:ListAliases"
],
"Resource": "*"
},
{
"Sid": "AccessAwsMarketplaceSubscriptions",
"Effect": "Allow",
"Action": [ 
  "aws-marketplace:ViewSubscriptions",
  "aws-marketplace:DescribeListings"
],
"Resource": "*"
},
{
"Effect": "Allow",
"Action": [ 
  "secretsmanager:CreateSecret",
  "secretsmanager:DescribeSecret",
  "secretsmanager:ListSecrets"
],
"Resource": "*"
},
{
"Sid": "ListAndCreateExecutionRoles",
"Effect": "Allow",
"Action": [ 
  "iam:ListRoles",
  "iam:CreateRole",
  "iam:CreatePolicy",
  "iam:AttachRolePolicy"
],
"Resource": "*"
},
{
"Sid": "PassRoleForExecutionRoles",
"Effect": "Allow",
"Action": [ 
  "iam:PassRole"
],
"Resource": "*",
"Condition": { 
  "StringEquals": { 
    "iam:PassedToService": "sagemaker.amazonaws.com"
  }
}
},
{
"Sid": "GroundTruthConsole",
"Effect": "Allow",
"Action": [ 
  "groundtruthlabeling:*",
  "lambda:InvokeFunction",
  "lambda:ListFunctions",
  "s3:GetObject",
  "s3:PutObject",
  "s3:ListBucket",
  "s3:DeleteObject"
],
"Resource": "*"
}
"s3:ListBucket",
"s3:GetBucketCors",
"s3:PutBucketCors",
"s3:SelectObjectContent",
"cognito-idp:AdminAddUserToGroup",
"cognito-idp:AdminCreateUser",
"cognito-idp:AdminDeleteUser",
"cognito-idp:AdminDisableUser",
"cognito-idp:AdminEnableUser",
"cognito-idp:AdminRemoveUserFromGroup",
"cognito-idp:CreateGroup",
"cognito-idp:CreateUserPool",
"cognito-idp:CreateUserPoolClient",
"cognito-idp:CreateUserPoolDomain",
"cognito-idp:DescribeUserPool",
"cognito-idp:DescribeUserPoolClient",
"cognito-idp:ListGroups",
"cognito-idp:ListIdentityProviders",
"cognito-idp:ListUsers",
"cognito-idp:ListUsersInGroup",
"cognito-idp:ListUserPoolClients",
"cognito-idp:ListUserPools",
"cognito-idp:UpdateUserPool",
"cognito-idp:UpdateUserPoolClient"
],
"Resource": "*
} }

To learn more about the permissions required to use the SageMaker console, see Using the SageMaker Console (p. 1626).

Grant Permissions for Private Workforce Creation

When added to a permissions policy, the following permission grants access to create and manage a private workforce and work team. To learn more about private workforces, see Use a Private Workforce (p. 430).

{"Effect": "Allow",
"Action": [
"cognito-idp:AdminAddUserToGroup",
"cognito-idp:AdminCreateUser",
"cognito-idp:AdminDeleteUser",
"cognito-idp:AdminDisableUser",
"cognito-idp:AdminEnableUser",
"cognito-idp:AdminRemoveUserFromGroup",
"cognito-idp:CreateGroup",
"cognito-idp:CreateUserPool",
"cognito-idp:CreateUserPoolClient",
"cognito-idp:CreateUserPoolDomain",
"cognito-idp:DescribeUserPool",
"cognito-idp:DescribeUserPoolClient",
"cognito-idp:ListGroups",
"cognito-idp:ListIdentityProviders",
"cognito-idp:ListUsers",
"cognito-idp:ListUsersInGroup",
"cognito-idp:ListUserPoolClients",
"cognito-idp:ListUserPools",
"cognito-idp:UpdateUserPool",
"cognito-idp:UpdateUserPoolClient"
],
"Resource": "*
} 
Create an Execution Role to Start a Labeling Job

When you configure your labeling job, you need to provide an execution role which is a role that SageMaker has permission to assume to start and run your labeling job.

This role must give SageMaker permission to access the following:

- Amazon S3 to retrieve your input data and write output data to an Amazon S3 bucket. You can either grant permission for an IAM role to access an entire bucket by providing the bucket ARN, or you can grant access to the role to access specific resources in a bucket. For example, the ARN for a bucket may look similar to `arn:aws:s3:::awsexamplebucket1` and the ARN of a resource in an Amazon S3 bucket may look similar to `arn:aws:s3:::awsexamplebucket1/resource-key`. For more information, see Amazon S3 Resources in the Amazon Simple Storage Service Developer Guide.
- CloudWatch to log worker metrics and labeling job statuses.
- (Optional) AWS KMS for data encryption.
- When you create a custom labeling workflow, AWS Lambda for processing input and output data.

All of the permissions above can be granted with the AmazonSageMakerGroundTruthExecution managed policy except for data and storage volume encryption if your S3 buckets, objects, and Lambda functions do not meet the conditions specified in the policy.

When you create your labeling job, if you set the Task time limit (TaskTimeLimitInSeconds when using the API) to be greater than one hour (3,600 seconds), you must increases the max session duration of your execution role to be greater than or equal to the task timeout.

You can modify the max session duration of your execution role using the IAM console, AWS CLI, and IAM API. To modify your execution role, go to Modifying a Role in the IAM User Guide, select your preferred method (console, CLI, or API) to modify the role from the Topics list, and then select Modifying a Role Maximum Session Duration to view the instructions. For 3D point cloud task types, refer to Increase MaxSessionDuration for Execution Role (p. 257).

Use the following policy examples to create an execution role that fits your specific use case.

The following policy grants permission to create a labeling job for a built-in task type. This execution policy does not include permissions for AWS KMS data encryption or decryption.

```json
{
"Version": "2012-10-17",
"Statement": [
{
"Effect": "Allow",
"Principal": {"Service": "sagemaker.amazonaws.com"},
"Action": "sts:AssumeRole"
},
{
"Effect": "Allow",
"Action": [
"iam:PassRole"
],
"Resource": "arn:{customer role arn}",
"Condition": {
"StringEquals": {
"iam:PassedToService": [
"sagemaker.amazonaws.com"
]
}
}
},
{...
```
You can use the following policy to create an execution role that works with an automated labeling job. Replace `arn:aws:iam::<account-number>:role/<role-name>` with your role ARN. You can find your IAM role ARN in the IAM console under Roles.

```json
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "s3:GetObject",
        "s3:PutObject",
        "s3:GetBucketLocation",
        "s3:ListBucket"
      ],
      "Resource": "*"
    },
    {
      "Effect": "Allow",
      "Action": [
        "iam:PassRole"
      ],
      "Resource": "arn:aws:iam::<account-number>:role/<role-name>",
      "Condition": {
        "StringEquals": {
          "iam:PassedToService": [
            "sagemaker.amazonaws.com",
            "sagemaker.amazonaws.com"
          ]
        }
      }
    },
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "sagemaker.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```
To create a custom labeling workflow, you need to add permission for AWS Lambda to process input and output data. Modify the following policy by adding the Lambda function ARNs for your pre- and post-annotation lambda functions in a list under `Resource`. Attach the policy to an execution role that you use to create custom labeling workflow.

```json
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "lambda:InvokeFunction"
            ],
            "Resource": [
            ]
        }
    ]
}
```

To add input data decryption or output data encryption using AWS KMS to any type of labeling job, modify the following policy by listing KMS key ARNs you want to grant permissions to use under `Resource`. Attach the policy to an execution role.

**Important**
You cannot use Ground Truth automated data setup if you use AWS KMS to encrypt buckets or objects in Amazon S3.

```json
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "kms:DescribeKey",
                "kms:Encrypt",
                "kms:Decrypt",
                "kms:ListAliases"
            ],
            "Resource": [
                "arn:aws:kms:<region>:<account-id>:key/<key-id>",
                "arn:aws:kms:<region>:<account-id>:key/<key-id>
            ]
        }
    ]
}
```

To encrypt the storage volume attached to the ML compute instances that run the training job for automated labeling, include the following in one of the execution policies above in the `Statement` section. Storage volume encryption is only available when you create a labeling job using the API operation `CreateLabelingJob`. For more information, see `LabelingJobResourceConfig`.

```json
{
    "Version": "2012-10-17",
    "Statement": [
```
Data and Storage Volume Encryption

Amazon SageMaker Ground Truth enables you to label highly sensitive data by allowing you to stay in control of your data and employing security best practices. Use the topics on this page to learn about Ground Truth security features.

By default, Ground Truth encrypts data in an internal database used by the service and in Amazon S3 buckets with Amazon-owned customer master key (CMK).

Optionally, you can provide an AWS Key Management Service (AWS KMS) key ID when you create a labeling job, which Ground Truth uses to encrypt your output data. If you use a KMS key ID or an alias of your master key, your Amazon SageMaker execution role must include permissions to call `kms:Encrypt`. To learn how to add this permission to an execution role, see Create an Execution Role to Start a Labeling Job (p. 420).

If you don't provide a KMS key ID, Amazon SageMaker uses the default AWS KMS key for Amazon S3 for your role's account. SageMaker uses server-side encryption with KMS-managed keys for `LabelingJobOutputConfig`. For more information, see `LabelingJobOutputConfig`.

When you create a labeling job with automated labeling using the `CreateLabelingJob` API operation, you have the option to encrypt the storage volume attached to the ML compute instances that run the training job. To add encryption to your storage volume, use the parameter `VolumeKmsKeyId` to input a AWS KMS key. For more information about this parameter, see `LabelingJobResourceConfig`. If you use a KMS key ID or key ARN, your SageMaker execution role must include permissions to call `kms:CreateGrant`. To learn how to add this permission to an execution role, see Create an Execution Role to Start a Labeling Job (p. 420).

Workforce Authentication and Restrictions

Ground Truth enables you to use your own private workforce to work on labeling jobs. A private workforce is an abstract concept and it refers to a set of people who work for you. Each labeling job is created using a work team, composed of workers in your workforce. Ground Truth supports private workforce creation using Amazon Cognito.

A Ground Truth workforce maps to a Cognito user pool. A Ground Truth work team maps to a Cognito user group. Cognito manages the worker authentication. Cognito supports Open ID connection (OIDC) and customers can set up Cognito federation with their own identity provider (IdP).

Ground Truth only allows one workforce per account per AWS Region. Each workforce has a dedicated Ground Truth work portal login URL.

You can also restrict workers to a Classless Inter-Domain Routing (CIDR) block/IP address range. This means annotators must be on a specific network to access the annotation site. You can add up to ten CIDR blocks for one workforce. To learn more, see Manage Private Workforce Using the Amazon SageMaker API (p. 445).

To learn how you can create a private workforce, see Create a Private Workforce (Amazon Cognito) (p. 430).
Restrict Access to Workforce Types

Amazon SageMaker Ground Truth work teams fall into one of three workforce types: public (with Amazon Mechanical Turk), private, and vendor. To restrict IAM user access to a specific work team using one of these types or the work team ARN, use the `sagemaker:WorkteamType` and/or the `sagemaker:WorkteamArn` condition keys. For the `sagemaker:WorkteamType` condition key, use string condition operators. For the `sagemaker:WorkteamArn` condition key, use Amazon Resource Name (ARN) condition operators. If the user attempts to create a labeling job with a restricted work team, SageMaker returns an access denied error.

The policies below demonstrate different ways to use the `sagemaker:WorkteamType` and `sagemaker:WorkteamArn` condition keys with appropriate condition operators and valid condition values.

The following example uses the `sagemaker:WorkteamType` condition key with the `StringEquals` condition operator to restrict access to a public work team. It accepts condition values in the following format: `workforcetype-crowd`, where `workforcetype` can equal `public`, `private`, or `vendor`.

```json
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Sid": "RestrictWorkteamType",
         "Effect": "Deny",
         "Action": "sagemaker:CreateLabelingJob",
         "Resource": "*",
         "Condition": {
            "StringEquals": {
               "sagemaker:WorkteamType": "public-crowd"
            }
         }
      }
   ]
}
```

The following policies show how to restrict access to a public work team using the `sagemaker:WorkteamArn` condition key. The first shows how to use it with a valid IAM regex-variant of the work team ARN and the `ArnLike` condition operator. The second shows how to use it with the `ArnEquals` condition operator and the work team ARN.

```json
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Sid": "RestrictWorkteamType",
         "Effect": "Deny",
         "Action": "sagemaker:CreateLabelingJob",
         "Resource": "*",
         "Condition": {
            "ArnLike": {
               "sagemaker:WorkteamArn": "arn:aws:sagemaker:*:*:workteam/public-crowd/*"
            }
         }
      }
   ]
}
```

```json
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Sid": "RestrictWorkteamType",
         "Effect": "Deny",
         "Action": "sagemaker:CreateLabelingJob",
         "Resource": "*",
         "Condition": {
            "ArnEquals": {
               "sagemaker:WorkteamArn": "arn:aws:sagemaker:*:*:workteam/public-crowd/"
            }
         }
      }
   ]
}
```
Monitor Labeling Job Status

To monitor the status of your labeling jobs, you can set up an Amazon CloudWatch Events (CloudWatch Events) rule for Amazon SageMaker Ground Truth (Ground Truth) to send an event to CloudWatch Events when a labeling job status changes to Completed, Failed, or Stopped.

Once you create a rule, you can add a target to it. CloudWatch Events uses this target to invoke another AWS service to process the event. For example, you can create a target using a Amazon Simple Notification Service (Amazon SNS) topic to send a notification to your email when a labeling job status changes.

Prerequisites:

To create a CloudWatch Events rule, you will need an AWS Identity and Access Management (IAM) role with an events.amazonaws.com trust policy attached. The following is an example of an events.amazonaws.com trust policy.

```json
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "",
      "Effect": "Allow",
      "Principal": {
        "Service": ["events.amazonaws.com"]
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

Topics

- Send Events to CloudWatch Events (p. 425)
- Set Up a Target to Process Events (p. 426)
- Labeling Job Expiration (p. 427)

Send Events to CloudWatch Events

To configure a CloudWatch Events rule to get status updates, or events, for your Ground Truth labeling jobs, use the AWS Command Line Interface (AWS CLI) `put-rule` command. You can filter events that are sent to your rule by status change. For example, you can create a rule that notifies you only if a labeling
job status changes to Completed. When using the put-rule command, specify the following to receive labeling job statuses:

- "source": ["aws.sagemaker"]
- "detail-type": ["SageMaker Ground Truth Labeling Job State Change"]

To configure a CloudWatch Events rule to watch for all status changes, use the following command and replace the placeholder text. For example, replace "GTLabelingJobStateChanges" with a unique CloudWatch Events rule name and "arn:aws:iam::111122223333:role/MyRoleForThisRule" with the Amazon Resource Number (ARN) of an IAM role with an events.amazonaws.com trust policy attached.

```
aws events put-rule --name "GTLabelingJobStateChanges"
  --event-pattern "{"source":["aws.sagemaker"],"detail-type":["SageMaker Ground Truth Labeling Job State Change"]}"
  --role-arn "arn:aws:iam::111122223333:role/MyRoleForThisRule"
  --region "region"
```

To filter by job status, use the "detail":{"LabelingJobStatus":["Status"]} syntax. Valid values for Status are Completed, Failed, and Stopped.

The following example creates a CloudWatch Events rule that notifies you when a labeling job in us-west-2 (Oregon) changes to Completed.

```
aws events put-rule --name "LabelingJobCompleted"
  --event-pattern "{"source":["aws.sagemaker"],"detail-type":["SageMaker Ground Truth Labeling Job State Change"], "detail":{"LabelingJobStatus":["Completed"]}"
  --role-arn "arn:aws:iam::111122223333:role/MyRoleForThisRule"
  --region us-west-2
```

The following example creates a CloudWatch Events rule that notifies you when a labeling job in us-east-1 (Virginia) changes to Completed or Failed.

```
aws events put-rule --name "LabelingJobCompletedOrFailed"
  --event-pattern "{"source":["aws.sagemaker"],"detail-type":["SageMaker Ground Truth Labeling Job State Change"], "detail":{"LabelingJobStatus":["Completed", "Failed"]}"
  --role-arn "arn:aws:iam::111122223333:role/MyRoleForThisRule"
  --region us-east-1
```

To learn more about the put-rule request, see Event Patterns in CloudWatch Events in the Amazon CloudWatch Events User Guide.

### Set Up a Target to Process Events

After you have created a rule, events similar to the following are sent to CloudWatch Events. In this example, the labeling job test-labeling-job's status changed to Completed.

```
{
  "version": "0",
  "id": "111e1111-11d1-111f-b111-111b11d1cb11",
  "detail-type": "SageMaker Ground Truth Labeling Job State Change",
  "source": "aws.sagemaker",
  "account": "111122223333",
  "time": "2018-10-06T12:26:13Z",
  "region": "us-east-1",
  "resources": ["arn:aws:sagemaker:us-east-1:111122223333:labeling-job/test-labeling-job"
```
To process events, you need to set up a target. For example, if you want to receive an email when your labeling job status changes, use a procedure in Setting Up Amazon SNS Notifications in the Amazon CloudWatch User Guide to set up an Amazon SNS topic and subscribe your email to it. Once you have create a topic, you can use it to create a target.

**To add a target to your CloudWatch Events rule**

1. Open the CloudWatch console: https://console.aws.amazon.com/cloudwatch/home
2. In the navigation pane, choose Rules.
3. Choose the rule that you want to add a target to.
4. Choose Actions, and then choose Edit.
5. Under Targets, choose Add Target and choose the AWS service you want to act when a labeling job status change event is detected.
6. Configure your target. For instructions, see the topic for configuring a target in the AWS documentation for that service.
7. Choose Configure details.
8. For Name, enter a name and, optionally, provide details about the purpose of the rule in Description.
9. Make sure that the check box next to State is selected so that your rule is listed as Enabled.
10. Choose Update rule.

**Labeling Job Expiration**

If your labeling job is not completed after 30 days, it will expire. If your labeling job expires, you can chain the job to create a new labeling job that will only send unlabeled data to workers. For more information, and to learn how to create a labeling job using chaining, see Chaining Labeling Jobs (p. 412).

**Create and Manage Workforces**

*This feature is not available in the China Regions.*

A *workforce* is the group of workers that you have selected to label your dataset. You can choose either the Amazon Mechanical Turk workforce, a vendor-managed workforce, or you can create your own private workforce to label or review your dataset. Whichever workforce type you choose, Amazon SageMaker takes care of sending tasks to workers.

When you use a private workforce, you also create *work teams*, a group of workers from your workforce that are assigned to specific jobs—Amazon SageMaker Ground Truth labeling jobs or Amazon Augmented AI human review tasks. You can have multiple work teams and can assign one or more work teams to each job.

You can use Amazon Cognito or your own private OpenID Connect (OIDD) Identity Provider (IdP) to manage your private workforce and work teams. For more information about the permissions required to manage your workforce this way, see Permissions Required to Use the Amazon SageMaker Ground Truth Console (p. 1628).

**Topics**
Using the Amazon Mechanical Turk Workforce

The Amazon Mechanical Turk workforce provides the most workers for your Amazon Augmented AI task review and Amazon SageMaker Ground Truth labeling job.

You can use the console to choose the Amazon Mechanical Turk workforce for your Amazon SageMaker Ground Truth labeling job or Amazon Augmented AI human review workflow, or you can provide the Amazon Resource Name (ARN) for the Amazon Mechanical Turk workforce when you use the Amazon A2I CreateLabelingJob operation.

Any Amazon Mechanical Turk workforce billing is handled as part of your Ground Truth or Amazon Augmented AI billing. You do not need to create a separate Mechanical Turk account to use the Amazon Mechanical Turk workforce.

The ARN for the Amazon Mechanical Turk workforce is:


The Amazon Mechanical Turk workforce is a world-wide resource. Workers are available 24 hours a day, 7 days a week. You typically get the fastest turn-around for your human review tasks and labeling jobs when you use the Amazon Mechanical Turk workforce.

Adjust the number of workers that annotate each data object based on the complexity of the job and the quality that you need. Amazon SageMaker Ground Truth uses annotation consolidation to improve the quality of the labels. More workers can make a difference in the quality of the labels for more complex labeling jobs, but might not make a difference for simpler jobs. For more information, see Consolidate Annotations (p. 405). Annotation consolidation is not supported for Amazon Augmented AI human review workflows.

Important
You should not share confidential information, personal information or protected health information with this workforce. For avoidance of doubt, you should not use the Amazon Mechanical Turk workforce when you use Amazon A2I in conjunction with AWS HIPAA-eligible services, such as Amazon Textract and Amazon Rekognition for workloads containing protected health information.

To choose the Amazon Mechanical Turk workforce when you are creating a labeling job or human review workflow using the console, do the following during the Select workers and configure tool step:

To use the Amazon Mechanical Turk workforce

1. Choose Amazon Mechanical Turk from Worker types.
2. Choose The dataset does not contain adult content if your dataset doesn't contain potentially offensive content. This enables workers to opt out if they don't want to work with it.
3. Acknowledge that your data will be viewed by the Amazon Mechanical Turk workforce and that all personally identifiable information (PII) has been removed.
4. Choose Additional configuration to set optional parameters.
5. Optional. Enable automated data labeling to have Ground Truth automatically label some of your dataset. For more information, see Automate Data Labeling (p. 406). Automated data labeling is not available for Amazon Augmented AI.
6. Optional. Set the number of workers that should see each object in your dataset. Using more workers can increase the quality of your labels but also increases the cost.
Managing Vendor Workforces

You can use a vendor-managed workforce to label your data using Amazon SageMaker Ground Truth (Ground Truth) and Amazon Augmented AI (Amazon A2I). Vendors have extensive experience in providing data labeling services for the purpose of machine learning. Vendor workforces for these two services must be created and managed separately through the Amazon SageMaker console.

Vendors make their services available via the AWS Marketplace. You can find details of the vendor's services on their detail page, such as the number of workers and the hours that they work. You can use these details to make estimates of how much the labeling job will cost and the amount of time that you can expect the job to take. Once you have chosen a vendor you subscribe to their services using the AWS Marketplace.

A subscription is an agreement between you and the vendor. The agreement spells out the details of the agreement, such as price, schedule, or refund policy. You work directly with the vendor if there are any issues with your labeling job.

You can subscribe to any number of vendors to meet your data annotation needs. When you create a labeling job or human review workflow you can specify that the job be routed to a specific vendor.

**Important**
Before you send sensitive data to a vendor, check the vendor's security and compliance practices on their detail page and review the end user license agreement (EULA) that is part of your subscription agreement. You are responsible for ensuring that the vendor meets your compliance requirements for personal or confidential information. Do not share protected health information with this workforce.

You must use the console to subscribe to a vendor workforce. Once you have a subscription, you can use the `ListSubscribedWorkteams` operation to list your subscribed vendors.

**To subscribe to a vendor workforce**

2. Choose the appropriate page in the SageMaker console.
   - For Ground Truth labeling jobs, choose **Labeling workforces**, choose **Vendor**, and then choose **Find data labeling services**.
   - For Amazon A2I human review workflows, choose **Human review workforces**, choose **Vendor**, and then choose **Find human review services**.
3. The console opens the AWS Marketplace with:
   - data labeling services category selected for Ground Truth
   - human review services category selected for Amazon A2I

   Here you see a list of the vendor services available for this service.
4. Choose a vendor. The AWS Marketplace shows detailed information about the data labeling or human review service. Use this information to determine if the vendor meets your requirements for your task.
5. If the vendor meets your requirements, choose **Continue to subscribe**.
6. Review the details of the subscription. If you agree to the terms, choose **Subscribe** to complete your subscription to the service.
Use a Private Workforce

A **private workforce** is a group of workers that you choose. These can be employees of your company or a group of subject matter experts from your industry. For example, if the task is to label medical images, you could create a private workforce of people knowledgeable about the images in question.

Each AWS account has access to a single private workforce per region, and the owner has the ability to create multiple **private work teams** within that workforce. A single private work team is used to complete a labeling job or human review task, or a job. You can assign each work team to a separate job or use a single team for multiple jobs. A single worker can be in more than one work team.

Your private workforce can either be created and managed using Amazon Cognito or your own private OpenID Connect (OIDC) Identity Provider (IdP).

If you are a new user of Amazon SageMaker Ground Truth or Amazon Augmented AI and do not require your workers to be managed with your own IdP, it is recommended that you use Amazon Cognito to create and manage your private workforce.

After you create a workforce, in addition to creating and managing work teams, you can do the following:

- Track worker performance
- Create and manage Amazon SNS topics to notify workers when labeling tasks are available
- Manage Private Workforce Access to Tasks Using IP Addresses

**Note**

Your private workforce is shared between Ground Truth and Amazon A2I. To create and manage private work teams used by Augmented AI, use the Ground Truth section of the SageMaker console.

**Topics**

- Create and Manage Amazon Cognito Workforce (p. 430)
- Create and Manage OIDC IdP Workforce (p. 438)
- Manage Private Workforce Using the Amazon SageMaker API (p. 445)
- Track Worker Performance (p. 446)
- Create and manage Amazon SNS topics for your work teams (p. 447)

Create and Manage Amazon Cognito Workforce

Create and manage your private workforce using Amazon Cognito when you want to create your workforce using the Amazon SageMaker console or you don’t want the overhead of managing worker credentials and authentication. When you create a private workforce with Amazon Cognito, it provides authentication, authorization, and user management for your private workers.

**Topics**

- Create a Private Workforce (Amazon Cognito) (p. 430)
- Manage a Private Workforce (Amazon Cognito) (p. 433)

Create a Private Workforce (Amazon Cognito)

When you use Amazon Cognito, you can create a private workforce in one of the following ways:
• Create a new workforce while you are creating your labeling job. To learn how, see Create an Amazon Cognito Workforce When Creating a Labeling Job (p. 431).

• Create a new workforce before you create your labeling job. To learn how, see Create an Amazon Cognito Workforce Using the Labeling Workforces Page (p. 432).

• Import an existing workforce after creating a user pool in the Amazon Cognito console. To learn how, see Create a Private Workforce (Amazon Cognito Console) (p. 432).

Once you create a private workforce, that workforce and all work teams and workers associated with it are available to use for all Ground Truth labeling job tasks and Amazon Augmented AI human review workflows tasks.

If you are new to Amazon SageMaker and want to test Ground Truth or Amazon A2I, we suggest that you create a private work team consisting of people from your organization using the console. Use this work team when creating labeling or human review workflows (flow definitions) to test your worker UI and job workflow.

Topics

• Create a Private Workforce (Amazon SageMaker Console) (p. 431)
• Create a Private Workforce (Amazon Cognito Console) (p. 432)

Create a Private Workforce (Amazon SageMaker Console)

You can create a private workforce in the Amazon SageMaker console in one of two ways:

• When creating a labeling job in the Labeling jobs page of the Amazon SageMaker Ground Truth section.

• Using the Labeling workforces page of the Amazon SageMaker Ground Truth section. If you are creating a private workforce for an Amazon A2I human review workflow, use this method.

Both of these methods also create a default work team containing all of the members of the workforce. This private workforce is available to use for both Ground Truth and Amazon Augmented AI jobs.

When you create a private workforce using the console, SageMaker uses Amazon Cognito as an identity provider for your workforce. If you want to use your own OpenID Connect (OIDC) Identity Provider (IdP) to create and manage your private workforce, you must create a workforce using the SageMaker API operation CreateWorkforce. To learn more, see Create a Private Workforce (OIDC IdP) (p. 438).

Create an Amazon Cognito Workforce When Creating a Labeling Job

If you haven't created a private workforce when you create your labeling job and you choose to use private workers, you are prompted to create a work team. This will create a private workforce using Amazon Cognito.

To create a workforce while creating a labeling job (console)

2. In the navigation pane, choose Labeling jobs and fill in all required fields. For instructions on how to start a labeling job, see Getting started (p. 162). Choose Next.
3. Choose Private for the workforce type.
4. In the Workers section, enter:
   a. The Team name.
b. Email addresses for up to 100 workforce members. Email addresses are case sensitive. Your workers must log in using the same case used when the address was initially entered. You can add additional workforce members after the job has been created.

c. The name of your organization. SageMaker uses this to customize the email sent to the workers.

d. A contact email address for workers to report issues related to the task.

When you create the labeling job, an email is sent to each worker inviting them to join the workforce. After creating the workforce, you can add, delete, and disable workers using the SageMaker console or the Amazon Cognito console.

Create an Amazon Cognito Workforce Using the Labeling Workforces Page

To create and manage your private workforce using Amazon Cognito, you can use the Labeling workforces page. When following the instructions below, you have the option to create a private workforce by entering worker emails importing a pre-existing workforce from an Amazon Cognito user pool. To import a workforce, see Create a Private Workforce (Amazon Cognito Console) (p. 432).

To create a private workforce using worker emails

2. In the navigation pane, choose Labeling workforces.
3. Choose Private, then choose Create private team.
4. Choose Invite new workers by email.
5. Paste or type a list of up to 50 email addresses, separated by commas, into the email addresses box.
6. Enter an organization name and contact email.
7. Optionally, choose an SNS topic to which to subscribe the team so workers are notified by email when new Ground Truth labeling jobs become available. Amazon SNS notifications are supported by Ground Truth and are not supported by Augmented AI. If you subscribe workers to receive SNS notifications, they only receive notifications about Ground Truth labeling jobs. They do not receive notifications about Augmented AI tasks.
8. Click the Create private team button.

After you import your private workforce, refresh the page. On the Private workforce summary page, you can see information about the Amazon Cognito user pool for your workforce, a list of work teams for your workforce, and a list of all of the members of your private workforce.

Note
If you delete all of your private work teams, you have to repeat this process to use a private workforce in that region.

Create a Private Workforce (Amazon Cognito Console)

Amazon Cognito is used to define and manage your private workforce and your work teams. It is a service that you can use to create identities for your workers and authenticate these identities with identity providers. A private workforce corresponds to a single Amazon Cognito user pool. Private work teams correspond to Amazon Cognito user groups within that user pool.

Example identity providers supported by Amazon Cognito:

- Social sign-in providers such as Facebook and Google
- OpenID Connect (OIDC) providers
- Security Assertion Markup Language (SAML) providers such as Active Directory
- The Amazon Cognito built-in identity provider

For more information, see What Is Amazon Cognito?
To create a private workforce using Amazon Cognito, you must have an existing Amazon Cognito user pool containing at least one user group. See Tutorial: Creating a User Pool to learn how to create a user pool. See Adding Groups to a User Pool to learn how to add a user group to a pool.

Once your user pool has been created, follow the steps below to create a private workforce by importing that user pool into Amazon SageMaker.

**To create a private workforce by importing a Amazon Cognito user pool**

2. In the navigation pane, choose Labeling workforces.
3. Choose Private.
4. Choose Create private team. This creates a private workforce and a work team.
5. Choose Import workers from existing Amazon Cognito user groups.
6. Choose a user pool that you have created. User pools require a domain and an existing user group. If you get an error that the domain is missing, set it in the Domain name options on the App integration page of the Amazon Cognito console for your group.
7. Choose an app client. We recommend using a client generated by SageMaker.
8. Choose a user group from your pool to import its members.
9. Optionally choose an Amazon Simple Notification Service (Amazon SNS) topic to which to subscribe the team so that workers are notified by email when new labeling jobs become available. Amazon SNS notifications are supported by Ground Truth and are not supported by Augmented AI. If you subscribe workers to receive SNS notifications, they only receive notifications about Ground Truth labeling jobs. They do not receive notifications about Augmented AI tasks.
10. Choose Create private team.

**Important**
After you create a workforce using an Amazon Cognito user pool, it should not be deleted without first deleting all work teams associated with that pool in the SageMaker console.

After you import your private workforce, refresh the page to see the Private workforce summary page. On this page, you can see information about the Amazon Cognito user pool for your workforce, a list of work teams for your workforce, and a list of all of the members of your private workforce. This workforce is now available to use in both Amazon Augmented AI and Amazon SageMaker Ground Truth for human review tasks and data labeling jobs respectively.

**Manage a Private Workforce (Amazon Cognito)**

After you have created a private workforce using Amazon Cognito, you can create and manage work teams using the Amazon SageMaker console and API operations.

You can do the following using either the SageMaker console or Amazon Cognito console.

- Add and delete work teams.
- Add workers to your workforce and one or more work teams.
- Disable or remove workers from your workforce and one or more workteams. If you add workers to a workforce using the Amazon Cognito console, you must use the same console to remove the worker from the workforce.

You can restrict access to tasks to workers at specific IP addresses using the SageMaker API. For more information, see Manage Private Workforce Using the Amazon SageMaker API (p. 445).

**Topics**
Use a Private Workforce

Manage a Workforce (Amazon SageMaker Console)

You can use the Amazon SageMaker console to create and manage the work teams and individual workers that make up a private workforce.

Use a work team to assign members of your private workforce to a labeling or human review job. When you create your workforce using the SageMaker console, there is a work team called **Everyone-in-private-workforce** that enables you to assign your entire workforce to a job. Because an imported Amazon Cognito user pool may contain members that you don’t want to include in your work teams, a similar work team is not created for Amazon Cognito user pools.

You have two choices to create a new work team:

- You can create a work team in the SageMaker console and add members from your workforce to the team.
- You can create a user group by using the Amazon Cognito console and then create a work team by importing the user group. You can import more than one user group into each work team. You manage the members of the work team by updating the user group in the Amazon Cognito console. See Manage a Private Workforce (Amazon Cognito Console) (p. 436) for more information.

Create a Work Team Using the SageMaker Console

You can create a new Amazon Cognito user group or import an existing user group using the SageMaker console, on the **Labeling workforces** page. For more information on creating a user group in the Amazon Cognito console, see Manage a Private Workforce (Amazon Cognito Console) (p. 436).

To create a work team using the SageMaker console

1. Open the SageMaker console at https://console.aws.amazon.com/sagemaker/
2. Choose **Labeling workforces** from the left menu.
3. Under **Private**, choose **Create private team**.
4. Under **Team details**, enter a **Team name**. The name must be unique in your account in an AWS Region.
5. Under **Add workers**, choose a method to add workers to the team using a user group.

   - If you chose **Create a team by adding workers to a new Amazon Cognito user group**, select the workers to add to the team.
   - If you chose **Create a team by importing existing Amazon Cognito user groups**, choose the user groups that are part of the new team.
6. If you select an **SNS topic**, all workers added to the team are subscribed to the Amazon SNS topic and notified when new work items are available to the team. Select from a list of your existing Ground Truth related Amazon SNS topics or select **Create new topic** to open a topic-creation dialog.

   Amazon SNS notifications are supported by Ground Truth and are not supported by Augmented AI. If you subscribe workers to receive SNS notifications, they only receive notifications about Ground Truth labeling jobs. They do not receive notifications about Augmented AI tasks.

   Workers in a workteam subscribed to a topic receive notifications when a new Ground Truth labeling job for that team becomes available and when one is about to expire.

   Read **Create and manage Amazon SNS topics for your work teams** (p. 447) for more information about using Amazon SNS topic.
Subscriptions

After you have created a work team, you can see more information about the team and change or set the Amazon SNS topic to which its members are subscribed by visiting the Amazon Cognito console. If you added any team members before you subscribed the team to a topic, you need to manually subscribe those members to that topic. Read Create and manage Amazon SNS topics for your work teams for more information on creating and managing the Amazon SNS topic.

Add or Remove Workers

A work team is a group of workers within your workforce to whom you can assign jobs. A worker can be added to more than one work team. Once a worker has been added to a work team, that worker can be disabled or removed.

Add Workers to the Workforce

Adding a worker to the workforce enables you to add that worker to any work team within that workforce.

To add workers using the private workforce summary page

1. Open the Amazon SageMaker console at https://console.aws.amazon.com/sagemaker/.
2. Choose Labeling workforces to navigate to your private workforce summary page.
3. Choose Private.
5. Paste or type a list of email addresses, separated by commas, into the email addresses box. You can have up to 50 email addresses in this list.

Add a Worker to a Work Team

A worker must be added to the workforce before being added to a work team. To add a worker to a work team, first navigate to the Private workforce summary page using the steps above.

To add a worker to a work team from the private workforce summary page

1. In the Private teams section, choose the team to which you want to add the workers.
2. Choose the Workers tab.
3. Choose Add workers to team and choose the boxes next to the workers that you want to add.
4. Click Add workers to team.

Disable and Remove a Worker from the Workforce

Disabling a worker stops the worker from receiving a job. This action does not remove the worker from the workforce, or from any work team with which the worker is associated. To disable or remove a worker from a work team, first navigate to the private workforce summary page using the steps above.

To deactivate a worker using the private workforce summary page

1. In the Workers section, choose the worker that you would like to disable.
2. Choose Disable.

If desired, you can subsequently Enable a worker after they have been disabled.

You can remove workers from your private workforce directly in the SageMaker console if that worker was added in this console. If you added the worker (user) in the Amazon Cognito console, see Manage
a Private Workforce (Amazon Cognito Console) (p. 436) to learn how to remove the worker in the Amazon Cognito console.

**To remove a worker using the private workforce summary page**

1. In the Workers section, choose the worker that you would like to delete.
2. If the worker has not been disabled, choose Disable.
3. Select the worker and choose Delete.

**Manage a Private Workforce (Amazon Cognito Console)**

A private workforce corresponds to a single Amazon Cognito user pool. Private work teams correspond to Amazon Cognito user groups within that user pool. Workers correspond to Amazon Cognito users within those groups.

After your workforce has been created, you can add work teams and individual workers through the Amazon Cognito console. You can also delete workers from your private workforce or remove them from individual teams in the Amazon Cognito console.

**Important**

You can't delete work teams from the Amazon Cognito console. Deleting a Amazon Cognito user group that is associated with an Amazon SageMaker work team will result in an error. To remove work teams, use the SageMaker console.

**Create Work Teams (Amazon Cognito Console)**

You can create a new work team to complete a job by adding a Amazon Cognito user group to the user pool associated with your private workforce. To add a Amazon Cognito user group to an existing worker pool, see Adding groups to a User Pool.

**To create a work team using an existing Amazon Cognito user group**

2. In the navigation pane, choose Workforces.
3. For Private teams, choose Create private team.
4. Under Team details, give the team a name. The name must be unique in your account in an AWS Region.
5. For Add workers, choose Import existing Amazon Cognito user groups, and choose one or more user groups that are part of the new team.
6. If you choose an SNS topic, all workers added to the team are subscribed to the Amazon Simple Notification Service (Amazon SNS) topic and notified when new work items are available to the team. Choose from a list of your existing SNS topics related to SageMaker Ground Truth or Amazon Augmented AI or choose Create new topic to create one.

**Note**

Amazon SNS notifications are supported by Ground Truth and are not supported by Augmented AI. If you subscribe workers to receive SNS notifications, they only receive notifications about Ground Truth labeling jobs. They do not receive notifications about Augmented AI tasks.

**Subscriptions**

After you have created a work team, you can see more information about the team and change or set the SNS topic to which its members are subscribed using the Amazon Cognito console. If you added any team members before you subscribed the team to a topic, you need to manually subscribe those
members to that topic. For more information, see Create and manage Amazon SNS topics for your work
teams (p. 447).

Add and Remove Workers (Amazon Cognito Console)

When using the Amazon Cognito console to add workers to a work team, you must add a user to the user
pool associated with the workforce before adding that user to a user group. Users can be added to a user
pool in various ways. For more information, see Signing Up and Confirming User Accounts.

Add a Worker to a Work Team

After a user has been added to a pool, the user can be associated with user groups inside of that pool.
After a user has been added to a user group, that user becomes a worker on any work team created using
that user group.

To add a user to a user group

1. Open the Amazon Cognito console: https://us-east-2.console.aws.amazon.com/Amazon Cognito/
home.
2. Choose Manage User Pools.
3. Choose the user pool associated with your SageMaker workforce.
4. Under General Settings, choose Users and Groups and do one of the following:
   • Choose Groups, choose the group that you want to add the user to, and choose Add users.
     Choose the users that you want to add by choosing the plus-icon to the right of the user’s
     name.
   • Choose Users, choose the user that you want to add to the user group, and choose Add to
group. From the dropdown menu, choose the group and choose Add to group.

Disable and Remove a Worker From a Work Team

Disabling a worker stops the worker from receiving jobs. This action doesn’t remove the worker from the
workforce, or from any work team the worker is associated with. To remove a user from a work team in
Amazon Cognito, you remove the user from the user group associated with that team.

To deactivate a worker (Amazon Cognito console)

1. Open the Amazon Cognito console: https://us-east-2.console.aws.amazon.com/Amazon Cognito/
home.
2. Choose Manage User Pools.
3. Choose the user pool associated with your SageMaker workforce.
4. Under General Settings, choose Users and Groups.
5. Choose the user that you want to disable.
6. Choose Disable User.

You can enable a disabled user by choosing Enable User.

To remove a user from a user group (Amazon Cognito console)

1. Open the Amazon Cognito console: https://us-east-2.console.aws.amazon.com/cognito/home.
2. Choose Manage User Pools.
3. Choose the user pool associated with your SageMaker workforce.
4. Under General Settings, choose Users and Groups.
5. For User tab, choose the X icon to the right of the group from which you want to remove the user.
Create and Manage OIDC IdP Workforce

Create a private workforce using an OpenID Connect (OIDC) Identity Provider (IdP) when you want to manage and authenticate your workers using your own OIDC IdP. Individual worker credentials and other data will be kept private. Ground Truth and Amazon A2I will only have visibility into worker information you provide through the claims that you send to these services. To create a workforce using an OIDC IdP, your IdP must support groups because Ground Truth and Amazon A2I map one or more groups in your IdP to a work team. To learn more, see Send Required and Optional Claims to Ground Truth and Amazon A2I (p. 439).

If you are a new user of Ground Truth or Amazon A2I, you can test your worker UI and job workflow by creating a private work team and adding yourself as a worker. Use this work team when you create a labeling job or human review workflow. First, create a private OIDC IdP workforce using the instructions in Create a Private Workforce (OIDC IdP) (p. 438). Next, refer to Manage a Private Workforce (OIDC IdP) (p. 442) to learn how to create a work team.

Topics
- Create a Private Workforce (OIDC IdP) (p. 438)
- Manage a Private Workforce (OIDC IdP) (p. 442)

Create a Private Workforce (OIDC IdP)

Create a private workforce using an OpenID Connect (OIDC) Identity Provider (IdP) when you want to authenticate and manage workers using your own identity provider. Use this page to learn how to configure your IdP to communicate with Amazon SageMaker Ground Truth (Ground Truth) or Amazon Augmented AI (Amazon A2I) and to learn how to create a workforce using your own IdP.

To create a workforce using an OIDC IdP, your IdP must support groups because Ground Truth and Amazon A2I use one or more groups that you specify to create work teams. You use work teams to specify workers for your labeling jobs and human review tasks. Because groups are not a standard claim, your IdP may have a different naming convention for a group of users (workers). Therefore, you must identify one or more user groups to which a worker belongs using the custom claim sagemaker:groups that is sent to Ground Truth or Amazon A2I from your IdP. To learn more, see Send Required and Optional Claims to Ground Truth and Amazon A2I (p. 439).

You create an OIDC IdP workforce using the SageMaker API operation CreateWorkforce. Once you create a private workforce, that workforce and all work teams and workers associated with it are available to use for all Ground Truth labeling job tasks and Amazon A2I human review workflows tasks. To learn more, see Create an OIDC IdP Workforce (p. 441).

Configure your OIDC IdP

How you configure your OIDC IdP depends on the IdP you use, and your business requirements.

When you configure your IdP, you must to specify a callback or redirect URI. After Ground Truth or Amazon A2I authenticates a worker, this URI will redirect the worker to the worker portal where the workers can access labeling or human review tasks. To create a worker portal URL, you need to create a workforce with your OIDC IdP details using the CreateWorkforce API operation. Specifically, you must configure your OIDC IdP with required custom sagemaker claims (see the next section for more details). Therefore, it is recommended that you configure your OIDC with a place-holder redirect URI, and then update the URI after you create the workforce. See Create an OIDC IdP Workforce (p. 441) to learn how to create a workforce using this API.

You can view your worker portal URL in the SageMaker Ground Truth console, or using the SageMaker API operation, DescribeWorkforce. The worker portal URL is in the SubDomain parameter in the response.

Important
Make sure you add the workforce subdomain to your OIDC IdP allow list.
To view your worker portal URL after creating a private workforce (Console):

2. In the navigation pane, choose **Labeling workforces**.
3. Select the **Private** tab.
4. In **Private workforce summary** you will see **Labeling portal sign-in URL**. This is your worker portal URL.

To view your worker portal URL after creating a private workforce (API):

When you create a private workforce using **CreateWorkforce**, you specify a **WorkforceName**. Use this name to call **DescribeWorkforce**. The following table includes examples of requests using the AWS CLI and AWS SDK for Python (Boto3).

**SDK for Python (Boto3)**

```python
response = client.describe_workforce(WorkforceName='string')
print(f'The workforce subdomain is: {response['SubDomain']}')
```

**AWS CLI**

```
# C:\> describe-workforce --workforce-name 'string'
```

Send Required and Optional Claims to Ground Truth and Amazon A2I

When you use your own IdP, Ground Truth and Amazon A2I use your **Issuer**, **ClientId**, and **ClientSecret** to authenticate workers by obtaining an authentication CODE from your **AuthorizationEndpoint**.

Ground Truth and Amazon A2I will use this CODE to obtain a custom claim from either your IdP's **TokenEndpoint** or **UserInfoEndpoint**. You can either configure **TokenEndpoint** to return a JSON web token (JWT) or **UserInfoEndpoint** to return a JSON object. The JWT or JSON object must contain required and optional claims that you specify. A **claim** is a key-value pair that contains information about a worker or metadata about the OIDC service. The following table lists the claims that must be included, and that can optionally be included in the JWT or JSON object that your IdP returns.

**Note**
Some of the parameters in the following table can be specified using a : or a -. For example, you can specify the groups a worker belongs to using **sagemaker:groups** or **sagemaker-groups** in your claim.

<table>
<thead>
<tr>
<th>Name</th>
<th>Required</th>
<th>Accepted Format and Values</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sagemaker:groups</code> or <code>sagemaker-groups</code></td>
<td>Yes</td>
<td>Data type:</td>
<td>Assigns a worker to one or more groups. Groups are used to map the worker into work teams.</td>
<td>Example of worker that belongs to a single group: &quot;work_team1&quot;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Example of a worker that belongs to more than one groups: [&quot;work_team1&quot;, &quot;work_team2&quot;]</td>
</tr>
</tbody>
</table>
Use a Private Workforce

<table>
<thead>
<tr>
<th>Name</th>
<th>Required</th>
<th>Accepted Format and Values</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Allowable characters:</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Regex: [p{L}\p{M}\p{S}\p{N}\p{P}]^+</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Quotas:</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>10 groups per worker</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>63 characters per group name</td>
<td></td>
</tr>
<tr>
<td>sagemaker:sub or sagemaker-sub</td>
<td>Yes</td>
<td>Data type: String</td>
<td>This is mandatory to track a worker identity inside the Ground Truth platform for auditing and to identify tasks worked on by that worker. For ADFS: Customers must use the Primary Security Identifier (SID).</td>
<td>&quot;111011101-123456789-3687056437-1111&quot;</td>
</tr>
<tr>
<td>sagemaker:client_id or sagemaker-client_id</td>
<td>Yes</td>
<td>Data type: String</td>
<td>A client ID. All tokens must be issued for this client ID.</td>
<td>&quot;00b600bb-1f00-05d0-bd00-00be00fbd0e0&quot;</td>
</tr>
<tr>
<td>sagemaker:name or sagemaker-name</td>
<td>Yes</td>
<td>Data type: String</td>
<td>The worker name to be displayed in the worker portal.</td>
<td>&quot;Jane Doe&quot;</td>
</tr>
</tbody>
</table>
Use a Private Workforce

<table>
<thead>
<tr>
<th>Name</th>
<th>Required</th>
<th>Accepted Format and Values</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>email</td>
<td>No</td>
<td>Data type: String</td>
<td>The worker email. Ground Truth uses this email to notify workers that they have been invited to work on labeling tasks. Ground Truth will also use this email to notify your workers when labeling tasks become available if you set up an Amazon SNS topic for a work team that this worker is on.</td>
<td>&quot;<a href="mailto:example-email@domain.com">example-email@domain.com</a>&quot;</td>
</tr>
<tr>
<td>email_verified</td>
<td>No</td>
<td>Data type: Bool</td>
<td>Indicates if the user email was verified or not.</td>
<td>True</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Accepted Values: True, False</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The following an example of the JSON object syntax your UserInfoEndpoint can return.

```json
{
  "sub": "122",
  "exp": "10000",
  "sagemaker-groups": ["group1", "group2"],
  "sagemaker-name": "name",
  "sagemaker-sub": "122",
  "sagemaker-client_id": "123456"
}
```

Ground Truth or Amazon A2I compares the groups listed in sagemaker:groups or sagemaker-groups to verify that your worker belongs to the work team specified in the labeling job or human review task. After the work team has been verified, labeling or human review tasks are sent to that worker.

Create an OIDC IdP Workforce

You can create a workforce using the SageMaker API operation CreateWorkforce and associated language-specific SDKs. Specify a WorkforceName and information about your OIDC IDP in the parameter OidcConfig. The following shows an example of the request. See CreateWorkforce to learn more about each parameter in this request.

```json
CreateWorkforceRequest: {
  # required fields
  WorkforceName: "example-oidc-workforce",
  OidcConfig: {
    ClientId: "clientId",
    ClientSecret: "secret",
    Issuer: "https://example-oidc-idp.com/adfs",
    TokenEndpoint: "https://example-oidc-idp.com/adfs/oauth2/token",
    UserInfoEndpoint: "https://example-oidc-idp.com/adfs/oauth2/userInfo",
  }
}
```
SourceIpConfig: {
  Cidrs: ["string", "string"]
}
}

Next Steps

After you create your OIDC IdP, make sure you add the workforce subdomain to your OIDC IdP allow list.

Once you've created a private workforce using your IdP, you can create work teams using your IdP groups. To learn more, see Manage a Private Workforce (OIDC IdP) (p. 442).

You can restrict worker access to tasks to specific IP addresses, and update or delete your workforce using the SageMaker API. To learn more, see Manage Private Workforce Using the Amazon SageMaker API (p. 445).

Manage a Private Workforce (OIDC IdP)

Once you've created a private workforce using your OpenID Connect (OIDC) Identity Provider (IdP), you can manage your workers using your IdP. For example, you can add, remove, and group workers directly through your IdP.

To add workers to an Amazon SageMaker Ground Truth (Ground Truth) labeling job or Amazon Augmented AI (Amazon A2I) human review task, you create work teams using 1-10 IdP groups and assign that work team to the job or task. You assign a work team to a job or task by specifying that work team when you create a labeling job (Ground Truth) or a human review workflow (Amazon A2I).

You can only assign one team to each labeling job or human review workflow. You can use the same team to create multiple labeling jobs or human review tasks. You can also create multiple work teams to work on different labeling jobs or human review tasks.

Prerequisites

To create and manage private work teams using your OIDC IdP groups, first you must create a workforce using the SageMaker API operation `CreateWorkforce`. To learn more, see Create a Private Workforce (OIDC IdP) (p. 438).

Add work teams

You can use the SageMaker console to create a private work team using your OIDC IdP workforce on the Labeling workforces page under Ground Truth. If you are creating a Ground Truth labeling job, you can also create a private work team while creating a labeling job.

Note
You create and manage work teams for Amazon A2I in the Ground Truth area of the SageMaker console.

You can also use the SageMaker API and associated language-specific SDKs to create a private work team.

Use the following procedures to learn how to create a private work team using the SageMaker console and API.

To create a private work team on the Labeling workforces page (console)

2. Select **Labeling workforces**.
3. Select **Private**.
4. In the **Private teams** section, select **Create private team**.
5. In the **Team details** section, enter a **Team name**.
6. In the **Add workers** section, enter the name of a single user group. All workers associated with this group in your IdP are added to this work team.
7. To add more than one user group, select **Add new user group** and enter the names of the user groups you want to add to this work team. Enter one user group per line.
8. (Optional) For Ground Truth labeling jobs, if you provide an email for workers in your JWT, Ground Truth notifies workers when a new labeling task is available if you select an SNS topic.
9. Select **Create private team**.

**To create a private work team while creating a Ground Truth labeling job (console)**

2. Select **Labeling jobs**.
3. Use the instructions in **Create a Labeling Job (Console)** (p. 314) to create a labeling job. Stop when you get to the **Workers** section on the second page.
4. Select **Private** for your worker type.
5. Enter a **Team name**.
6. In the **Add workers** section, enter the name of a single user group under **User groups**. All workers associated with this group in your IdP are added to this work team.

**Important**

The group names you specify for **User groups** must match the group names specified in your OIDC IdP.
7. To add more than one user group, select **Add new user group** and enter the names of the user groups you want to add to this work team. Enter one user group per line.
8. Complete all remaining steps to create your labeling job.

The private team that you create is used for this labeling job, and is listed in the **Labeling workforces** section of the SageMaker console.

**To create a private work team using the SageMaker API**

You can create a private work team using the SageMaker API operation **CreateWorkteam**.

When you use this operation, list all user groups that you want included in the work team in the OidcMemberDefinition parameter **Groups**.

**Important**

The group names you specify for **Groups** must match the group names specified in your OIDC IdP.

For example, if your user group names are **group1**, **group2**, and **group3** in your OIDC IdP, configure **OidcMemberDefinition** as follows:

```json
"OidcMemberDefinition": {
   "Groups": ["group1", "group2", "group3"]
}
```

Additionally, you must give the work team a name using the **WorkteamName** parameter.
Add or remove IdP groups from work teams

After you've created a work team, you can use the SageMaker API to manage that work team. Use the `UpdateWorkteam` operation to update the IdP user groups included in that work team.

- Use the `WorkteamName` parameter to identify the work team that you want to update.
- When you use this operation, list all user groups that you want included in the work team in the `OidcMemberDefinition` parameter `Groups`. If a user group is associated with a work team and you do not include it in this list, that user group is no longer associated with this work team.

Delete a work team

You can delete a work team using the SageMaker console and SageMaker API.

**To delete a private work team in the SageMaker console**

2. Select **Labeling workforces**.
3. Select **Private**.
4. In the **Private teams** section, select the work team that you want to delete.
5. Select **Delete**.

**To delete a private work team (API)**

You can delete a private work team using the SageMaker API operation `DeleteWorkteam`.

Manage Individual Workers

When you create a workforce using your own OIDC IdP, you cannot use Ground Truth or Amazon A2I to manage individual workers.

- To add a worker to a work team, add that worker to a group associated with that work team.
- To remove a worker from a work team, remove that worker from all user groups associated with that work team.

Update, Delete, and Describe Your Workforce

You can update, delete, and describe your OIDC IdP workforce using the SageMaker API. The following is a list of API operations that you can use to manage your workforce. For additional details, including how you can locate your workforce name, see Manage Private Workforce Using the Amazon SageMaker API (p. 445).

- **UpdateWorkforce** – You may want to update a workforce created using your own OIDC IdP to specify a different authorization endpoint, token endpoint, or issuer. You can update any parameter found in `OidcConfig` using this operation.

  You can only update your OIDC IdP configuration when there are no work teams associated with your workforce. To learn how to delete work teams, see Delete a work team (p. 444).

- **DeleteWorkforce** – Use this operation to delete your private workforce. If you have any work teams associated with your workforce, you must delete those work teams before you delete your workforce.

  For more information, see Delete a work team (p. 444).

- **DescribeWorkforce** – Use this operation to list private workforce information, including workforce name, Amazon Resource Name (ARN), and, if applicable, allowed IP address ranges (CIDRs).
Manage Private Workforce Using the Amazon SageMaker API

You can use Amazon SageMaker API operations to manage, update, and delete your private workforce. For each API operation linked on this page, you can find a list of supported language-specific SDKs and their documentation in the **See Also** section of the API documentation.

Find Your Workforce Name

Some of the SageMaker workforce-related API operations require your workforce name as input. You can see your Amazon Cognito or OIDC IdP private and vendor workforce names in an AWS Region using the **ListWorkforces** API operation in that AWS Region.

If you created your workforce using your own OIDC IdP, you can find your workforce name in the Ground Truth area of the SageMaker console.

**To find your workforce name in the SageMaker console**

2. Select **Labeling workforces**.
3. Select **Private**.
4. In the **Private workforce summary** section, locate your workforce ARN. Your workforce name is located at the end of this ARN. For example, if the ARN is `arn:aws:sagemaker:us-east-2:111122223333:workforce/example-workforce`, the workforce name is `example-workforce`.

Restrict Worker Access to Tasks to Allowable IP Addresses

By default, a workforce isn't restricted to specific IP addresses. You can use the **UpdateWorkforce** operation to require that workers use a specific range of IP addresses (CIDRs) to access tasks. If you specify one or more CIDRs, workers who attempt to access tasks using any IP address outside the specified ranges are denied and get a **Not Found** error message on the worker portal. You can specify up to 10 CIDR values using **UpdateWorkforce**.

After you have restricted your workforce to one or more CIDRs, the output of **UpdateWorkforce** lists all allowable CIDRs. You can also use the **DescribeWorkforce** operation to view all allowable CIDRs for a workforce.

Update OIDC Identity Provider Workforce Configuration

You may want to update a workforce created using your own OIDC IdP to specify a different authorization endpoint, token endpoint, or issuer. You can update any parameter found in **OidcConfig** using the **UpdateWorkforce** operation.

**Important**

You can only update your OIDC IdP configuration when there are no work teams associated with your workforce. You can delete a private work team using the **DeleteWorkteam** operation.

Delete a Private Workforce

You can only have one private workforce in each AWS Region. You may want to delete your private workforce in an AWS Region when:

- You want to create a workforce using a new Amazon Cognito user pool.
- You have already created a private workforce using Amazon Cognito and you want to create a workforce using your own OpenID Connect (OIDC) Identity Provider (IdP).
To delete a private workforce, use the `DeleteWorkforce` API operation. If you have any work teams associated with your workforce, you must delete those work teams before you delete your workforce. You can delete a private work team using the `DeleteWorkteam` operation.

**Track Worker Performance**

Amazon SageMaker Ground Truth logs worker events to Amazon CloudWatch, such as when a worker starts or submits a task. Use Amazon CloudWatch metrics to measure and track throughput across a team or for individual workers.

**Important**
Worker event tracking is not available for Amazon Augmented AI human review workflows.

**Enable Tracking**

During the set-up process for a new work team, the permissions for Amazon CloudWatch logging of worker events are created. Since this feature was added in August 2019, work teams created prior to that may not have the correct permissions. If all of your work teams were created before August 2019, create a new work team. It does not need any members and may be deleted after creation, but by creating it, you establish the permissions and apply them to all of your work teams, regardless of when they were created.

**Examine Logs**

After tracking is enabled, the activity of your workers is logged. Open the Amazon CloudWatch console and choose `Logs` in the navigation pane. You should see a log group named `/aws/sagemaker/groundtruth/WorkerActivity`.

Each completed task is represented by a log entry, which contains information about the worker, their team, the job, when the task was accepted, and when it was submitted.

**Example Log entry**

```json
{
 "worker_id": "cd449a289e129409",
 "cognito_user_pool_id": "us-east-2_IpicJXXX",
 "cognito_sub_id": "d6947aeb-0650-447a-ab5d-894db61017fd",
 "task_accepted_time": "Wed Aug 14 16:00:59 UTC 2019",
 "task_submitted_time": "Wed Aug 14 16:01:04 UTC 2019",
 "task_returned_time": "",
 "workteam_arn": "arn:aws:sagemaker:us-east-2:############:workteam/private-crowd/Sample-labeling-team",
 "labeling_job_arn": "arn:aws:sagemaker:us-east-2:############:labeling-job/metrics-demo",
 "work_requester_account_id": "############",
 "job_reference_code": "############",
 "job_type": "Private",
 "event_type": "TasksSubmitted",
 "event_timestamp": "1565798464"
}
```

A useful data point in each event is the `cognito_sub_id`. You can match that to an individual worker.

2. Under the **Ground Truth** section, choose **Workforces**.
3. Choose **Private**.
4. Choose the name of a team in the **Private teams** section.
5. In the **Team summary** section, choose the user group identified under **Amazon Cognito user group**. That will take you to the group in the Amazon Cognito console.
6. The Group page lists the users in the group. Choose any user's link in the Username column to see more information about the user, including a unique sub ID.

To get information about all of the team's members, use the ListUsers action (examples) in the Amazon Cognito API.

**Use Log Metrics**

If you don't want to write your own scripts to process and visualize the raw log information, Amazon CloudWatch metrics provide insights into worker activity for you.

**To view metrics**

2. In the navigation pane, choose Metrics.
3. Choose the AWS/SageMaker/Workteam name space, then explore the available metrics (p. 1704). For example, selecting the Workflow, Workteam metrics lets you calculate the average time per submitted task for a specific labeling job.

For more information, see Using Amazon CloudWatch Metrics.

**Create and manage Amazon SNS topics for your work teams**

Use the procedures in this topic when you want to:

- Create a topic to which you want an existing work team to subscribe.
- Create a topic before you've created a work team.
- Create or modify the work team with an API call, and specify a topic Amazon Resource Name (ARN).

If you create a work team using the console, the console provides an option to create a new topic for the team so that you don't have to perform these steps.

**Important**

The Amazon SNS feature is not supported by Amazon A2I. If you subscribe your work team to an Amazon SNS topic, workers will only receive notifications about Ground Truth labeling jobs. Workers will not receive notifications about new Amazon A2I human review tasks.

**Create the Amazon SNS topic**

The steps for creating Amazon SNS topics for work team notifications are similar to the steps in Getting Started in the Amazon SNS Developer Guide, with one significant addition—you must add an access policy so that Amazon SageMaker can publish messages to the topic on your behalf.

**To add the policy when you create the topic**

1. Open the Amazon SNS console at https://console.aws.amazon.com/sns/.
2. In Create topic, enter the name of your topic and then choose Next steps.
4. In the JSON editor, find the Resource property, which displays the topic's ARN.
5. Copy the Resource ARN value.
6. Before the final closing brace (}), add the following policy.

```json
, 
```
7. Create the topic.

After you create the topic, it appears in your Topics summary screen. For more information about creating topics, see Creating a Topic in the Amazon SNS Developer Guide.

Manage worker subscriptions

If you subscribe a work team to a topic after you've already created the work team, the individual work team members who were added to the team when the work team was created are not automatically subscribed to the topic. For information about subscribing workers' email addresses to the topic, see Subscribing an Endpoint to an Amazon SNS Topic in the Amazon SNS Developer Guide.

The only situation in which workers are automatically subscribed to your topic is when you create or import an Amazon Cognito user group at the time that you create a work team and you set up the topic subscription when you create that work team. For more information about creating and managing your workteams with Amazon Cognito, see Create Work Teams (Amazon Cognito Console) (p. 436).

Crowd HTML Elements Reference

This feature is not available in the China Regions.

Crowd HTML Elements are web components, a web standard that abstracts HTML markup, CSS, and JavaScript functionality into an HTML tag or set of tags. Amazon SageMaker provides customers with the ability to design their own custom task templates in HTML.

As a starting point, you can use a template built using Crowd HTML Elements from one of the following GitHub repositories:

- Example task UIs for Amazon SageMaker Ground Truth
- Over 60 example task UIs for Amazon Augmented AI (A2I)

These repositories include templates designed for audio, image, text, video, and other types of data labeling and annotation tasks.

For more information about how to implement custom templates in Amazon SageMaker Ground Truth, see Creating Custom Labeling Workflows (p. 291). To learn more about custom templates in Amazon Augmented AI, see Create Custom Worker Task Template (p. 1566).

SageMaker Crowd HTML Elements

Following is a list of Crowd HTML Elements that make building a custom template easier and provide a familiar UI for workers. These elements are supported in Ground Truth, Augmented AI, and Mechanical Turk.

Topics
crowd-alert (p. 449)
crowd-badge (p. 451)
crowd-button (p. 452)
crowd-bounding-box (p. 454)
crowd-card (p. 458)
crowd-checkbox (p. 460)
crowd-classifier (p. 462)
crowd-classifier-multi-select (p. 464)
crowd-entity-annotation (p. 465)
crowd-fab (p. 469)
crowd-form (p. 470)
crowd-icon-button (p. 471)
crowd-image-classifier (p. 473)
crowd-image-classifier-multi-select (p. 476)
crowd-input (p. 478)
crowd-instance-segmentation (p. 480)
crowd-instructions (p. 484)
crowd-keypoint (p. 486)
crowd-line (p. 489)
crowd-modal (p. 492)
crowd-polygon (p. 493)
crowd-polyline (p. 499)
crowd-radio-button (p. 502)
crowd-radio-group (p. 504)
crowd-semantic-segmentation (p. 506)
crowd-slider (p. 509)
crowd-tab (p. 511)
crowd-tabs (p. 513)
crowd-text-area (p. 515)
crowd-toast (p. 516)
crowd-toggle-button (p. 517)

crowd-alert

A message that alerts the worker to a current situation.

The following is an example of a Liquid template that uses the <crowd-alert> element. Copy the following code and save it in a file with the extension .html. Open the file in any browser to preview and interact with this template.

```
<script src="https://assets.crowd.aws/crowd-html-elements.js"></script>
<crowd-form>
  <div id="errorBox"></div>
</crowd-form>
```
<crowd-keypoint
  src="{{ task.input.taskObject | grant_read_access }}"
  labels="['Item A', 'Item B', 'Item C']"
  header="Please locate the centers of each item."
  name="annotatedResult">
  <short-instructions>
    Describe your task briefly here and give examples
  </short-instructions>
  <full-instructions>
    Give additional instructions and good/bad examples here
  </full-instructions>
</crowd-keypoint>
</crowd-form>

<script>
  var num_obj = 1;

  document.querySelector('crowd-form').onsubmit = function(e) {
    const keypoints = document.querySelector('crowd-keypoint').value.keypoints ||
      document.querySelector('crowd-keypoint')._submittableValue.keypoints;
    const labels = keypoints.map(function(p) {
      return p.label;
    });

    // 1. Make sure total number of keypoints is correct.
    var original_num_labels = document.getElementsByTagName("crowd-keypoint")[0].getAttribute("labels");
    original_num_labels = original_num_labels.substring(2, original_num_labels.length - 2).split("", ",");
    var goalNumKeypoints = num_obj*original_num_labels.length;
    if (keypoints.length != goalNumKeypoints) {
      e.preventDefault();
      errorBox.innerHTML = '<crowd-alert type="error" dismissible>You must add all keypoint annotations and use each label only once.</crowd-alert>';
      errorBox.scrollIntoView();
      return;
    }

    // 2. Make sure all labels are unique.
    labelCounts = {};
    for (var i = 0; i < labels.length; i++) {
      if (!labelCounts[labels[i]]) {
        labelCounts[labels[i]] = 0;
      }
      labelCounts[labels[i]]++;
    }
    const goalNumSingleLabel = num_obj;
    const numLabels = Object.keys(labelCounts).length;
    Object.entries(labelCounts).forEach(entry => {
      if (entry[1] != goalNumSingleLabel) {
        e.preventDefault();
        errorBox.innerHTML = '<crowd-alert type="error" dismissible>You must use each label only once.</crowd-alert>';
        errorBox.scrollIntoView();
      }
    });
  }
</script>

Attributes

The following attributes are supported by this element.
dismissible

A Boolean switch that, if present, allows the message to be closed by the worker.

type

A string that specifies the type of message to be displayed. The possible values are "info" (the default), "success", "error", and "warning".

Element Hierarchy

This element has the following parent and child elements.

- Parent elements: crowd-form (p. 470)
- Child elements: none

See Also

For more information, see the following.

- Use Amazon SageMaker Ground Truth to Label Data (p. 161)
- Crowd HTML Elements Reference (p. 448)

crowd-badge

An icon that floats over the top right corner of another element to which it is attached.

The following is an example of a template that uses the <crowd-badge> element. Copy the following code and save it in a file with the extension .html. Open the file in any browser to preview and interact with this template.

```html
<script src="https://assets.crowd.aws/crowd-html-elements.js"></script>

<crowd-form>
  <crowd-image-classifier
    name="crowd-image-classifier"
    src="https://unsplash.com/photos/NLUkAA-nDdE"
    header="Choose the correct category for this image."
    categories="['Person', 'Umbrella', 'Chair', 'Dolphin']"
  >
    <full-instructions header="Classification Instructions">
      <p>Read the task carefully and inspect the image.</p>
      <p>Choose the appropriate label that best suits the image.</p>
    </full-instructions>
  
    <short-instructions id="short-instructions">
      <p>Read the task carefully and inspect the image.</p>
      <p>Choose the appropriate label that best suits the image.</p>
    </short-instructions>
  </crowd-image-classifier>
</crowd-form>
```

Attributes

The following attributes are supported by this element.
for

A string that specifies the ID of the element to which the badge is attached.

icon

A string that specifies the icon to be displayed in the badge. The string must be either the name of an icon from the open-source iron-icons set, which is pre-loaded, or the URL to a custom icon.

This attribute overrides the label attribute.

The following is an example of the syntax that you can use to add an iron-icon to a <crowd-badge> HTML element. Replace icon-name with the name of the icon you'd like to use from this Icons set.

```html
<crowd-badge icon="icon-name" for="short-instructions"/>
```

label

The text to display in the badge. Three characters or less is recommended because text that is too large will overflow the badge area. An icon can be displayed instead of text by setting the icon attribute.

Element Hierarchy

This element has the following parent and child elements.

- Parent elements: crowd-form (p. 470)
- Child elements: none

See Also

For more information, see the following.

- Use Amazon SageMaker Ground Truth to Label Data (p. 161)
- Crowd HTML Elements Reference (p. 448)

crowd-button

A styled button that represents some action.

Attributes

The following attributes are supported by this element.

The following is an example of a template that uses the <crowd-button> element. Copy the following code and save it in a file with the extension .html. Open the file in any browser to preview and interact with this template.

```html
<script src="https://assets.crowd.aws/crowd-html-elements.js"></script>

<crowd-form>
  <crowd-image-classifier
    name="crowd-image-classifier"
    src="https://unsplash.com/photos/NLUkAA--nDdE"
    header="Please select the correct category for this image"
    categories="['Person', 'Umbrella', 'Chair', 'Dolphin']"
  />
</crowd-form>
```
disabled

A Boolean switch that, if present, displays the button as disabled and prevents clicks.

form-action

A switch that either submits its parent <crowd-form> element, if set to "submit", or resets its parent <crowd-form> element, if set to "reset".

href

The URL to an online resource. Use this property if you need a link styled as a button.

icon

A string that specifies the icon to be displayed next to the button's text. The string must be the name of an icon from the open-source iron-icons set, which is pre-loaded. For example, to insert the search iron-icon, use the following:

```xml
<iron-icon icon="search"/>
</crowd-button>
```

The icon is positioned to either the left or the right of the text, as specified by the icon-align attribute.

To use a custom icon see icon-url.

icon-align

The left or right position of the icon relative to the button's text. The default is "left".

icon-url

A URL to a custom image for the icon. A custom image can be used in place of a standard icon that is specified by the icon attribute.

loading

A Boolean switch that, if present, displays the button as being in a loading state. This attribute has precedence over the disabled attribute if both attributes are present.

target

When you use the href attribute to make the button act as a hyperlink to a specific URL, the target attribute optionally targets a frame or window where the linked URL should load.
variant

The general style of the button. Use "primary" for primary buttons, "normal" for secondary buttons, "link" for tertiary buttons, or "icon" to display only the icon without text.

Element Hierarchy

This element has the following parent and child elements.

- **Parent elements**: crowd-form (p. 470)
- **Child elements**: none

See Also

For more information, see the following.

- Use Amazon SageMaker Ground Truth to Label Data (p. 161)
- Crowd HTML Elements Reference (p. 448)

crowd-bounding-box

A widget for drawing rectangles on an image and assigning a label to the portion of the image that is enclosed in each rectangle.

The following is an example of a Liquid template that uses the `<crowd-bounding-box>` element. Copy the following code and save it in a file with the extension `.html`. Open the file in any browser to preview and interact with this template. For more examples, see this [GitHub repository](#).

```html
<script src="https://assets.crowd.aws/crowd-html-elements.js"></script>

<crowd-form>
  <crowd-bounding-box
    name="annotatedResult"
    src="{{ task.input.taskObject | grant_read_access }}"
    header="Draw bounding boxes around all the cats and dogs in this image"
    labels="['Cat', 'Dog']"
  >
    <full-instructions header="Bounding Box Instructions" >
      <p>Use the bounding box tool to draw boxes around the requested target of interest:</p>
      <ol>
        <li>Draw a rectangle using your mouse over each instance of the target.</li>
        <li>Make sure the box does not cut into the target, leave a 2 - 3 pixel margin</li>
        <li>When targets are overlapping, draw a box around each object, include all contiguous parts of the target in the box. Do not include parts that are completely overlapped by another object.</li>
        <li>Do not include parts of the target that cannot be seen, even though you think you can interpolate the whole shape of the target.</li>
        <li>Avoid shadows, they're not considered as a part of the target.</li>
        <li>If the target goes off the screen, label up to the edge of the image.</li>
      </ol>
    </full-instructions>
    <short-instructions>
      Draw boxes around the requested target of interest.
    </short-instructions>
  </crowd-bounding-box>
</crowd-form>
```
Attributes

The following attributes are supported by this element.

header

The text to display above the image. This is typically a question or simple instruction for the worker.

initial-value

An array of JSON objects, each of which sets a bounding box when the component is loaded. Each JSON object in the array contains the following properties. Bounding boxes set via the initial-value property can be adjusted and whether or not a worker answer was adjusted is tracked via an initialValueModified boolean in the worker answer output.

- **height** – The height of the box in pixels.
- **label** – The text assigned to the box as part of the labeling task. This text must match one of the labels defined in the labels attribute of the <crowd-bounding-box> element.
- **left** – Distance of the top-left corner of the box from the left side of the image, measured in pixels.
- **top** – Distance of the top-left corner of the box from the top of the image, measured in pixels.
- **width** – The width of the box in pixels.

You can extract the bounding box initial value from a manifest file of a previous job in a custom template using the Liquid templating language:

```liquid
initial-value="[{ % for box in task.input.manifestLine.label-attribute-name-from-prior-job.annotations %} {% capture class_id %}{{ box.class_id }}{% endcapture %} {% assign label = task.input.manifestLine.label-attribute-name-from-prior-job-metadata.class-map[class_id] %} { label: {{label | to_json}}, left: {{box.left}}, top: {{box.top}}, width: {{box.width}}, height: {{box.height}}, }, { % endfor %}]"
```

labels

A JSON formatted array of strings, each of which is a label that a worker can assign to the image portion enclosed by a rectangle. **Limit**: 10 labels.

name

The name of this widget. It's used as a key for the widget's input in the form output.

src

The URL of the image on which to draw bounding boxes.
Element Hierarchy

This element has the following parent and child elements.

- **Parent elements**: crowd-form (p. 470)
- **Child elements**: full-instructions (p. 456), short-instructions (p. 456)

Regions

The following regions are required by this element.

- full-instructions
  - General instructions about how to draw bounding boxes.
- short-instructions
  - Important task-specific instructions that are displayed in a prominent place.

Output

The following output is supported by this element.

- boundingBoxes
  - An array of JSON objects, each of which specifies a bounding box that has been created by the worker. Each JSON object in the array contains the following properties.
    - **height** – The height of the box in pixels.
    - **label** – The text assigned to the box as part of the labeling task. This text must match one of the labels defined in the labels attribute of the <crowd-bounding-box> element.
    - **left** – Distance of the top-left corner of the box from the left side of the image, measured in pixels.
    - **top** – Distance of the top-left corner of the box from the top of the image, measured in pixels.
    - **width** – The width of the box in pixels.

- inputImageProperties
  - A JSON object that specifies the dimensions of the image that is being annotated by the worker. This object contains the following properties.
    - **height** – The height, in pixels, of the image.
    - **width** – The width, in pixels, of the image.

Example: Sample Element Outputs

The following are samples of outputs from common use scenarios for this element.

**Single Label, Single Box / Multiple Label, Single Box**

```json
[
  {
    "annotatedResult": {
      "boundingBoxes": [
        {
          "height": 401,
          "label": "Dog",
```
Single Label, Multiple Box

[{
  "annotatedResult": {
    "boundingBoxes": [
      {
        "height": 401,
        "label": "Dog",
        "left": 243,
        "top": 117,
        "width": 187
      },
      {
        "height": 283,
        "label": "Dog",
        "left": 684,
        "top": 120,
        "width": 116
      }
    ],
    "inputImageProperties": {
      "height": 533,
      "width": 800
    }
  }
}]

Multiple Label, Multiple Box

[
  {
    "annotatedResult": {
      "boundingBoxes": [
        {
          "height": 395,
          "label": "Dog",
          "left": 241,
          "top": 125,
          "width": 158
        },
        {
          "height": 298,
          "label": "Cat",
          "left": 699,
          "top": 116,
          "width": 101
        }
      ],
      "inputImageProperties": {
        "height": 533,
        "width": 800
      }
    }
  }
]
"inputImageProperties": {
  "height": 533,
  "width": 800
}
}
]

You could have many labels available, but only the ones that are used appear in the output.

**See Also**

For more information, see the following.

- Use Amazon SageMaker Ground Truth to Label Data (p. 161)
- Crowd HTML Elements Reference (p. 448)

**crowd-card**

A box with an elevated appearance for displaying information.

The following is an example of a template designed for sentiment analysis tasks that uses the `<crowd-card>` element. Copy the following code and save it in a file with the extension `.html`. Open the file in any browser to preview and interact with this template.

```html
<script src="https://assets.crowd.aws/crowd-html-elements.js"></script>

<style>
  h3 { margin-top: 0; }
  crowd-card {
    width: 100%;
  }
  .card {
    margin: 10px;
  }
  .left {
    width: 70%;
    margin-right: 10px;
    display: inline-block;
    height: 200px;
  }
  .right {
    width: 20%;
    height: 200px;
    display: inline-block;
  }
</style>

<crowd-form>
  <short-instructions>
    Your short instructions here.
  </short-instructions>
  <full-instructions>
    Your full instructions here.
  </full-instructions>
</crowd-form>
```
Attributes

The following attributes are supported by this element.

heading

The text displayed at the top of the box.

image

A URL to an image to be displayed within the box.

Element Hierarchy

This element has the following parent and child elements.

- **Parent elements**: crowd-form (p. 470)
- **Child elements**: none
Amazon SageMaker Developer Guide
SageMaker Crowd HTML Elements

See Also
For more information, see the following.
• Use Amazon SageMaker Ground Truth to Label Data (p. 161)
• Crowd HTML Elements Reference (p. 448)

crowd-checkbox
A UI component that can be checked or unchecked allowing a user to select multiple options from a set.
The following is an example of a Liquid template that uses the <crowd-checkbox> element. Copy the
following code and save it in a ﬁle with the extenion .html. Open the ﬁle in any browser to preview and
interact with this template.
<script src="https://assets.crowd.aws/crowd-html-elements.js"></script>
<crowd-form>
<p>Find the official website for: <strong>{{ task.input.company }}</strong></p>
<p>Do not give Yelp pages, LinkedIn pages, etc.</p>
<p>Include the http:// prefix from the website</p>
<crowd-input name="website" placeholder="http://example.com"></crowd-input>
<crowd-checkbox name="website-found">Website Found</crowd-checkbox>
</crowd-form>

Attributes
The following attributes are supported by this element.

checked
A Boolean switch that, if present, displays the check box as checked.
The following is an example of the syntx used to check a checkbox by default.
<crowd-checkbox name="checkedBox" value="checked" checked>This box is checked</crowdcheckbox>

disabled
A Boolean switch that, if present, displays the check box as disabled and prevents it from being checked.
The following is an example of the syntax used to disable a checkbox.
<crowd-checkbox name="disabledCheckBox" value="Disabled" disabled>Cannot be selected</
crowd-checkbox>

name
A string that is used to identify the answer submitted by the worker. This value will match a key in the
JSON object that speciﬁes the answer.

required
A Boolean switch that, if present, requires the worker to provide input.

460


The following is an example of the syntax used to require a checkbox be selected.

```
<crowd-checkbox name="work_verified" required>Instructions were clear</crowd-checkbox>
```

**value**

A string used as the name for the check box state in the output. Defaults to "on" if not specified.

**Element Hierarchy**

This element has the following parent and child elements.

- **Parent elements**: crowd-form (p. 470)
- **Child elements**: none

**Output**

Provides a JSON object. The name string is the object name and the value string is the property name for a Boolean value based on the check box state; true if checked, false if not checked.

**Example: Sample Element Outputs**

**Using the same name value for multiple boxes.**

```
<!-- INPUT -->
<div><crowd-checkbox name="image_attributes" value="blurry"> Blurry </crowd-checkbox></div>
<div><crowd-checkbox name="image_attributes" value="dim"> Too Dim </crowd-checkbox></div>
<div><crowd-checkbox name="image_attributes" value="exposed"> Too Bright </crowd-checkbox></div>
```

//Output with "blurry" and "dim" checked

```
[  
  {  
    "image_attributes": {  
      "blurry": true,  
      "dim": true,  
      "exposed": false  
    }  
  } 
]
```

Note that all three color values are properties of a single object.

**Using different name values for each box.**

```
<!-- INPUT -->
<div><crowd-checkbox name="Stop" value="Red"> Red </crowd-checkbox></div>
<div><crowd-checkbox name="Slow" value="Yellow"> Yellow </crowd-checkbox></div>
<div><crowd-checkbox name="Go" value="Green"> Green </crowd-checkbox></div>
```

//Output with "Red" checked

```
[  
  {  
    "Go": {  
      "Green": false  
    },  
    "Slow": {  
      "Yellow": false  
    }  
  } 
]
```
See Also

For more information, see the following.

- Use Amazon SageMaker Ground Truth to Label Data (p. 161)
- Crowd HTML Elements Reference (p. 448)

**crowd-classifier**

A widget for classifying non-image content, such as audio, video, or text.

The following is an example of an HTML worker task template built using `crowd-classifier`. This example uses the Liquid template language to automate:

- Label categories in the `categories` parameter
- The objects that are being classified in the `classification-target` parameter.

Copy the following code and save it in a file with the extension `.html`. Open the file in any browser to preview and interact with this template.

```html
<script src="https://assets.crowd.aws/crowd-html-elements.js"></script>

<crowd-form>
  <crowd-classifier
    name="category"
    categories="{{ task.input.labels | to_json | escape }}"
    header="What type of a document is this?"
  >
    <classification-target>
      <iframe style="width: 100%; height: 600px;" src="{{ task.input.taskObject | grant_read_access }}" type="application/pdf"></iframe>
    </classification-target>

    <full-instructions header="Document Classification Instructions">
      <p>Read the task carefully and inspect the document.</p>
      <p>Choose the appropriate label that best suits the document.</p>
    </full-instructions>

    <short-instructions>
      Please choose the correct category for the document
    </short-instructions>

  </crowd-classifier>
</crowd-form>
```

**Attributes**

The following attributes are supported by this element.

**categories**

A JSON formatted array of strings, each of which is a category that a worker can assign to the text. You should include “other” as a category, otherwise the worker may not be able to provide an answer.
header

The text to display above the image. This is typically a question or simple instruction for the worker.

name

The name of this widget. It is used as a key for the widget's input in the form output.

Element Hierarchy

This element has the following parent and child elements.

- **Parent elements**: crowd-form (p. 470)
- **Child elements**: classification-target (p. 463), full-instructions (p. 463), short-instructions (p. 463)

Regions

The following regions are supported by this element.

**classification-target**

The content to be classified by the worker. This can be plain text or HTML. Examples of how the HTML can be used include *but are not limited to* embedding a video or audio player, embedding a PDF, or performing a comparison of two or more images.

**full-instructions**

General instructions about how to do text classification.

**short-instructions**

Important task-specific instructions that are displayed in a prominent place.

Output

The output of this element is an object using the specified name value as a property name, and a string from the categories as the property's value.

Example: Sample Element Outputs

The following is a sample of output from this element.

```
[{
   "<name>": {
      "label": "<value>
   }
}
```

See Also

For more information, see the following.

- Use Amazon SageMaker Ground Truth to Label Data (p. 161)
- Crowd HTML Elements Reference (p. 448)
**crowd-classifier-multi-select**

A widget for classifying various forms of content—such as audio, video, or text—into one or more categories. The content to classify is referred to as an object.

The following is an example of an HTML worker task template built using this crowd element. Copy the following code and save it in a file with the extension .html. Open the file in any browser to preview and interact with this template.

```html
<script src="https://assets.crowd.aws/crowd-html-elements.js"></script>

<crowd-form>
  <crowd-classifier-multi-select
    name="category"
    categories="['Positive', 'Negative', 'Neutral']"
    header="Select the relevant categories"
    exclusion-category="{ text: 'None of the above' }"
  >
    <classification-target>
      {{ task.input.taskObject }}
    </classification-target>

    <full-instructions header="Text Categorization Instructions">
      <p><strong>Positive</strong> sentiment include: joy, excitement, delight</p>
      <p><strong>Negative</strong> sentiment include: anger, sarcasm, anxiety</p>
      <p><strong>N/A</strong>: when the text cannot be understood</p>
      <p>When the sentiment is mixed, such as both joy and sadness, choose both labels.</p>
    </full-instructions>

    <short-instructions>Choose all categories that are expressed by the text.</short-instructions>
  </crowd-classifier-multi-select>
</crowd-form>
```

**Attributes**

The following attributes are supported by the `crowd-classifier-multi-select` element. Each attribute accepts a string value or string values.

- **categories**
  
  Required. A JSON-formatted array of strings, each of which is a category that a worker can assign to the object.

- **header**
  
  Required. The text to display above the image. This is typically a question or simple instruction for workers.

- **name**
  
  Required. The name of this widget. In the form output, the name is used as a key for the widget’s input.

- **exclusion-category**
  
  Optional. A JSON-formatted string with the following format: `"{ text: 'default-value' }"`. This attribute sets a default value that workers can choose if none of the labels applies to the object shown in the worker UI.
Element Hierarchy

This element has the following parent and child elements:

- **Parent elements**: crowd-form (p. 470)
- **Child elements**: classification-target (p. 463), full-instructions (p. 463), short-instructions (p. 463)

Regions

This element uses the following regions.

**classification-target**

The content to be classified by the worker. Content can be plain text or an object that you specify in the template using HTML. For example, you can use HTML elements to include a video or audio player, embedding a PDF file, or include a comparison of two or more images.

**full-instructions**

General instructions about how to classify text.

**short-instructions**

Important task-specific instructions. These instructions are displayed prominently.

Output

The output of this element is an object that uses the specified name value as a property name, and a string from categories as the property's value.

**Example : Sample Element Outputs**

The following is a sample of output from this element.

```
[
  {
    "<name>": {
      labels: ["label_a", "label_b"]
    }
  }
]
```

See Also

For more information, see the following:

- Text Classification (Multi-label) (p. 191)
- Use Amazon SageMaker Ground Truth to Label Data (p. 161)
- Crowd HTML Elements Reference (p. 448)

**crowd-entity-annotation**

A widget for labeling words, phrases, or character strings within a longer text. Workers select a label, and highlight the text that the label applies to.
Important: Self-contained Widget

Do not use `<crowd-entity-annotation>` element with the `<crowd-form>` element. It contains its own form submission logic and Submit button.

The following is an example of a template that uses the `<crowd-entity-annotation>` element. Copy the following code and save it in a file with the extension `.html`. Open the file in any browser to preview and interact with this template.

```html
<script src="https://assets.crowd.aws/crowd-html-elements.js"></script>

<crowd-entity-annotation
name="crowd-entity-annotation"
header="Highlight parts of the text below"
labels="[{
'label': 'person', 'shortDisplayName': 'per', 'fullDisplayName': 'Person'}, {
'label': 'date', 'shortDisplayName': 'dat', 'fullDisplayName': 'Date'}, {
'label': 'company', 'shortDisplayName': 'com', 'fullDisplayName': 'Company'}]"
text="Amazon SageMaker Ground Truth helps you build highly accurate training datasets for machine learning quickly."
>
  <full-instructions header="Named entity recognition instructions">
  <ol>
    <li><strong>Read</strong> the text carefully.</li>
    <li><strong>Highlight</strong> words, phrases, or sections of the text.</li>
    <li><strong>Choose</strong> the label that best matches what you have highlighted.</li>
    <li>To <strong>change</strong> a label, choose highlighted text and select a new label.</li>
    <li>To <strong>remove</strong> a label from highlighted text, choose the X next to the abbreviated label name on the highlighted text.</li>
    <li>You can select all of a previously highlighted text, but not a portion of it.</li>
  </ol>
  </full-instructions>

  <short-instructions>
  Apply labels to words or phrases.
  </short-instructions>

  <div id="additionalQuestions" style="margin-top: 20px">
    <h3>What is the overall subject of this text?</h3>
    <crowd-radio-group>
      <crowd-radio-button name="tech" value="tech">Technology</crowd-radio-button>
      <crowd-radio-button name="politics" value="politics">Politics</crowd-radio-button>
    </crowd-radio-group>
  </div>

<script>
document.addEventListener('all-crowd-elements-ready', () => {
  document.querySelector('crowd-entity-annotation').shadowRoot.querySelector('crowd-form').form.appendChild(additionalQuestions);
});
</script>

Attributes

The following attributes are supported by this element.
header

The text to display above the image. This is typically a question or simple instruction for the worker.

initial-value

A JSON formatted array of objects, each of which defines an annotation to apply to the text at initialization. Objects contain a label value that matches one in the labels attribute, an integer startOffset value for labeled span's starting unicode offset, and an integer endOffset value for the ending unicode offset.

Example

```
[
  {
    label: 'person',
    startOffset: 0,
    endOffset: 16
  },
  ...
]
```

labels

A JSON formatted array of objects, each of which contains:

- label (required): The name used to identify entities.
- fullDisplayName (optional): Used for the label list in the task widget. Defaults to the label value if not specified.
- shortDisplayName (optional): An abbreviation of 3-4 letters to display above selected entities. Defaults to the label value if not specified.

**shortDisplayName is highly recommended**

Values displayed above the selections can overlap and create difficulty managing labeled entities in the workspace. Providing a 3-4 character shortDisplayName for each label is highly recommended to prevent overlap and keep the workspace manageable for your workers.

Example

```
[
  {
    label: 'person',
    shortDisplayName: 'per',
    fullDisplayName: 'person'
  }
]
```

name

Serves as the widget's name in the DOM. It is also used as the label attribute name in form output and the output manifest.

text

The text to be annotated. The templating system escapes quotes and HTML strings by default. If your code is already escaped or partially escaped, see Variable filters (p. 295) for more ways to control escaping.
Element Hierarchy

This element has the following parent and child elements.

- **Child elements**: full-instructions (p. 468), short-instructions (p. 468)

Regions

The following regions are supported by this element.

**full-instructions**

General instructions about how to work with the widget.

**short-instructions**

Important task-specific instructions that are displayed in a prominent place.

Output

The following output is supported by this element.

**entities**

A JSON object that specifies the start, end, and label of an annotation. This object contains the following properties.

- **label** – The assigned label.
- **startOffset** – The Unicode offset of the beginning of the selected text.
- **endOffset** – The Unicode offset of the first character after the selection.

**Example : Sample Element Outputs**

The following is a sample of the output from this element.

```json
{
  "myAnnotatedResult": {
    "entities": [
      {
        "endOffset": 54,
        "label": "person",
        "startOffset": 47
      },
      {
        "endOffset": 97,
        "label": "event",
        "startOffset": 93
      },
      {
        "endOffset": 219,
        "label": "date",
        "startOffset": 212
      },
      {
        "endOffset": 271,
        "label": "location",
        "startOffset": 260
      }
    ]
  }
}
```
crowd-fab

A floating button with an image in its center.

The following is an example of a Liquid template designed for image classification that uses the `<crowd-fab>` element. This template uses JavaScript to enable workers to report issues with the worker UI. Copy the following code and save it in a file with the extension `.html`. Open the file in any browser to preview and interact with this template.

```html
<script src="https://assets.crowd.aws/crowd-html-elements.js"></script>
<crowd-form>
  <crowd-image-classifier src="${image_url}" categories="["Cat", "Dog", "Bird", "None of the Above"]" header="Choose the correct category for the image" name="category">
    <short-instructions>
      <p>Read the task carefully and inspect the image.</p>
      <p>Choose the appropriate label that best suits the image.</p>
      <b>None of the Above</b>, if there is an issue with the image or tools, please select
      <crowd-input label="Report an Issue" name="template-issues"></crowd-input>
      <crowd-fab id="button1" icon="report-problem" title="Issue"/>
    </short-instructions>
    <full-instructions header="Classification Instructions">
      <p>Read the task carefully and inspect the image.</p>
      <p>Choose the appropriate label that best suits the image. Use the <b>None of the Above</b> option if none of the other labels suit the image.</p>
    </full-instructions>
  </crowd-image-classifier>
</crowd-form>
<script>
  [button1, button2].forEach(function(button) {
    button.addEventListener('click', function() {
      document.querySelector('crowd-form').submit();
    });
  });
</script>

Attributes

The following attributes are supported by this element.
**disabled**

A Boolean switch that, if present, displays the floating button as disabled and prevents clicks.

**icon**

A string that specifies the icon to be displayed in the center of the button. The string must be either the name of an icon from the open-source *iron-icons* set, which is pre-loaded, or the URL to a custom icon.

The following is an example of the syntax that you can use to add an iron-icon to a `<crowd-fab>` HTML element. Replace `icon-name` with the name of the icon you'd like to use from this *Icons set*.

```html
<crowd-fab "id="button1" icon="icon-name" title="Issue"/>
```

**label**

A string consisting of a single character that can be used instead of an icon. Emojis or multiple characters may result in the button displaying an ellipsis instead.

**title**

A string that will display as a tool tip when the mouse hovers over the button.

**Element Hierarchy**

This element has the following parent and child elements.

- **Parent elements**: crowd-form (p. 470)
- **Child elements**: none

**See Also**

For more information, see the following.

- Use Amazon SageMaker Ground Truth to Label Data (p. 161)
- Crowd HTML Elements Reference (p. 448)

**crowd-form**

The form wrapper for all custom tasks. Sets and implements important actions for the proper submission of your form data.

If a crowd-button (p. 452) of type "submit" is not included inside the `<crowd-form>` element, it will automatically be appended within the `<crowd-form>` element.

The following is an example of an image classification template that uses the `<crowd-form>` element. Copy the following code and save it in a file with the extension `.html`. Open the file in any browser to preview and interact with this template.

```html
<script src="https://assets.crowd.aws/crowd-html-elements.js"></script>

<crowd-form>
    <crowd-image-classifier
        src="${image_url}"
        categories="['Cat', 'Dog', 'Bird', 'None of the Above']"
        header="Choose the correct category for the image"
        name="category"/>
</crowd-form>
```
<short-instructions>
  <p>Read the task carefully and inspect the image.</p>
  <p>Choose the appropriate label that best suits the image.</p>
</short-instructions>

<full-instructions header="Classification Instructions">
  <p>Read the task carefully and inspect the image.</p>
  <p>Choose the appropriate label that best suits the image. Use the "None of the Above" option if none of the other labels suit the image.</p>
</full-instructions>
</crowd-image-classifier>
</crowd-form>

Element Hierarchy
This element has the following parent and child elements.

- **Parent elements**: none
- **Child elements**: Any of the UI Template (p. 448) elements

Element Events
The `crowd-form` element extends the standard HTML form element and inherits its events, such as onclick and onsubmit.

See Also
For more information, see the following.

- Use Amazon SageMaker Ground Truth to Label Data (p. 161)
- Crowd HTML Elements Reference (p. 448)

crowd-icon-button
A button with an image placed in the center. When the user touches the button, a ripple effect emanates from the center of the button.

The following is an example of a Liquid template designed for image classification that uses the `<crowd-icon-button>` element. This template uses JavaScript to enable workers to report issues with the worker UI. Copy the following code and save it in a file with the extension .html. Open the file in any browser to preview and interact with this template.

```html
<script src="https://assets.crowd.aws/crowd-html-elements.js"></script>
<crowd-form>
  <crowd-image-classifier
    src="${image_url}"
    categories="['Cat', 'Dog', 'Bird', 'None of the Above']"
    header="Choose the correct category for the image"
    name="category">

  </crowd-image-classifier>

  <short-instructions>
    <p>Read the task carefully and inspect the image.</p>
  </short-instructions>
</crowd-form>
```
Choose the appropriate label that best suits the image.<p>
If there is an issue with the image or tools, please select None of the Above, describe the issue in the text box and click the button below.</p>

Attributes

The following attributes are supported by this element.

disabled

A Boolean switch that, if present, displays the button as disabled and prevents clicks.

icon

A string that specifies the icon to be displayed in the center of the button. The string must be either the name of an icon from the open-source iron-icons set, which is pre-loaded, or the URL to a custom icon.

Element Hierarchy

This element has the following parent and child elements.

- Parent elements: crowd-form (p. 470)
- Child elements: none

See Also

For more information, see the following.

- Use Amazon SageMaker Ground Truth to Label Data (p. 161)
- Crowd HTML Elements Reference (p. 448)
Amazon SageMaker Developer Guide
SageMaker Crowd HTML Elements

crowd-image-classiﬁer
A widget for classifying an image. Use one of the following supported image formats: APNG, BMP, GIF,
ICO, JPEG, PNG, SVG. Images do not have a size limit.
The following is an example of an image classiﬁcation template that uses the <crowd-imageclassifier> element. Copy the following code and save it in a ﬁle with the extension .html. Open the
ﬁle in any browser to preview and interact with this template.
<script src="https://assets.crowd.aws/crowd-html-elements.js"></script>
<crowd-form>
<crowd-image-classifier
src="${image_url}"
categories="['Cat', 'Dog', 'Bird', 'None of the Above']"
header="Choose the correct category for the image"
name="category">
<short-instructions>
<p>Read the task carefully and inspect the image.</p>
<p>Choose the appropriate label that best suits the image.</p>
</short-instructions>
<full-instructions header="Classification Instructions">
<p>Read the task carefully and inspect the image.</p>
<p>Choose the appropriate label that best suits the image.
Use the <b>None of the Above</b> option if none of the other labels suit the
image.</p>
</full-instructions>
</crowd-image-classifier>
</crowd-form>

Attributes
The following attributes are required by this element.

categories
A JSON formatted array of strings, each of which is a category that a worker can assign to the image. You
should include "other" as a category, so that the worker can provide an answer. You can specify up to 10
categories.

header
The text to display above the image. This is typically a question or simple instruction for the worker.

name
The name of this widget. It is used as a key for the widget's input in the form output.

overlay
Information to be overlaid on the source image. This is for veriﬁcation workﬂows of bounding-box,
semantic-segmentation, and instance-segmentation tasks.
It is a JSON object containing an object with the name of the task-type in camelCase as the key. That
key's value is an object that contains the labels and other necessary information from the previous task.
An example of a crowd-image-classifier element with attributes for verifying a bounding-box task
follows:

473


A semantic segmentation verification task would use the overlay value as follows:

```xml
<crowd-image-classifier
    name='crowd-image-classifier'
categories='["good", "bad"]'
src='URL of image to be classified'
header='Please classify'
overlay='{
    "semanticSegmentation": {
        "labels": ["Cat", "Dog", "Bird", "Cow"],
        "labelMappings": {
            "Bird": { "color": "#ff7f0e" },
            "Cat": { "color": "#2ca02c" },
            "Cow": { "color": "#d62728" },
            "Dog": { "color": "#2acf59" }
        },
        "src": "URL of overlay image",
    },
}> ... </crowd-image-classifier>
```

An instance-segmentation task would use the overlay value as follows:

```xml
<crowd-image-classifier
    name='crowd-image-classifier'
categories='["good", "bad"]'
overlay='{
    "semanticSegmentation": {
        "labels": ["Cat", "Dog", "Bird", "Cow"],
        "labelMappings": {
            "Bird": { "color": "#ff7f0e" },
            "Cat": { "color": "#2ca02c" },
            "Cow": { "color": "#d62728" },
            "Dog": { "color": "#2acf59" }
        },
        "src": "URL of overlay image",
    },
}> ... </crowd-image-classifier>
```
The URL of the image to be classified.

**Element Hierarchy**

This element has the following parent and child elements.

- **Parent elements**: crowd-form (p. 470)
- **Child elements**: full-instructions (p. 475), short-instructions (p. 475), worker-comment (p. 475)

**Regions**

The following regions are used by this element.

**full-instructions**

General instructions for the worker on how to classify an image.

**short-instructions**

Important task-specific instructions that are displayed in a prominent place.

**worker-comment**

Use this in verification workflows when you need workers to explain why they made the choice they did. Use the text between the opening and closing tags to provide instructions for workers on what information should be included in the comment.

It uses the following attributes:

**header**

A phrase with a call to action for leaving a comment. Used as the title text for a modal window where the comment is added.

Optional. Defaults to "Add a comment."
link-text

This text appears below the categories in the widget. When clicked, it opens a modal window where the worker may add a comment.

Optional. Defaults to "Add a comment."

placeholder

An example text in the comment text area that is overwritten when worker begins to type. This does not appear in output if the worker leaves the field blank.

Optional. Defaults to blank.

Output

The output of this element is a string that specifies one of the values defined in the categories attribute of the <crowd-image-classifier> element.

Example: Sample Element Outputs

The following is a sample of output from this element.

```json
[
  {
    "<name>": {
      "label": "<value>",
      "workerComment": "Comment - if no comment is provided, this field will not be present"
    }
  }
]
```

See Also

For more information, see the following.

- Use Amazon SageMaker Ground Truth to Label Data (p. 161)
- Crowd HTML Elements Reference (p. 448)

crowd-image-classifier-multi-select

A widget for classifying an image into one or more categories. Use one of the following supported image formats: APNG, BMP, GIF, ICO, JPEG, PNG, SVG. Images do not have a size limit.

The following is an example of an HTML worker task template built using this crowd element. Copy the following code and save it in a file with the extension .html. Open the file in any browser to preview and interact with this template.

```html
<script src="https://assets.crowd.aws/crowd-html-elements.js"></script>

<crowd-form>
  <crowd-image-classifier-multi-select
    name="animals"
    categories="['Cat', 'Dog', 'Horse', 'Pig', 'Bird']"
    src="https://images.unsplash.com/photo-1509205477838-a534e43a849f?ixlib=rb-1.2.1&ixid=eyJhcHBfaWQiOjEyMDd9&auto=format&fit=crop&w=1998&q=80"
    header="Please identify the animals in this image"
    exclusion-category="{ text: 'None of the above' }"
  ></crowd-image-classifier-multi-select>
</crowd-form>
```
<full-instructions header="Classification Instructions">
  <p>If more than one label applies to the image, select multiple labels.</p>
  <p>If no labels apply, select <b>None of the above</b></p>
</full-instructions>

<short-instructions>
  <p>Read the task carefully and inspect the image.</p>
  <p>Choose the appropriate label(s) that best suit the image.</p>
</short-instructions>

Attributes

The following attributes are supported by the crowd-image-classifier-multi-select element. Each attribute accepts a string value or string values.

categories

Required. A JSON-formatted array of strings, each of which is a category that a worker can assign to the image. A worker must choose at least one category and can choose all categories.

header

Required. The text to display above the image. This is typically a question or simple instruction for workers.

name

Required. The name of this widget. In the form output, the name is used as a key for the widget's input.

src

Required. The URL of the image to be classified.

exclusion-category

Optional. A JSON-formatted string with the following format: "{ text: 'default-value' }". This attribute sets a default value that workers can choose if none of the labels applies to the image shown in the worker UI.

Element Hierarchy

This element has the following parent and child elements:

- **Parent elements**: crowd-form (p. 470)
- **Child elements**: full-instructions (p. 475), short-instructions (p. 475), worker-comment (p. 475)

Regions

This element uses the following regions

full-instructions

General instructions for the worker on how to classify an image.

short-instructions

Important task-specific instructions. These instructions are displayed prominently.
Output

The output of this element is a string that specifies one or more of the values defined in the categories attribute of the `<crowd-image-classifier-multi-select>` element.

Example: Sample Element Outputs

The following is a sample of output from this element.

```
[
  {
    "<name>": {
      labels: ["label_a", "label_b"]
    }
  }
]
```

See Also

For more information, see the following:

- Image Classification (Multi-label) (p. 180)
- Use Amazon SageMaker Ground Truth to Label Data (p. 161)
- Crowd HTML Elements Reference (p. 448)

crowd-input

A box that accepts input data.

**Cannot be self-closing**

Unlike the `input` element in the HTML standard, this element cannot be self-closed by putting a slash before the ending bracket, e.g. `<crowd-input ... />`. It must be followed with a `</crowd-input>` to close the element.

The following is an example of a Liquid template that uses the `<crowd-input>` element. Copy the following code and save it in a file with the extension `.html`. Open the file in any browser to preview and interact with this template.

```
<script src="https://assets.crowd.aws/crowd-html-elements.js"></script>

<crowd-form>
  <img style="max-width: 35vw; max-height: 50vh" src="{{ task.input.taskObject | grant_read_access }}">
  <crowd-input name="tag1" label="Word/phrase 1" required></crowd-input>
  <crowd-input name="tag2" label="Word/phrase 2" required></crowd-input>
  <crowd-input name="tag3" label="Word/phrase 3" required></crowd-input>
  
  <short-instructions>
    Your custom quick instructions and examples
  </short-instructions>
  
  <full-instructions>
    Your custom detailed instructions and more examples
  </full-instructions>
</crowd-form>
```

Attributes

The following attributes are supported by this element.
allowed-pattern
A regular expression that is used with the `auto-validate` attribute to ignore non-matching characters as the worker types.

auto-focus
When the value is set to true, the browser places focus inside the input area after loading. This way, the worker can start typing without having to select it first.

auto-validate
A Boolean switch that, if present, turns on input validation. The behavior of the validator can be modified by the `error-message` and `allowed-pattern` attributes.

disabled
A Boolean switch that, if present, displays the input area as disabled.

error-message
The text to be displayed below the input field, on the left side, if validation fails.

label
A string that is displayed inside a text field.

This text shrinks and rises up above a text field when the worker starts typing in the field or when the `value` attribute is set.

max-length
A maximum number of characters the input will accept. Input beyond this limit is ignored.

min-length
A minimum length for the input in the field

name
Sets the name of the input to be used in the DOM and the output of the form.

placeholder
A string value that is used as placeholder text, displayed until the worker starts entering data into the input, it is not used as a default value.

required
A Boolean switch that, if present, requires the worker to provide input.

type
Takes a string to set the HTML5 `input-type` behavior for the input. Examples include `file` and `date`.

value
A preset that becomes the default if the worker does not provide input. The preset appears in a text field.

Element Hierarchy
This element has the following parent and child elements.
• Parent elements: crowd-form (p. 470)
• Child elements: none

Output

Provides a name string as the property name, and the text that was entered in the field as its value.

Example : Sample JSON Output

The values for multiple elements are output in the same object, with their name attribute value as their property name. Elements with no input do not appear in the output. For example, let's use three inputs:

```html
<crowd-input name="tag1" label="Word/phrase 1"></crowd-input>
<crowd-input name="tag2" label="Word/phrase 2"></crowd-input>
<crowd-input name="tag3" label="Word/phrase 3"></crowd-input>
```

This is the output if only two have input:

```json
[
   {
      "tag1": "blue",
      "tag2": "red"
   }
]
```

This means any code built to parse these results should be able to handle the presence or absence of each input in the answers.

See Also

For more information, see the following.

• Use Amazon SageMaker Ground Truth to Label Data (p. 161)
• Crowd HTML Elements Reference (p. 448)

crowd-instance-segmentation

A widget for identifying individual instances of specific objects within an image and creating a colored overlay for each labeled instance.

The following is an example of a Liquid template that uses the `<crowd-instance-segmentation>`.
Copy the following code and save it in a file with the extension .html. Open the file in any browser to preview and interact with this template.

```html
<script src="https://assets.crowd.aws/crowd-html-elements.js"></script>

<crowd-form
    <crowd-instance-segmentation
        name="annotatedResult"
        src="{{ task.input.taskObject | grant_read_access }}"
        header="Please label each of the requested objects in this image"
        labels="['Cat', 'Dog', 'Bird']"
    >
    <full-instructions header="Segmentation Instructions">
        <ol>
            <li><strong>Read</strong> the task carefully and inspect the image.</li>
        </ol>
    </full-instructions>
</crowd-instance-segmentation>
```
Read the options and review the examples provided to understand more about the labels.

Choose the appropriate label that best suits the image.

Use the tools to label all instances of the requested items in the image.

Use a template similar to the following to allow workers to add their own categories (labels).

```html
<ol>
  <li><strong>Read</strong> the options and review the examples provided to understand more about the labels.</li>
  <li><strong>Choose</strong> the appropriate label that best suits the image.</li>
</ol>

<p>Use the tools to label all instances of the requested items in the image.</p>

Use a template similar to the following to allow workers to add their own categories (labels).

```html
<script src="https://assets.crowd.aws/crowd-html-elements.js"></script>

<crowd-instance-segmentation id="annotator" name="myTexts"
src="{{ task.input.taskObject | grant_read_access }}"
header="Click Instructions to add new labels." labels="['placeholder']">

  <short-instructions>
    <h3>Add a label to describe each type of object in this image.</h3>
    <h3>Cover each instance of each object with a segmentation mask.</h3>
    <h3>Add new label</h3>
  </short-instructions>

  <crowd-input name="_customLabel" id="customLabel"></crowd-input>
  <crowd-button id="addLabel">Add</crowd-button>

  <br><br><br>

  <h3>Manage labels</h3>
  <div id="labelsSection"></div>

  <full-instructions>
    Describe your task in more detail here.
  </full-instructions>

</crowd-instance-segmentation>
</crowd-form>

<script>
  document.addEventListener('all-crowd-elements-ready', function(event) {
    document.querySelector('crowd-instance-segmentation').labels = [];
  });

  function populateLabelsSection() {
    labelsSection.innerHTML = '';
    annotator.labels.forEach(function(label) {
      const labelContainer = document.createElement('div');
      labelContainer.innerHTML = label + ' <a href="javascript:void(0)">(Delete)</a>';
      labelContainer.querySelector('a').onclick = function() {
        annotator.labels = annotator.labels.filter(function(l) {
          return l !== label;
        });
        populateLabelsSection();
      }
      labelsSection.appendChild(labelContainer);
    });
  }
</script>
```
addLabel.onclick = function() {
    annotator.labels = annotator.labels.concat([customLabel.value]);
    customLabel.value = null;
    populateLabelsSection();
};
</script>

Attributes

The following attributes are supported by this element.

header

The text to display above the image. This is typically a question or simple instruction for the worker.

labels

A JSON formatted array of strings, each of which is a label that a worker can assign to an instance of an object in the image. Workers can generate different overlay colors for each relevant instance by selecting "add instance" under the label in the tool.

name

The name of this widget. It is used as a key for the labeling data in the form output.

src

The URL of the image that is to be labeled.

initial-value

A JSON object containing the color mappings of a prior instance segmentation job and a link to the overlay image output by the prior job. Include this when you want a human worker to verify the results of a prior labeling job and adjust it if necessary.

The attribute will appear as follows:

```json
initial-value="{
    "instances": [
        {
            "color": "#2ca02c",
            "label": "Cat"
        },
        {
            "color": "#1f77b4",
            "label": "Cat"
        },
        {
            "color": "#d62728",
            "label": "Dog"
        }
    ],
    "src": {{ "S3 file URL for image" | grant_read_access }}
}"
```

Element Hierarchy

This element has the following parent and child elements.
- **Parent elements**: crowd-form (p. 470)
- **Child elements**: full-instructions (p. 483), short-instructions (p. 483)

**Regions**

The following regions are supported by this element.

- **full-instructions**
  General instructions about how to do image segmentation.

- **short-instructions**
  Important task-specific instructions that are displayed in a prominent place.

**Output**

The following output is supported by this element.

- **labeledImage**
  A JSON Object containing a Base64 encoded PNG of the labels.

- **instances**
  A JSON Array containing objects with the instance labels and colors.
  - **color** – The hexadecimal value of the label's RGB color in the labeledImage PNG.
  - **label** – The label given to overlay(s) using that color. This value may repeat, because the different instances of the label are identified by their unique color.

- **inputImageProperties**
  A JSON object that specifies the dimensions of the image that is being annotated by the worker. This object contains the following properties.
  - **height** – The height, in pixels, of the image.
  - **width** – The width, in pixels, of the image.

**Example: Sample Element Outputs**

The following is an example of output from this element.

```
[
  {
    "annotatedResult": {
      "inputImageProperties": {
        "height": 533,
        "width": 800
      },
      "instances": [
        {
          "color": "#1f77b4",
          "label": "<Label 1>"
        },
        {
          "color": "#2ca02c",
```
"label": "<Label 1>":
},
{
  "color": "#ff7f0e",
  "label": "<Label 3>":
},
"labeledImage": {
  "pngImageData": "<Base-64 Encoded Data>"
}
}
"
]

See Also

For more information, see the following.

- Use Amazon SageMaker Ground Truth to Label Data (p. 161)
- Crowd HTML Elements Reference (p. 448)

crowd-instructions

An element that displays instructions on three tabbed pages, Summary, Detailed Instructions, and Examples, when the worker clicks on a link or button.

The following is an example of a Liquid template that used the <crowd-instructions> element. Copy the following code and save it in a file with the extension .html. Open the file in any browser to preview and interact with this template.

```html
<script src="https://assets.crowd.aws/crowd-html-elements.js"></script>

<crowd-form>
<crowd-instructions link-text="View instructions" link-type="button">
  <short-summary>
    <p>Given an image, write three words or short phrases that summarize its contents.</p>
  </short-summary>

  <detailed-instructions>
    <p>Imagine that you are describing an image to a friend or tagging it for a news website. Provide three specific words or short phrases that describe it.</p>
  </detailed-instructions>

  <positive-example>
    <p><img src="https://s3.amazonaws.com/cv-demo-images/highway.jpg"></p>
    <ul>
      <li>Highway</li>
      <li>Cars</li>
      <li>Gas station</li>
    </ul>
  </positive-example>

  <negative-example>
    <p><img src="https://s3.amazonaws.com/cv-demo-images/highway.jpg"></p>
    <ol>
      <li>Trees</li>
      <li>Outside</li>
      <li>Daytime</li>
    </ol>
  </negative-example>
</crowd-instructions>
```

484
Instructions: Given an image, write three words or short phrases that summarize its contents.
If someone were to see these three words or phrases, they should understand the subject and context of the image, as well as any important actions.

View the instructions for detailed instructions and examples.

Attributes
The following attributes are supported by this element.

- **link-text**
  The text to display for opening the instructions. The default is Click for instructions.

- **link-type**
  A string that specifies the type of trigger for the instructions. The possible values are "link" (default) and "button".

Element Hierarchy
This element has the following parent and child elements.

- Parent elements: crowd-form (p. 470)
- Child elements: none

Regions
The following regions are supported by this element.

- **detailed-instructions**
  Content that provides specific instructions for a task. This appears on the page of the "Detailed Instructions" tab.

- **negative-example**
  Content that provides examples of inadequate task completion. This appears on the page of the "Examples" tab. More than one example may be provided within this element.

- **positive-example**
  Content that provides examples of proper task completion. This appears on the page of the "Examples" tab.

- **short-summary**
  A brief statement that summarizes the task to be completed. This appears on the page of the "Summary" tab. More than one example may be provided within this element.
crowd-keypoint

Generates a tool to select and annotate key points on an image.

The following is an example of an Liquid template that uses the `<crowd-keypoint>` element. Copy the following code and save it in a file with the extension `.html`. Open the file in any browser to preview and interact with this template.

```html
<script src="https://assets.crowd.aws/crowd-html-elements.js"></script>

<crowd-form>
  <div id="errorBox"></div>
  <crowd-keypoint
    src="{{ task.input.taskObject | grant_read_access }}"
    labels="['Item A', 'Item B', 'Item C']"
    header="Please locate the centers of each item."
    name="annotatedResult">
    <short-instructions>
      Describe your task briefly here and give examples
    </short-instructions>
    <full-instructions>
      Give additional instructions and good/bad examples here
    </full-instructions>
  </crowd-keypoint>
</crowd-form>

<script>
  var num_obj = 1;

  document.querySelector('crowd-form').onsubmit = function(e) {
    const keypoints = document.querySelector('crowd-keypoint').value.keypoints ||
                     document.querySelector('crowd-keypoint')._submittableValue.keypoints;
    const labels = keypoints.map(function(p) {
      return p.label;
    });

    // 1. Make sure total number of keypoints is correct.
    var original_num_labels = document.getElementsByTagName("crowd-keypoint")
                             [0].getAttribute("labels");
    original_num_labels = original_num_labels.substring(2, original_num_labels.length - 2).split("", ",");
    var goalNumKeypoints = num_obj*original_num_labels.length;
    if (keypoints.length != goalNumKeypoints) {
      e.preventDefault();
      errorBox.innerHTML = '<crowd-alert type="error" dismissible>You must add all keypoint annotations and use each label only once.</crowd-alert>';
      errorBox.scrollIntoView();
      return;
    }

    // 2. Make sure all labels are unique.
    labelCounts = {};
    for (var i = 0; i < labels.length; i++) {
...
if (!labelCounts[labels[i]]) {
    labelCounts[labels[i]] = 0;
} else {
    labelCounts[labels[i]]++;
}

const goalNumSingleLabel = num_obj;

const numLabels = Object.keys(labelCounts).length;

Object.entries(labelCounts).forEach(entry => {
    if (entry[1] != goalNumSingleLabel) {
        e.preventDefault();
        errorBox.innerHTML = '<crowd-alert type="error" dismissible>You must use each label only once.</crowd-alert>,'
        errorBox.scrollIntoView();
    }
});
</script>

**Attributes**

The following attributes are supported by this element.

**header**

The text to display above the image. This is typically a question or simple instruction for the worker.

**initial-value**

An array, in JSON format, of keypoints to be applied to the image on start. For example:

```json
initial-value="[
    {
        'label': 'Left Eye',
        'x': 1022,
        'y': 429
    },
    {
        'label': 'Beak',
        'x': 941,
        'y': 403
    }
]
```

**Note**

Please note that label values used in this attribute must have a matching value in the `labels` attribute or the point will not be rendered.

**labels**

An array, in JSON format, of strings to be used as keypoint annotation labels.

**name**

A string used to identify the answer submitted by the worker. This value will match a key in the JSON object that specifies the answer.

**src**

The source URI of the image to be annotated.
Element Hierarchy

This element has the following parent and child elements.

- **Parent elements**: crowd-form (p. 470)
- **Child elements**: full-instructions (p. 488), short-instructions (p. 488)

Regions

The following regions are required by this element.

full-instructions

General instructions about how to annotate the image.

short-instructions

Important task-specific instructions that are displayed in a prominent place.

Output

The following output is supported by this element.

**inputImageProperties**

A JSON object that specifies the dimensions of the image that is being annotated by the worker. This object contains the following properties.

- **height** – The height, in pixels, of the image.
- **width** – The width, in pixels, of the image.

**keypoints**

An array of JSON objects containing the coordinates and label of a keypoint. Each object contains the following properties.

- **label** – The assigned label for the keypoint.
- **x** – The X coordinate, in pixels, of the keypoint on the image.
- **y** – The Y coordinate, in pixels, of the keypoint on the image.

Note

X and Y coordinates are based on 0,0 being the top left corner of the image.

**Example : Sample Element Outputs**

The following is a sample output from using this element.

```
[
    {
        "crowdKeypoint": {
            "inputImageProperties": {
                "height": 1314,
                "width": 962
            },
            "keypoints": [
                {
```
You may have many labels available, but only the ones that are used appear in the output.

See Also

For more information, see the following.

- Use Amazon SageMaker Ground Truth to Label Data (p. 161)
- Crowd HTML Elements Reference (p. 448)

**crowd-line**

A widget for drawing lines on an image. Each line is associated with a label, and output data will report the starting and ending points of each line.

The following is an example of a Liquid template that uses the `<crowd-line>` element. Copy the following code and save it in a file with the extension `.html`. Open the file in any browser to preview and interact with this template. For more examples, see this GitHub repository.

```html
<script src="https://assets.crowd.aws/crowd-html-elements.js"></script>

<form>
  <crowd-line
    name="crowdLine"
    src="{{ task.input.taskObject | grant_read_access }}"
    header="Add header here to describe the task"
    labels="['car', 'pedestrian', 'street car']"
  />
</form>
```
Attributes

The following attributes are supported by this element.

header

Optional. The text to display above the image. This is typically a question or simple instruction for the worker.

initial-value

Optional. An array of JSON objects, each of which sets a line when the component is loaded. Each JSON object in the array contains the following properties:

- **label** – The text assigned to the line as part of the labeling task. This text must match one of the labels defined in the *labels* attribute of the `<crowd-line>` element.
- **vertices** – the x and y pixel coordinates of the start point and end point of the line, relative to the top-left corner of the image.

```json
initial-value="{
  lines: [
    {   
      label: 'sideline', // label of this line annotation
      vertices:[        // an array of vertices which decide the position of the line
        { x: 84,       
          y: 110
        },
        { x: 60,       
          y: 100
        }
      ]
    },
    {   
      label: 'yardline',
      vertices:[
        { x: 651,     
          y: 498
        }
      ]
    }
  
```
Lines set via the `initial-value` property can be adjusted. Whether or not a worker answer was adjusted is tracked via an `initialValueModified` boolean in the worker answer output.

**labels**

Required. A JSON formatted array of strings, each of which is a label that a worker can assign to the line.

**Limit:** 10 labels

**label-colors**

Optional. An array of strings. Each string is a hexadecimal (hex) code for a label.

**name**

Required. The name of this widget. It's used as a key for the widget's input in the form output.

**src**

Required. The URL of the image on which to draw lines.

**Regions**

The following regions are required by this element.

**full-instructions**

General instructions about how to draw lines.

**short-instructions**

Important task-specific instructions that are displayed in a prominent place.

**Element Hierarchy**

This element has the following parent and child elements.

- **Parent elements:** crowd-form (p. 470)
- **Child elements:** short-instructions (p. 491), full-instructions (p. 491)

**Output**

**inputImageProperties**

A JSON object that specifies the dimensions of the image that is being annotated by the worker. This object contains the following properties.

- **height** – The height, in pixels, of the image.
- **width** – The width, in pixels, of the image.
lines

A JSON Array containing objects with the line labels and vertices.

- **label** – The label given to a line.
- **vertices** – the x and y pixel coordinates of the start point and end point of the line, relative to the top-left corner of the image.

**Example: Sample Element Outputs**

The following is an example of output from this element.

```json
{
    "crowdLine": { //This is the name you set for the crowd-line
        "inputImageProperties": {
            "height": 1254,
            "width": 2048
        },
        "lines": [
            {
                "label": "yardline",
                "vertices": [
                    {
                        "x": 58,
                        "y": 295
                    },
                    {
                        "x": 1342,
                        "y": 398
                    }
                ]
            },
            {
                "label": "sideline",
                "vertices": [
                    {
                        "x": 472,
                        "y": 910
                    },
                    {
                        "x": 1480,
                        "y": 600
                    }
                ]
            }
        ]
    }
}
```

**See Also**

For more information, see the following.

- [Use Amazon SageMaker Ground Truth to Label Data](p. 161)
- [Crowd HTML Elements Reference](p. 448)

**crowd-modal**

A small window that pops up on the display when it is opened.
The following is an example of the syntax that you can use with the `<crowd-modal>` element. Copy the following code and save it in a file with the extension `.html`. Open the file in any browser to preview and interact with this template.

```html
<script src="https://assets.crowd.aws/crowd-html-elements.js"></script>

<crowd-modal
    link-text = "See Examples"
    link-type = "button">
    Example Modal Text</crowd-modal>
```

**Attributes**

The following attributes are supported by this element.

- **link-text**
  The text to display for opening the modal. The default is "Click to open modal".

- **link-type**
  A string that specifies the type of trigger for the modal. The possible values are "link" (default) and "button".

**Element Hierarchy**

This element has the following parent and child elements.

- **Parent elements**: crowd-form (p. 470)
- **Child elements**: none

**See Also**

For more information, see the following.

- Use Amazon SageMaker Ground Truth to Label Data (p. 161)
- Crowd HTML Elements Reference (p. 448)

**crowd-polygon**

A widget for drawing polygons on an image and assigning a label to the portion of the image that is enclosed in each polygon.

The following is an example of a Liquid template that uses the `<crowd-polygon>` element. Copy the following code and save it in a file with the extension `.html`. Open the file in any browser to preview and interact with this template.

```html
<script src="https://assets.crowd.aws/crowd-html-elements.js"></script>

<crowd-form>
    <crowd-polygon
        name="annotatedResult"
        src="{{ task.input.taskObject | grant_read_access }}"
        header="Draw a polygon around each of the requested target(s) of interest"
        labels="['Cat', 'Dog', 'Bird']">
```

493
<full-instructions header="Polygon instructions">
  <ul>
    <li>Make the polygon tight around the object</li>
    <li>You need to select a label before starting a polygon</li>
    <li>You will need to select a label again after completing a polygon</li>
    <li>To select a polygon, you can click on its borders</li>
    <li>You can start drawing a polygon from inside another polygon</li>
    <li>You can undo and redo while you’re drawing a polygon to go back and forth between points you’ve placed</li>
    <li>You are prevented from drawing lines that overlap other lines from the same polygon</li>
  </ul>
</full-instructions>

<short-instructions>
  <p>Draw a polygon around each of the requested target(s) of interest</p>
  <p>Make the polygon tight around the object</p>
</short-instructions>

Attributes

The following attributes are supported by this element.

header

The text to display above the image. This is typically a question or simple instruction for the worker.

labels

A JSON formatted array of strings, each of which is a label that a worker can assign to the image portion enclosed by a polygon.

name

The name of this widget. It's used as a key for the widget's input in the form output.

src

The URL of the image on which to draw polygons.

initial-value

An array of JSON objects, each of which defines a polygon to be drawn when the component is loaded. Each JSON object in the array contains the following properties.

- label – The text assigned to the polygon as part of the labeling task. This text must match one of the labels defined in the labels attribute of the <crowd-polygon> element.
- vertices – An array of JSON objects. Each object contains an x and y coordinate value for a point in the polygon.

Example

An initial-value attribute might look something like this.

```plaintext
initial-value =
  
  ['
    
    "label": "dog",
```
"vertices":
[
  {
    "x": 570,
    "y": 239
  },
  ...
  {
    "x": 759,
    "y": 281
  }
],
}

Because this will be within an HTML element, the JSON array must be enclosed in single or double quotes. The example above uses single quotes to encapsulate the JSON and double quotes within the JSON itself. If you must mix single and double quotes inside your JSON, replace them with their HTML entity codes (< for double quote, & for single) to safely escape them.

**Element Hierarchy**

This element has the following parent and child elements.

- **Parent elements**: crowd-form (p. 470)
- **Child elements**: full-instructions (p. 495), short-instructions (p. 495)

**Regions**

The following regions are required.

- **full-instructions**
  General instructions about how to draw polygons.

- **short-instructions**
  Important task-specific instructions that are displayed in a prominent place.

**Output**

The following output is supported by this element.

- **polygons**
  An array of JSON objects, each of which describes a polygon that has been created by the worker. Each JSON object in the array contains the following properties.

  - **label** – The text assigned to the polygon as part of the labeling task.
  - **vertices** – An array of JSON objects. Each object contains an x and y coordinate value for a point in the polygon. The top left corner of the image is 0,0.

**inputImageProperties**

A JSON object that specifies the dimensions of the image that is being annotated by the worker. This object contains the following properties.

- **height** – The height, in pixels, of the image.
- **width** – The width, in pixels, of the image.

**Example: Sample Element Outputs**

The following are samples of outputs from common use scenarios for this element.

**Single Label, Single Polygon**

```json
{
   "annotatedResult": {
      "inputImageProperties": {
         "height": 853,
         "width": 1280
      },
      "polygons": [
         {
            "label": "dog",
            "vertices": [
               {
                  "x": 570,
                  "y": 239
               },
               {
                  "x": 603,
                  "y": 513
               },
               {
                  "x": 823,
                  "y": 645
               },
               {
                  "x": 901,
                  "y": 417
               },
               {
                  "x": 759,
                  "y": 281
               }
            ]
         }
      ]
   }
}
```

**Single Label, Multiple Polygons**

```json
[
   {
      "annotatedResult": {
         "inputImageProperties": {
            "height": 853,
            "width": 1280
         },
         "polygons": [
            {
               "label": "dog",
               "vertices": [
                  {
                     "x": 570,
                     "y": 239
                  },
                  {
                     "x": 603,
                     "y": 513
                  },
                  {
                     "x": 823,
                     "y": 645
                  },
                  {
                     "x": 901,
                     "y": 417
                  },
                  {
                     "x": 759,
                     "y": 281
                  }
               ]
            }
         ]
      }
   }
]```
Multiple Labels, Multiple Polygons

[]
```json
{
  "annotatedResult": {
    "inputImageProperties": {
      "height": 853,
      "width": 1280
    },
    "polygons": [
      {
        "label": "dog",
        "vertices": [
          {
            "x": 570,
            "y": 239
          },
          {
            "x": 603,
            "y": 513
          },
          {
            "x": 823,
            "y": 645
          },
          {
            "x": 901,
            "y": 417
          },
          {
            "x": 759,
            "y": 281
          }
        ]
      },
      {
        "label": "cat",
        "vertices": [
          {
            "x": 870,
            "y": 278
          },
          {
            "x": 908,
            "y": 446
          },
          {
            "x": 1009,
            "y": 602
          },
          {
            "x": 1116,
            "y": 519
          },
          {
            "x": 1174,
            "y": 498
          },
          {
            "x": 1227,
            "y": 479
          },
          {
            "x": 1179,
            "y": 405
          },
          {
            "x": 1179,
            "y": 337
          }
        ]
      }
    ]
  }
}
```
You could have many labels available, but only the ones that are used appear in the output.

See Also

For more information, see the following.

• Use Amazon SageMaker Ground Truth to Label Data (p. 161)
• Crowd HTML Elements Reference (p. 448)

crowd-polyline

A widget for drawing polylines or lines on an image. Each polyline is associated with a label and can include two or more vertices. A polyline can intersect itself and its starting and ending points can be placed anywhere on the image.

The following is an example of a Liquid template that uses the <crowd-polyline> element. Copy the following code and save it in a file with the extension .html. Open the file in any browser to preview and interact with this template. For more examples, see this GitHub repository.

```html
<script src="https://assets.crowd.aws/crowd-html-elements.js"></script>

<crowd-form>

<crowd-polyline
    name="crowdPolyline"
    src="{{ task.input.taskObject | grant_read_access }}"
    header="Add header here to describe the task"
    labels="['car', 'pedestrian', 'street car']"
>

    <full-instructions>
        <p>Read the task carefully and inspect the image.</p>
        <p>Choose the appropriate label that best suits the image.</p>
        <p>Draw a polyline around the boundaries of all objects that the label applies to.</p>
        <p>Use the <b>Enter</b> key to complete a polyline.</p>
        <p>Make sure that the polyline fits tightly around the boundary of the object.</p>
    </full-instructions>

    <short-instructions>
        <p>Read the task carefully and inspect the image.</p>
        <p>Review the tool guide to learn how to use the polyline tool.</p>
        <p>Choose the appropriate label that best suits the image.</p>
        <p>To draw a polyline, select a label that applies to an object of interest and add a single point to the photo by clicking on that point. Continue to draw the polyline around the object by adding additional points around the object boundary.</p>
        <p>After you place the final point on the polyline, press <b>Enter</b> on your keyboard to complete the polyline.</p>
    </short-instructions>

</crowd-polyline>
</crowd-form>
```
Attributes

The following attributes are supported by this element.

header

Optional. The text to display above the image. This is typically a question or simple instruction for the worker.

initial-value

Optional. An array of JSON objects, each of which sets a polyline when the component is loaded. Each JSON object in the array contains the following properties:

- **label** – The text assigned to the polyline as part of the labeling task. This text must match one of the labels defined in the `labels` attribute of the `<crowd-polyline>` element.
- **vertices** – the x and y pixel coordinates of the vertices of a polyline, relative to the top-left corner of the image.

```json
initial-value= "{
  polylines: [
    {
      label: 'sideline', // label of this line annotation
      vertices:[        // an array of vertices which decide the position of the line
        {x: 84,        // an array of vertices which decide the position of the line
        y: 110},
        {x: 60,        // an array of vertices which decide the position of the line
        y: 100}
      ]
    },
    {
      label: 'yardline',
      vertices:[
        {x: 651,      // an array of vertices which decide the position of the line
        y: 498},
        {x: 862,      // an array of vertices which decide the position of the line
        y: 869},
        {x: 1000,     // an array of vertices which decide the position of the line
        y: 869}
      ]
    }
  ]
}"
```

Polylines set via the `initial-value` property can be adjusted. Whether or not a worker answer was adjusted is tracked via an `initialValueModified` boolean in the worker answer output.

labels

Required. A JSON formatted array of strings, each of which is a label that a worker can assign to the line.

**Limit:** 10 labels
label-colors
Optional. An array of strings. Each string is a hexadecimal (hex) code for a label.

name
Required. The name of this widget. It's used as a key for the widget's input in the form output.

src
Required. The URL of the image on which to draw polylines.

Regions
The following regions are required by this element.

full-instructions
General instructions about how to draw polylines.

short-instructions
Important task-specific instructions that are displayed in a prominent place.

Element Hierarchy
This element has the following parent and child elements.

- **Parent elements**: crowd-form (p. 470)
- **Child elements**: short-instructions (p. 501), full-instructions (p. 501)

Output

inputImageProperties
A JSON object that specifies the dimensions of the image that is being annotated by the worker. This object contains the following properties.

- **height** – The height, in pixels, of the image.
- **width** – The width, in pixels, of the image.

polylines
A JSON Array containing objects with polylines' labels and vertices.

- **label** – The label given to a line.
- **vertices** – the x and y pixel coordinates of the vertices of a polyline, relative to the top-left corner of the image.

Example: Sample Element Outputs
The following is an example of output from this element.

```json
{
    "crowdPolyline": { //This is the name you set for the crowd-polyline
        "inputImageProperties": { 
            "height": 1254,
            "width": 2048
    ```
},
  "polylines": [
    {
      "label": "sideline",
      "vertices": [
        {
          "x": 651,
          "y": 498
        },
        {
          "x": 862,
          "y": 869
        },
        {
          "x": 1449,
          "y": 611
        }
      ]
    },
    {
      "label": "yardline",
      "vertices": [
        {
          "x": 1148,
          "y": 322
        },
        {
          "x": 1705,
          "y": 474
        },
        {
          "x": 1755,
          "y": 474
        }
      ]
    }
  ]
}]

See Also

For more information, see the following.

- Use Amazon SageMaker Ground Truth to Label Data (p. 161)
- Crowd HTML Elements Reference (p. 448)

crowd-radio-button

A button that can be either checked or unchecked. When radio buttons are inside a radio group, exactly one radio button in the group can be checked at any time. The following is an example of how to configure a crowd-radio-button element inside of a crowd-radio-group element.

The following is an example of the syntax that you can use with the <crowd-radio-button> element. Copy the following code and save it in a file with the extension .html. Open the file in any browser to preview and interact with this template.

```html
<script src="https://assets.crowd.aws/crowd-html-elements.js"></script>
<crowd-form>
<crowd-radio-group>
```
The previous example can be seen in a custom worker task template in this GitHub example: entity recognition labeling job custom template.

Crowd HTML Element radio buttons do not support the HTML tag, `required`. To make a radio button selection required, use `<input type="radio">` elements to create radio buttons and add the `required` tag. The name attribute for all `<input>` elements that belong to the same group of radio buttons must be the same. For example, the following template requires the user to select a radio button in the `animal-type` group before submitting.

```html
<script src="https://assets.crowd.aws/crowd-html-elements.js"></script>
<crowd-form>
    <p> Select an animal type: </p>
    <img src="https://images.unsplash.com/photo-1537151608828-ea2b11777ee8?ixlib=rb-1.2.1&ixid=eyJhcHBfaWQiOjEyMDd9&auto=format&fit=crop&w=1539&q=80" style="height: 500; width: 400;"/>
    <br><br>
    <div>
        <input type="radio" id="cat" name="animal-type" value="cat" required>
        <label for="cat">Cat</label>
    </div>
    <div>
        <input type="radio" id="dog" name="animal-type" value="dog">
        <label for="dog">Dog</label>
    </div>
    <div>
        <input type="radio" id="unknown" name="animal-type" value="unknown">
        <label for="unknown">Unknown</label>
    </div>
</crowd-form>
```

**Attributes**

The following attributes are supported by this element.

- **checked**
  
  A Boolean switch that, if present, displays the radio button as checked.

- **disabled**
  
  A Boolean switch that, if present, displays the button as disabled and prevents it from being checked.

- **name**
  
  A string that is used to identify the answer submitted by the worker. This value will match a key in the JSON object that specifies the answer.

**Note**

If you use the buttons outside of a `crowd-radio-group` element, but with the same name string and different value strings, the `name` object in the output will contain a Boolean
value for each value string. To ensure that only one button in a group is selected, make them children of a crowd-radio-group (p. 504) element and use different name values.

value

A property name for the element's boolean value. If not specified, it uses "on" as the default, e.g. 

```{ "<name>": { "<value>": <true or false> } }```

Element Hierarchy

This element has the following parent and child elements.

- **Parent elements**: crowd-radio-group (p. 504)
- **Child elements**: none

Output

Outputs an object with the following pattern:

```{ "<name>": { "<value>": <true or false> } }```

If you use the buttons outside of a crowd-radio-group (p. 504) element, but with the same name string and different value strings, the name object will contain a Boolean value for each value string. To ensure that only one in a group of buttons is selected, make them children of a crowd-radio-group (p. 504) element and use different name values.

Example Sample output of this element

```
[ 
  { 
    "btn1": { 
      "yes": true 
    }, 
    "btn2": { 
      "no": false 
    } 
  } 
]
```

See Also

For more information, see the following.

- Use Amazon SageMaker Ground Truth to Label Data (p. 161)
- Crowd HTML Elements Reference (p. 448)

crowd-radio-group

A group of radio buttons. Only one radio button within the group can be selected. Choosing one radio button clears any previously chosen radio button within the same group. For an example of a custom UI template that uses the crowd-radio-group element, see this entity recognition labeling job custom template.

The following is an example of the syntax that you can use with the <crowd-radio-group> element. Copy the following code and save it in a file with the extension .html. Open the file in any browser to preview and interact with this template.

```<script src="https://assets.crowd.aws/crowd-html-elements.js"></script>```
<style>
body {
  padding-left: 20px;
  margin-bottom: 20px;
}
#outer-container {
  display: flex;
  justify-content: space-around;
  max-width: 900px;
  margin-left: 100px;
}
.left-container {
  margin-right: auto;
  padding-right: 50px;
}
.right-container {
  margin-left: auto;
  padding-left: 50px;
}
#vertical-separator {
  border: solid 1px #d5dbdb;
}</style>

crowd-form

  <div>
    <h1>Instructions</h1>
    Lorem ipsum...
  </div>

  <div>
    <h2>Background</h2>
    Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.
  </div>

  <div id="outer-container">
    <span class="left-container">
      <h2>Option 1</h2>
      <p>Nulla facilisi morbi tempus iaculis urna. Orci dapibus ultrices in iaculis nunc sed augue lacus.</p>
    </span>
    <span id="vertical-separator"></span>
    <span class="right-container">
      <h2>Option 2</h2>
      <p>Ultrices vitae auctor eu augue ut. Pellentesque massa placerat dui ultricies lacus sed turpis tincidunt id.</p>
    </span>
  </div>

  <div>
    <h2>Question</h2>
    Which do you agree with?
  </div>

  <crowd-radio-group>
    <crowd-radio-button name="option1" value="Option 1">Option 1</crowd-radio-button>
    <crowd-radio-button name="option2" value="Option 2">Option 2</crowd-radio-button>
  </crowd-radio-group>

  <p>Why did you choose this answer?</p>
  <crowd-text-area name="explanation" placeholder="Explain how you reached your conclusion..."></crowd-text-area>

</crowd-form>

Attributes

No special attributes are supported by this element.
Element Hierarchy

This element has the following parent and child elements.

- **Parent elements**: crowd-form (p. 470)
- **Child elements**: crowd-radio-button (p. 502)

Output

Outputs an array of objects representing the crowd-radio-button (p. 502) elements within it.

**Example Sample of Element Output**

```json
[
  
  "btn1": {
    "yes": true
  },
  
  "btn2": {
    "no": false
  }
]
```

See Also

For more information, see the following.

- Use Amazon SageMaker Ground Truth to Label Data (p. 161)
- Crowd HTML Elements Reference (p. 448)

**crowd-semantic-segmentation**

A widget for segmenting an image and assigning a label to each image segment.

The following is an example of a Liquid template that uses the `<crowd-semantic-segmentation>` element. Copy the following code and save it in a file with the extension `.html`. Open the file in any browser to preview and interact with this template.

```html
<template src="https://assets.crowd.aws/crowd-html-elements.js"></template>
<template>
  <crowd-form
    name="annotatedResult"
    src="{{ task.input.taskObject | grant_read_access }}"
    header="Please label each of the requested objects in this image"
    labels="["Cat", "Dog", "Bird"]"
  >
    <full-instructions header="Segmentation Instructions">
      <ol>
        <li><strong>Read</strong> the task carefully and inspect the image.</li>
        <li><strong>Read</strong> the options and review the examples provided to understand more about the labels.</li>
        <li><strong>Choose</strong> the appropriate label that best suits the image.</li>
      </ol>
    </full-instructions>
    <short-instructions>
    </short-instructions>
  </crowd-form>
</template>
```
Attributes

The following attributes are supported by this element.

**header**

The text to display above the image. This is typically a question or simple instruction for the worker.

**initial-value**

A JSON object containing the color mappings of a prior semantic segmentation job and a link to the overlay image output by the prior job. Include this when you want a human worker to verify the results of a prior labeling job and adjust it if necessary.

The attribute would appear as follows:

```json
initial-value='{
    "header": "Use the tools to label the requested items in the image",
    "initial-value": {
        "labelMappings": {
            "Bird": { "color": "#ff7f0e" },
            "Cat": { "color": "#2ca02c" },
            "Cow": { "color": "#d62728" },
            "Dog": { "color": "#1f77b4" }
        },
        "src": "S3 file URL for image" | grant_read_access
    }
}
```

When using Ground Truth **built in task types** with **annotation consolidation** (where more than one worker labels a single image), label mappings are included in individual worker output records, however the overall result is represented as the **internal-color-map** in the consolidated results.

You can convert the **internal-color-map** to **label-mappings** in a custom template using the Liquid templating language:

```liquid
initial-value="{
    'header': "Use the tools to label the requested items in the image",
    'initial-value': {
        'labelMappings': {
            '% for box in task.input.manifestLine.label-attribute-name-from-prior-job-
            metadata.internal-color-map %
            '% if box[1]["class-name"] != 'BACKGROUND' %
            '{% box[1]["class-name"] | to_json %}
            '% endif %
            '
        },
        'src': '{{ task.input.manifestLine.label-attribute-name-from-prior-job-
        grant_read_access %}'
    }
}
```

labels
A JSON formatted array of strings, each of which is a label that a worker can assign to a segment of the image.

name
The name of this widget. It is used as a key for the widget’s input in the form output.

src
The URL of the image that is to be segmented.

Element Hierarchy
This element has the following parent and child elements.
- Parent elements: crowd-form (p. 470)
- Child elements: full-instructions (p. 508), short-instructions (p. 508)

Regions
The following regions are supported by this element.

full-instructions
General instructions about how to do image segmentation.

short-instructions
Important task-specific instructions that are displayed in a prominent place.

Output
The following output is supported by this element.

labeledImage
A JSON Object containing a Base64 encoded PNG of the labels.

labelMappings
A JSON Object containing objects with named with the segmentation labels.
- color – The hexadecimal value of the label’s RGB color in the labeledImage PNG.

initialValueModified
A boolean representing whether the initial values have been modified. This is only included when the output is from an adjustment task.

inputImageProperties
A JSON object that specifies the dimensions of the image that is being annotated by the worker. This object contains the following properties.
- height – The height, in pixels, of the image.
- width – The width, in pixels, of the image.
Example: Sample Element Outputs

The following is a sample of output from this element.

```json
[
   {
      "annotatedResult": {
         "inputImageProperties": {
            "height": 533,
            "width": 800
         },
         "labelMappings": {
            "<Label 2>": {
               "color": "#ff7f0e"
            },
            "<label 3>": {
               "color": "#2ca02c"
            },
            "<label 1>": {
               "color": "#1f77b4"
            }
         },
         "labeledImage": {
            "pngImageData": "<Base-64 Encoded Data>"
         }
      }
   }
]
```

See Also

For more information, see the following.

- Use Amazon SageMaker Ground Truth to Label Data (p. 161)
- Crowd HTML Elements Reference (p. 448)

crowd-slider

A bar with a sliding knob that allows a worker to select a value from a range of values by moving the knob. The slider makes it a great choice for settings that reflect intensity levels, such as volume, brightness, or color saturation.

The following is an example of a survey template that uses the `<crowd-slider>` element. Copy the following code and save it in a file with the extension `.html`. Open the file in any browser to preview and interact with this template.

```html
<script src="https://assets.crowd.aws/crowd-html-elements.js"></script>

<crowd-form>
  <crowd-instructions link-text="View instructions" link-type="button">
    <short-summary>
      <p>Provide a brief instruction here</p>
    </short-summary>

    <detailed-instructions>
      <h3>Provide more detailed instructions here</h3>
      <p>Include additional information</p>
    </detailed-instructions>
  </crowd-instructions>
  <positive-example>
```
Attributes

The following attributes are supported by this element.

**disabled**

A Boolean switch that, if present, displays the slider as disabled.

**editable**

A Boolean switch that, if present, displays an up/down button that can be chosen to select the value.

Selecting the value via the up/down button is an alternative to selecting the value by moving the knob on the slider. The knob on the slider will move synchronously with the up/down button choices.

**max**

A number that specifies the maximum value on the slider.

**min**

A number that specifies the minimum value on the slider.

**name**

A string that is used to identify the answer submitted by the worker. This value will match a key in the JSON object that specifies the answer.

**pin**

A Boolean switch that, if present, displays the current value above the knob as the knob is moved.
required

A Boolean switch that, if present, requires the worker to provide input.

secondary-progress

When used with a `crowd-slider-secondary-color` CSS attribute, the progress bar is colored to the point represented by the `secondary-progress`. For example, if this was representing the progress on a streaming video, the `value` would represent where the viewer was in the video timeline. The `secondary-progress` value would represent the point on the timeline to which the video had buffered.

step

A number that specifies the difference between selectable values on the slider.

value

A preset that becomes the default if the worker does not provide input.

Element Hierarchy

This element has the following parent and child elements.

- **Parent elements**: `crowd-form` (p. 470)
- **Child elements**: none

See Also

For more information, see the following.

- Use Amazon SageMaker Ground Truth to Label Data (p. 161)
- Crowd HTML Elements Reference (p. 448)

crowd-tab

A component styled to look like a tab with information below.

The following is an example template that uses the `<crowd-tab>` element. Copy the following code and save it in a file with the extension `.html`. Open the file in any browser to preview and interact with this template.

```html
<script src="https://assets.crowd.aws/crowd-html-elements.js"></script>

<crowd-form>
  <crowd-tabs>
    <crowd-tab header="Tab 1">
      <h2>Image</h2>
      <img src="https://images.unsplash.com/photo-1478382188900-5bb598fe27d3?ixlib=rb-1.2.1&ixid=eyJhcHBfaWQiOjEyMDd9&auto=format&fit=crop&w=1351&q=80" style="max-width: 40%"
    
    <h2>Text</h2>
    <p>
```
Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.

Sed risus ultricies tristique nulla aliquet enim tortor at auctor. Tempus egestas sed sed risus.

<crowd-tab header="Tab 2">
<h2>Description</h2>
<p>Sed risus ultricies tristique nulla aliquet enim tortor at auctor. Tempus egestas sed sed risus.</p>
</crowd-tab>

<crowd-tab header="Tab 3">
<short-instructions>
<p>Sed risus ultricies tristique nulla aliquet enim tortor at auctor. Tempus egestas sed sed risus.</p>
</short-instructions>
<full-instructions header="Classification Instructions">
<p>Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.</p>
<p>Tempus egestas sed sed risus.</p>
</full-instructions>
</crowd-tab>

Attributes

The following attributes are supported by this element.
header

The text appearing on the tab. This is usually some short descriptive name indicative of the information contained below the tab.

Element Hierarchy

This element has the following parent and child elements.

- **Parent elements**: crowd-tabs (p. 513)
- **Child elements**: none

See Also

For more information, see the following.

- Use Amazon SageMaker Ground Truth to Label Data (p. 161)
- Crowd HTML Elements Reference (p. 448)

crowd-tabs

A container for tabbed information.

The following is an example template that uses the `<crowd-tabs>` element. Copy the following code and save it in a file with the extension `.html`. Open the file in any browser to preview and interact with this template.

```html
<script src="https://assets.crowd.aws/crowd-html-elements.js"></script>

<crowd-form>
  <crowd-tabs>
    <crowd-tab header="Tab 1">
      <h2>Image</h2>
      <img src="https://images.unsplash.com/photo-1478382188900-5bb598fe27d3?ixlib=r1.2.1&ixid=eyJhcHBfaWQiOjEyMDd9&auto=format&fit=crop&w=1351&q=80" style="max-width: 40%"/>
      <h2>Text</h2>
      <p>Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.</p>
      <p>Sed risus ultricies tristique nulla aliquet enim tortor at auctor. Tempus egestas sed sed risus.</p>
    </crowd-tab>
    <crowd-tab header="Tab 2">
      <h2>Description</h2>
      <p>Sed risus ultricies tristique nulla aliquet enim tortor at auctor. Tempus egestas sed sed risus.</p>
    </crowd-tab>
  </crowd-tabs>
</crowd-form>
```
Attributes

This element has no attributes.

Element Hierarchy

This element has the following parent and child elements.

- **Parent elements**: crowd-form (p. 470)
- **Child elements**: crowd-tab (p. 511)

See Also

For more information, see the following.

- Use Amazon SageMaker Ground Truth to Label Data (p. 161)
- Crowd HTML Elements Reference (p. 448)
Amazon SageMaker Developer Guide
SageMaker Crowd HTML Elements

crowd-text-area
A ﬁeld for text input.
The following is an example of a Liquid template designed to transcribe audio clips that uses the
<crowd-text-area> element. Copy the following code and save it in a ﬁle with the extenion .html.
Open the ﬁle in any browser to preview and interact with this template.
<script src="https://assets.crowd.aws/crowd-html-elements.js"></script>
<crowd-form>
<audio controls>
<source src="{{ task.input.taskObject | grant_read_access }}" type="audio/mpeg">
Your browser does not support the audio element.
</audio>
<h3>Instructions</h3>
<p>Transcribe the audio</p>
<p>Ignore "umms", "hmms", "uhs" and other non-textual phrases</p>
<crowd-text-area name="transcription" rows="4"></crowd-text-area>
</crowd-form>

Attributes
The following attributes are supported by this element.

auto-focus
A Boolean switch that, if present, puts the cursor in this element on-load so that users can immediately
begin typing without having to click inside the element.

auto-validate
A Boolean switch that, if present, turns on input validation. The behavior of the validator can be
modiﬁed by the error-message and allowed-pattern attributes.

char-counter
A Boolean switch that, if present, puts a small text ﬁeld beneath the lower-right corner of the element,
displaying the number of characters inside the element.

disabled
A Boolean switch that, if present, displays the input area as disabled.

error-message
The text to be displayed below the input ﬁeld, on the left side, if validation fails.

label
A string that is displayed inside a text ﬁeld.
This text shrinks and rises up above a text ﬁeld when the worker starts typing in the ﬁeld or when the
value attribute is set.

max-length
An integer that speciﬁes the maximum number of characters allowed by the element. Characters typed
or pasted beyond the maximum are ignored.

515


max-rows
An integer that specifies the maximum number of rows of text that are allowed within a crowd-text-area. Normally the element expands to accommodate new rows. If this is set, after the number of rows exceeds it, content scrolls upward out of view and a scrollbar control appears.

name
A string used to represent the element's data in the output.

placeholder
A string presented to the user as placeholder text. It disappears after the user puts something in the input area.

rows
An integer that specifies the height of the element in rows of text.

value
A preset that becomes the default if the worker does not provide input. The preset appears in a text field.

Element Hierarchy
This element has the following parent and child elements.

- **Parent elements**: crowd-form (p. 470)
- **Child elements**: none

Output
This element outputs the name as a property name and the element's text contents as the value. Carriage returns in the text are represented as \

Example Sample output for this element

```
[  
  {  
    "textInput1": "This is the text; the text that\nmakes the crowd go wild."  
  }  
]
```

See Also
For more information, see the following.

- Use Amazon SageMaker Ground Truth to Label Data (p. 161)
- Crowd HTML Elements Reference (p. 448)

crowd-toast
A subtle notification that temporarily appears on the display. Only one crowd-toast is visible.

The following is an example of a Liquid template that uses the `<crowd-toast>` element. Copy the following code and save it in a file with the extension `.html`. Open the file in any browser to preview and interact with this template.
<script src="https://assets.crowd.aws/crowd-html-elements.js"></script>

<p>Find the official website for: <strong>{{ task.input.company }}</strong></p>
<p>Do not give Yelp pages, LinkedIn pages, etc.</p>
<p>Include the http:// prefix from the website</p>
<p><crowd-input name="website" placeholder="http://example.com"></crowd-input></p>
<p>This is a message that you want users to see when opening the template. This message will disappear in 10 seconds.</p>

<p>Attributes</p>

The following attributes are supported by this element.

- **duration**
  A number that specifies the duration, in milliseconds, that the notification appears on the screen.

- **text**
  The text to display in the notification.

<p>Element Hierarchy</p>

This element has the following parent and child elements.

- **Parent elements**: crowd-form (p. 470)
- **Child elements**: none

<p>See Also</p>

For more information, see the following.

- Use Amazon SageMaker Ground Truth to Label Data (p. 161)
- Crowd HTML Elements Reference (p. 448)

<p>crowd-toggle-button</p>

A button that acts as an ON/OFF switch, toggling a state. The following example shows different ways you can use to use the <crowd-toggle-button> HTML element. Copy the following code and save it in a file with the extension .html. Open the file in any browser to preview and interact with this template.

```html
<crowd-toggle-button name="toggleButtonWithoutValue"></crowd-toggle-button>
<crowd-toggle-button name="toggleButtonWithValue" value="someValue"></crowd-toggle-button>
<crowd-toggle-button name="toggleButtonDisabled" disabled></crowd-toggle-button>
```
Attributes
The following attributes are supported by this element.

**checked**
A Boolean switch that, if present, displays the button switched to the ON position.

**disabled**
A Boolean switch that, if present, displays the button as disabled and prevents toggling.

**invalid**
When in an off position, a button using this attribute, will display in an alert color. The standard is red, but may be changed in CSS. When toggled on, the button will display in the same color as other buttons in the on position.

**name**
A string that is used to identify the answer submitted by the worker. This value matches a key in the JSON object that specifies the answer.

**required**
A Boolean switch that, if present, requires the worker to provide input.

**value**
A value used in the output as the property name for the element's Boolean state. Defaults to "on" if not provided.

Element Hierarchy
This element has the following parent and child elements.

- **Parent elements**: crowd-form (p. 470)
- **Child elements**: none

Output
This element outputs the name as the name of an object, containing the value as a property name and the element's state as Boolean value for the property. If no value for the element is specified, the property name defaults to "on."

Example Sample output for this element

```
[
    {
        "theToggler": {
            "on": true
        }
    }
]```
See Also
For more information, see the following.
- Use Amazon SageMaker Ground Truth to Label Data (p. 161)
- Crowd HTML Elements Reference (p. 448)

Augmented AI Crowd HTML Elements
The following Crowd HTML Elements are only available for Amazon Augmented AI human workflow tasks.

Topics
- crowd-textract-document-analysis (p. 519)
- crowd-rekognition-detect-moderation-labels (p. 523)

crowd-textract-document-analysis
A widget to enable human review of a Amazon Textract document analysis result.

Attributes
The following attributes are supported by this element.

header
This is the text that is displayed as the header.

src
This is a link to the image to be analyzed by the worker.

initialValue
This sets initial values for attributes found in the worker UI.

The following is an example of an initialValue input:

```json
[
  {"blockType": "KEY_VALUE_SET",
   "confidence": 38.43309020996094,
   "geometry": {
     "boundingBox": {
       "width": 0.32613086700439453,
       "weight": 0.0942094624042511,
       "left": 0.4833833575248718,
       "top": 0.522798898358765
     },
     "polygon": [
       {"X": 0.123, "Y": 0.345}, ...
     ]
   },
   "id": "8c97b240-0969-4678-834a-646c95da9cf4",
   "relationships": [
     ...
   ]
  }
]```
"type": "CHILD",
"ids": [
  "7ee7b7da-ee1b-428d-a567-55a3e3affa56",
  "4d6da730-ba43-467c-a9a5-c6137ba0c472"
],
},
{
  "type": "VALUE",
  "ids": [
  "6ee7b7da-ee1b-428d-a567-55a3e3affa54"
  ],
  "entityTypes": [
  "KEY"
  ],
  "text": "Foo bar"
}]

blockTypes

This determines the kind of analysis the workers can do. Only KEY_VALUE_SET is currently supported.

keys

This specifies new keys and the associated text value the worker can add. The input values for keys can include the following elements:

- importantFormKey accepts strings, and is used to specify a single key.
- importantFormKeyAliases can be used to specify aliases that are acceptable alternatives to the keys supplied. Use this element to identify alternative spellings or presentations of your keys. This parameter accepts a list of one or more strings.

The following is an example of an input for keys.

[
  {
    importantFormKey: 'Address',
    importantFormKeyAliases: [
      'address',
      'Addr.',
      'Add.'
    ]
  },
  {
    importantFormKey: 'Last name',
    importantFormKeyAliases: ['Surname']
  }
]

no-key-edit

This prevents the workers from editing the keys of annotations passed through initialValue. This prevents workers from editing the keys that have been detected on your documents. This is required.

no-geometry-edit

This prevents workers from editing the polygons of annotations passed through initialValue. For example, this would prevent the worker from editing the bounding box around a given key. This is required.
Element Hierarchy

This element has the following parent and child elements.

- Parent elements – crowd-form
- Child elements – full-instructions (p. 521), short-instructions (p. 521)

Regions

The following regions are supported by this element. You can use custom HTML and CSS code within these regions to format your instructions to workers. For example, use the short-instructions section to provide good and bad examples of how to complete a task.

full-instructions

General instructions about how to work with the widget.

short-instructions

Important task-specific instructions that are displayed in a prominent place.

Example of a Worker Template Using the crowd Element

An example of a worker template using this crowd element would look like the following.

```html
<script src="https://assets.crowd.aws/crowd-html-elements.js"></script>
{% capture s3_arn %}http://s3.amazonaws.com/{{ task.input.aiServiceRequest.document.s3Object.bucket }}/{{ task.input.aiServiceRequest.document.s3Object.name }}{% endcapture %}

<crowd-form>
  <crowd-textract-analyze-document
    src="{{ s3_arn | grant_read_access }}"
    initial-value="{{ task.input.selectedAiServiceResponse.blocks }}"
    header="Review the key-value pairs listed on the right and correct them if they don't match the following document."
    no-key-edit
    no-geometry-edit
    keys="{{ task.input.humanLoopContext.importantFormKeys }}"
    block-types=['KEY_VALUE_SET']">
    <short-instructions header="Instructions">
      <style>
        .instructions {
          white-space: pre-wrap;
        }
        .instructionsImage {
          display: inline-block;
          max-width: 100%;
        }
      </style>
      <p class='instructions'>Click on a key-value block to highlight the corresponding key-value pair in the document.

      If it is a valid key-value pair, review the content for the value. If the content is incorrect, correct it.

      The text of the value is incorrect, correct it.</p>
    </short-instructions>
  </crowd-textract-analyze-document>
</crowd-form>
```

521
A wrong value is identified, correct it.
<img class='instructionsImage' src="https://assets.crowd.aws/images/a2i-console/correct-value.png" />

If it is not a valid key-value relationship, choose No.
<img class='instructionsImage' src="https://assets.crowd.aws/images/a2i-console/not-a-key-value-pair.png" />

If you can’t find the key in the document, choose Key not found.
<img class='instructionsImage' src="https://assets.crowd.aws/images/a2i-console/key-is-not-found.png" />

If the content of a field is empty, choose Value is blank.
<img class='instructionsImage' src="https://assets.crowd.aws/images/a2i-console/value-is-blank.png" />

<b>Examples</b>
Key and value are often displayed next or below to each other.

Key and value displayed in one line.
<img class='instructionsImage' src="https://assets.crowd.aws/images/a2i-console/sample-key-value-pair-1.png" />

Key and value displayed in two lines.
<img class='instructionsImage' src="https://assets.crowd.aws/images/a2i-console/sample-key-value-pair-2.png" />

If the content of the value has multiple lines, enter all the text without line break.
Include all value text even if it extends beyond the highlight box.
<img class='instructionsImage' src="https://assets.crowd.aws/images/a2i-console/multiple-lines.png" />

Output

The following is a sample of the output from this element. You can find a detailed explanation of this output in the Amazon Textract AnalyzeDocument API documentation.

```json
{
  "AWS/Textract/AnalyzeDocument/Forms/V1": {
    "blocks": [
      {
        "blockType": "KEY_VALUE_SET",
        "id": "8c97b240-0969-4678-834a-646c95da9cf4",
        "relationships": [
          {
            "type": "CHILD",
            "ids": ["7ee7b7da-ee1b-428d-a567-55a3e3affa56", "4d6da730-ba43-467c-a9a5-c6137ba0c472"]
          },
          {
            "type": "VALUE",
            "ids": ["6ee7b7da-ee1b-428d-a567-55a3e3affa54"]
          }
        ],
        "entityTypes": ["KEY"],
        "text": "Foo bar baz"
      }
    ]
  }
}
```
crowd-rekognition-detect-moderation-labels

A widget to enable human review of an Amazon Rekognition image moderation result.

Attributes

The following attributes are supported by this element.

header

This is the text that is displayed as the header.

categories

This supports categories as an array of strings or an array of objects where each object has a name field.

If the categories come in as objects, the following applies:

• The displayed categories are the value of the name field.
• The returned answer contains the full objects of any selected categories.

If the categories come in as strings, the following applies:

• The returned answer is an array of all the strings that were selected.

exclusion-category

By setting this attribute you create a button underneath the categories in the UI.

• When a user chooses the button, all categories are deselected and disabled.
• Choosing the button again re-enables the categories so that users can choose them.
• If you submit after choosing the button, it returns an empty array.

Element Hierarchy

This element has the following parent and child elements.

• Parent elements – crowd-form
• Child elements – full-instructions (p. 523), short-instructions (p. 524)

AWS Regions

The following AWS Regions are supported by this element. You can use custom HTML and CSS code within these Regions to format your instructions to workers. For example, use the short-instructions section to provide good and bad examples of how to complete a task.

crowd-rekognition-detect-moderation-labels

full-instructions

General instructions about how to work with the widget.
short-instructions

Important task-specific instructions that are displayed in a prominent place.

Example Worker Template with the crowd Element

An example of a worker template using the crowd element would look like the following.

```html
<script src="https://assets.crowd.aws/crowd-html-elements.js"></script>
{% capture s3_arn %}http://s3.amazonaws.com/{{ task.input.aiServiceRequest.image.s3Object.bucket}}/{{ task.input.aiServiceRequest.image.s3Object.name }}{% endcapture %}

<crowd-form>
  <crowd-rekognition-detect-moderation-labels
categories='[
    {% for label in task.input.selectedAiServiceResponse.moderationLabels %}
    {
      name: "{{ label.name }}",
      parentName: "{{ label.parentName }}",
    },
    {% endfor %}
  ]
  src="{{ s3_arn | grant_read_access }}"
  header="Review the image and choose all applicable categories."
>
  <short-instructions header="Instructions">
    <style>
      .instructions {
        white-space: pre-wrap;
      }
    </style>
    <p class='instructions'>Review the image and choose all applicable categories. If no categories apply, choose None.

    <b>Nudity</b>
    Visuals depicting nude male or female person or persons

    <b>Graphic Male Nudity</b>
    Visuals depicting full frontal male nudity, often close ups

    <b>Graphic Female Nudity</b>
    Visuals depicting full frontal female nudity, often close ups

    <b>Sexual Activity</b>
    Visuals depicting various types of explicit sexual activities and pornography

    <b>Illustrated Nudity or Sexual Activity</b>
    Visuals depicting animated or drawn sexual activity, nudity or pornography

    <b>Adult Toys</b>
    Visuals depicting adult toys, often in a marketing context

    <b>Female Swimwear or Underwear</b>
    Visuals depicting female person wearing only swimwear or underwear

    <b>Male Swimwear Or Underwear</b>
    Visuals depicting male person wearing only swimwear or underwear

    <b>Partial Nudity</b>
    Visuals depicting covered up nudity, for example using hands or pose

    <b>Revealing Clothes</b>
    Visuals depicting revealing clothes and poses, such as deep cut dresses
  </short-instructions>
</crowd-form>
```
<b>Graphic Violence or Gore</b>
Visuals depicting prominent blood or bloody injuries

<b>Physical Violence</b>
Visuals depicting violent physical assault, such as kicking or punching

<b>Weapon Violence</b>
Visuals depicting violence using weapons like firearms or blades, such as shooting

<b>Weapons</b>
Visuals depicting weapons like firearms and blades

<b>Self Injury</b>
Visuals depicting self-inflicted cutting on the body, typically in distinctive patterns using sharp objects

<b>Emaciated Bodies</b>
Visuals depicting extremely malnourished human bodies

<b>Corpses</b>
Visuals depicting human dead bodies

<b>Hanging</b>
Visuals depicting death by hanging

</short-instructions>

</full-instructions header="Instructions"></full-instructions>
</crowd-rekognition-detect-moderation-labels>
</crowd-form>

**Output**

The following is a sample of the output from this element. For details about this output, see Amazon Rekognition DetectModerationLabels API documentation.

```json
{
    "AWS/Rekognition/DetectModerationLabels/Image/V3": {
        "ModerationLabels": [
            { name: 'Gore', parentName: 'Violence' },
            { name: 'Corpses', parentName: 'Violence' }
        ]
    }
}
```
Prepare and Analyze Datasets

Import, prepare, transform, visualize and analyze data with Amazon SageMaker Data Wrangler. You can integrate Data Wrangler into your machine learning workflows to simplify and streamline data preprocessing and feature engineering using little to no coding. You can also add your own Python scripts and transformations to customize your data prep workflow.

Import data from Amazon S3, Amazon Redshift, Amazon Athena, and use Data Wrangler to create sophisticated machine learning data prep workflows with built-in and custom data transformations and analysis including feature target leakage and quick modeling.

After you have defined a data prep workflow, or data flow, you can integrate it with SageMaker Processing, SageMaker Pipelines, and SageMaker Feature Store, simplify the task of processing, sharing and storing ML training data. You can also export your data flow to a python script and create a custom ML data prep pipeline.

For more information, see Prepare ML Data with Amazon SageMaker Data Wrangler (p. 526).

To analyze your data and evaluate machine learning models on Amazon SageMaker, use Amazon SageMaker Processing. With Processing, you can use a simplified, managed experience on SageMaker to run your data processing workloads, such as feature engineering, data validation, model evaluation, and model interpretation. You can also use the Amazon SageMaker Processing APIs during the experimentation phase and after the code is deployed in production to evaluate performance.

Topics
• Prepare ML Data with Amazon SageMaker Data Wrangler (p. 526)
• Process Data and Evaluate Models (p. 580)

Prepare ML Data with Amazon SageMaker Data Wrangler

Amazon SageMaker Data Wrangler (Data Wrangler) is a feature of SageMaker Studio that provides an end-to-end solution to import, prepare, transform, featurize, and analyze data. You can integrate a Data Wrangler data flow into your machine learning (ML) workflows to simplify and streamline data preprocessing and feature engineering using little to no coding. You can also add your own Python scripts and transformations to customize a Data Wrangler data prep workflow.

The following are the core functionalities that Data Wrangler provides to help you analyze and prepare data for machine learning applications.

• **Import** – Connect to and import data from Amazon Simple Storage Service (Amazon S3), Amazon Athena (Athena), and Amazon Redshift.
• **Data Flow** – Create a data flow to define a series of ML data prep steps. You can use a flow to combine datasets from different data sources, identify the number and types of transformations you want to apply to datasets, and define a data prep workflow that can be easily integrated into an ML pipeline.
• **Transform** – Clean and transform your dataset using standard transforms like string, vector, and numeric data formatting tools. Featurize your data using transforms like text and date/time embedding and categorical encoding.
• **Analyze** – Analyze features in your dataset at any point in your flow. Data Wrangler includes built-in data visualization tools like scatter plots and histograms, as well as data analysis tools like target leakage analysis and quick modeling to understand feature correlation.

• **Export** – Data Wrangler offers export options to other SageMaker services, including Data Wrangler jobs, Feature Store, and pipelines, making it easy to integrate your data prep flow into your ML workflow. You can also export your Data Wrangler flow to Python code.

To start using Data Wrangler, see *Get Started with Data Wrangler* (p. 527).

**Topics**

- Get Started with Data Wrangler (p. 527)
- Import (p. 536)
- Create and Use a Data Wrangler Flow (p. 543)
- Transform Data (p. 545)
- Analyze and Visualize (p. 562)
- Export (p. 569)
- Security and Permissions (p. 573)
- Troubleshoot (p. 578)

**Get Started with Data Wrangler**

Amazon SageMaker Data Wrangler is a feature in SageMaker Studio. Use this section to learn how to access and get started using Data Wrangler. Do the following:

1. Complete each step in **Prerequisites** (p. 527).
2. Follow the procedure in **Access Data Wrangler** (p. 527) to start using Data Wrangler.

**Prerequisites**

To use Data Wrangler, you must do the following:

1. To use Data Wrangler, you need access to a **m5.4xlarge** Amazon Elastic Compute Cloud (Amazon EC2) instance. To learn how to view your quotas, and if necessary, request a quota increase, see *AWS service quotas*.
2. Configure the required permissions described in **Security and Permissions** (p. 573).

To use Data Wrangler, you need an active SageMaker Studio instance. To learn how to launch a new instance, see *Onboard to Amazon SageMaker Studio* (p. 34). When your Studio instance is **Ready**, use the instructions in **Access Data Wrangler** (p. 527).

**Access Data Wrangler**

The following procedure assumes you have completed the **Prerequisites** (p. 527).

**To access Data Wrangler in Studio:**

1. Next to the user you want to use to launch Studio, select **Open Studio**.
2. When Studio opens, select the + sign on the **New data flow** card under **ML tasks and components**. This creates a new folder in Studio with a .flow file inside, which contains your data flow. The .flow file automatically opens in Studio.
You can also create a new flow by selecting **File**, then **New**, and choosing **Flow** in the top navigation bar.

3. (Optional) Rename the new folder and the .flow file.
4. When you create a new .flow file in Studio, you may see a message at the top of the Data Wrangler interface that says:

   **Connecting to engine**

   **Establishing connection to engine...**
5. To get started, choose a data source and use it to import a dataset. See Import (p. 536) to learn more.

When you import a dataset, it appears in your data flow. To learn more, see Create and Use a Data Wrangler Flow (p. 543).

6. After you import a dataset, Data Wrangler automatically infers the type of data in each column. Choose + next to the Data types step and select Edit data types.

   Important
   After you add transforms to the Data types step, you cannot bulk-update column types using Update types.

7. Use the data flow to add transforms and analyses. To learn more see Transform Data (p. 545) and Analyze and Visualize (p. 562).
8. To export a complete data flow, choose **Export** and choose an export option. To learn more, see **Export (p. 569).**

9. Finally, choose the **Components and registries** icon, and select **Data Wrangler** from the dropdown list to see all .flow files you’ve created. You can use this menu to find and move between data flows.

After you have launched Data Wrangler, you can use the following section to walk through how you might use Data Wrangler to create an ML data prep flow.

**Demo: Data Wrangler Titanic Dataset Walkthrough**

The following sections provide a walkthrough to help you get started using Data Wrangler. This walkthrough assumes that you have already followed the steps in **Access Data Wrangler (p. 527)** and have a new data flow file open that you intend to use for the demo. You may want to rename this .flow file to something similar to titanic-demo.flow.

This walk through uses the **Titanic dataset**. This data set contains the survival status, age, gender, and class (which serves as a proxy for economic status) of passengers aboard the maiden voyage of the RMS Titanic in 1912. For more information about how this dataset was constructed see **this description of the Titanic dataset**.

In this tutorial, you:

- Upload the **Titanic dataset** to Amazon Simple Storage Service (Amazon S3), and then import this dataset into Data Wrangler.
- Analyze this dataset using Data Wrangler analyses.
- Define a data flow using Data Wrangler data transforms.
- Export your flow to a Jupyter Notebook that you can use to create a Data Wrangler job.
- Process your data, and kick off a SageMaker training job to train a XGBoost Binary Classifier.

**Upload Dataset to S3 and Import**

To get started, download the **Titanic dataset** and upload it to an S3 bucket in the AWS Region in which you want to complete this demo.

If you are a new user of Amazon S3, you can do this using drag and drop in the Amazon S3 console. To learn how, see **Uploading Files and Folders by Using Drag and Drop** in the Amazon Simple Storage Service Console User Guide.
Important

Upload your dataset to an S3 bucket in the same AWS Region you want to use to complete this demo.

When your dataset has been successfully uploaded to Amazon S3, it you can import it into Data Wrangler.

**Import the Titanic dataset to Data Wrangler**

1. Select the **Import** tab in your Data Wrangler flow file.
2. Select **Amazon S3**.
3. Use the **Import a dataset from S3** table to find the bucket to which you added the Titanic dataset. Choose the Titanic dataset CSV file to open the **Details** pane.
4. Under **Details**, the **File type** should be CSV. Choose **Add header to table** to specify that the first row of the dataset is a header. You can also name the dataset something more friendly, such as **Titanic-train**.
5. Select **Import dataset**.

When your dataset is imported into Data Wrangler, it appears in your data flow. In the next section, you use this data flow to add analysis and transform steps.

**Data Flow**

In the data flow section, you’ll notice that the only steps in the data flow are your recently imported dataset and a **Data type** step. After applying transformations, you can come back to this tab see what the data flow looks like. Now, add some basic transformations under the **Prepare** and **Analyze** tabs.

**Prepare and Visualize**

Data Wrangler has built-in transformations and visualizations that you can use to analyze, clean, and transform your data.

In the **Prepare** tab, all built-in transformations are listed in the right panel, which also contains an area in which you can add custom transformations. The following use case showcases how to use these transformations.
Data Exploration

First, create a table summary of the data using an analysis. Do the following:

1. Choose the + next to the Data type step in your data flow and select Add analysis.
2. In the Analysis area, select Table summary from the dropdown list.
3. Give the table summary a Name.
4. Choose Create to save it to your data flow.

Using the statistics you see, you can make observations similar to the following about this dataset:

- Fare average is around $30, while the max is over $500. This column likely has outliers.
- This dataset uses ? to indicate missing values. A number of columns have missing values: cabin, embarked, and home.dest
- The age category is missing 264 values.

Choose Prepare to go back to the data flow. Next, clean your data using the insights gained from these stats.

Drop Unused Columns

Using the analysis from the previous section, clean up the dataset to prepare it for training. To add a new transform to your data flow, choose + next to the Data type step in your data flow and choose Add transform.

First, drop columns that you don't want to use for training. You can use Pandas data analysis library to do this, or you can use one of the built-in transforms.

To do this using Pandas, do the following:

1. In the Custom Transform section, select Python (Pandas) from the dropdown list.
2. Enter the following in the code box.

```python
cols = ['name', 'ticket', 'cabin', 'sibsp', 'parch', 'home.dest', 'boat', 'body']
df = df.drop(cols, axis=1)
```
3. Choose Preview to preview the change and then choose Add to add the transformation.
To use the built-in transformations, do the following:

1. Choose Manage columns from the right panel.
2. For Input column, choose cabin, and choose Preview.
3. Verify that the cabin column has been dropped, then choose Add.
4. Repeat these steps for the following columns: ticket, name, sibsp, parch, home, dest, boat, and body.

Clean up Missing Values

Now, clean up missing values. You can do this with the Handling missing values transform group.

A number of columns have missing values. Of the remaining columns, age and fare contain missing values. Inspect this using the Custom Transform.

Using the Python (Pandas) option, use the following to quickly review the number of entries in each column:

```python
df.info()
```
To drop rows with missing values in the age category, do the following:

1. Choose **Handling missing values**.
2. Choose *age* for the **Input column**.
3. Choose **Numeric** for the **Input data type**.
4. Choose **Drop missing** for the **Transformer**.
5. Choose **Preview** to see the new data frame, and then choose **Add** to add the transform to your flow.
6. Repeat the same process for *fare*.

You can use `df.info()` in the **Custom transform** section to confirm that all rows now have 1,045 values.

### Custom Pandas: Encode

Try flat encoding using Pandas. Encoding categorical data is the process of creating a numerical representation for categories. For example, if your categories are Dog and Cat, you may encode this information into two vectors: `[1, 0]` to represent Dog, and `[0, 1]` to represent Cat.

1. In the **Custom Transform** section, choose **Python (Pandas)** from the dropdown list.
2. Enter the following in the code box.
import pandas as pd

dummies = []
cols = ['pclass','sex','embarked']
for col in cols:
    dummies.append(pd.get_dummies(df[col]))

encoded = pd.concat(dummies, axis=1)
df = pd.concat((df, encoded), axis=1)

3. Choose Preview to preview the change. The encoded version of each column is added to the dataset.
4. Choose Add to add the transformation.

**Custom SQL: SELECT Columns**

Now, select the columns you want to keep using SQL. For this demo, select the columns listed in the following SELECT statement. Because survived is your target column for training, put that column first.

1. In the Custom Transform section, select SQL (PySpark SQL) from the dropdown list.
2. Enter the following in the code box.

```
SELECT survived, age, fare, 1, 2, 3, female, male, C, Q, S FROM df;
```

3. Choose Preview to preview the change. The columns listed in your SELECT statement above are the only remaining columns.
4. Choose Add to add the transformation.

**Export**

When you've finished creating a data flow, you have a number of export options. The following section explains how to export to a Data Wrangler job notebook. A Data Wrangler job is used to process your data using the steps defined in your data flow. To learn more about all export options, see Export (p. 569).

**Export to Data Wrangler Job Notebook**

When you export your data flow using a Data Wrangler job, a Jupyter Notebook is automatically created. This notebook automatically opens in your Studio instance and is configured to run a SageMaker processing job to execute your Data Wrangler data flow, which is referred to as a Data Wrangler job.

1. Choose the Export tab.
2. Select the last step in your data flow.

3. Choose Data Wrangler Job. This opens a Jupyter Notebook.
4. Choose any Python 3 (Data Science) kernel for the Kernel.
5. When the kernel starts, run all of the steps in the notebook through the code block after the section Start ProcessingJob.
You can monitor your Data Wrangler job status in the SageMaker console in the **Processing** tab. Additionally, you can monitor your Data Wrangler job using Amazon CloudWatch. For additional information, see [Monitor Amazon SageMaker Processing Jobs with CloudWatch Logs and Metrics](#).

### Training XGBoost Classifier

In the same notebook that kicked off the Data Wrangler job, you can pull the data and train a XGBoost Binary Classifier using the prepared data with minimal data preparation.

1. First, upgrade necessary modules using `pip` and remove the `_SUCCESS` file (this last file is problematic when using `awswrangler`).

   ```bash
   pip install --upgrade awscli awswrangler boto sklearn
   aws s3 rm {output_path} --recursive --exclude "*" --include "*_SUCCESS*"
   ```

2. Read the data from Amazon S3. You can use `awswrangler` to recursively read all the CSV files in the S3 prefix. The data is then split into features and labels. The label is the first column of the dataframe.

   ```python
   import awswrangler as wr
   import pandas as pd
   
   df = wr.s3.read_csv(path=output_path, dataset=True)
   X, y = df.iloc[:,:-1], df.iloc[:,-1]
   ```

   - Finally, create DMatrices (the XGBoost primitive structure for data) and do cross-validation using the XGBoost binary classification.

   ```python
   import xgboost as xgb
   
   dmatrix = xgb.DMatrix(data=X, label=y)
   
   params = {
           "objective":"binary:logistic","learning_rate": 0.1, 'max_depth': 5, 
           'alpha': 10
   }
   
   xgb.cv(        
           dtrain=dmatrix, 
           params=params, 
           nfold=3, 
           num_boost_round=50, 
           early_stopping_rounds=10, 
           metrics="rmse", 
           as_pandas=True, 
           seed=123)
   ```

### Import

You can use Amazon SageMaker Data Wrangler to import data from the following *data sources*: Amazon Simple Storage Service (Amazon S3), Amazon Athena, and Amazon Redshift.

Some data sources allow you to add multiple *data connections*:

- You can connect to multiple Amazon Redshift clusters. Each cluster becomes a data source.
- You can query any Athena database in your account to import data from that database.
When you import a dataset from a data source, it appears in your data flow. Data Wrangler automatically infers the data type of each column in your dataset. To modify these types, select the Data types step and select **Edit data types**.

When you import data from Athena or Amazon Redshift, the imported data is automatically stored in the default SageMaker S3 bucket for the AWS Region in which you are using Studio. Additionally, Athena stores data you preview in Data Wrangler in this bucket. To learn more, see Imported Data Storage (p. 542).

**Topics**
- Import data from Amazon S3 (p. 537)
- Import data from Athena (p. 538)
- Import data from Amazon Redshift (p. 539)
- Imported Data Storage (p. 542)

**Import data from Amazon S3**

Amazon Simple Storage Service (Amazon S3) can be used to store and retrieve any amount of data at any time, from anywhere on the web. You can accomplish these tasks using the AWS Management Console, which is a simple and intuitive web interface, and the Amazon S3 API. If your dataset is stored locally, we recommend that you add it to an S3 bucket for import into Data Wrangler. To learn how, see Uploading an object to a bucket in the Amazon Simple Storage Service Developer Guide.

Data Wrangler uses **S3 Select** to allow you to preview your Amazon S3 files in Data Wrangler. You incur standard charges for each file preview. To learn more about pricing, see the Requests & data retrievals tab on Amazon S3 pricing.

**Important**

If you plan to export a data flow and launch a Data Wrangler job, ingest data into a SageMaker feature store, or create a SageMaker pipeline, be aware that these integrations require Amazon S3 input data to be located in the same AWS region used to set up the integration (run the Jupyter Notebook exported from Data Wrangler).

You can browse all buckets in your AWS account and import CSV and Parquet files using the Amazon S3 import in Data Wrangler.

When you choose a dataset for import, you can rename it, specify the file type, and identify the first row as a header.

**To import a dataset into Data Wrangler from Amazon S3:**

1. If you are not currently on the **Import** tab, choose **Import**.
2. Under **Data Preparation**, choose **Amazon S3** to see the **Import S3 Data Source** view.
3. From the table of available S3 buckets, select a bucket and navigate to the dataset you want to import.
4. Select the file that you want to import. You can import CSV and Parquet files. If your dataset does not have a .csv or .parquet extension, select the data type from the **File Type** dropdown list.
5. If your CSV file has a header, select the check box next to **Add header to table**.
6. Use the **Preview** table to preview your dataset. This table shows up to 500 rows.
7. In the **Details** pane, verify or change the **Name** and **File Type** for your dataset. If you add a **Name** that contains spaces, these spaces are replaced with underscores when your dataset is imported.
8. **Enable sampling** is selected by default. If you do not uncheck this box, Data Wrangler will sample and import up to 100 MB of data. To disable sampling, uncheck this check box.
9. Choose **Import dataset**.

### Import data from Athena

Amazon Athena is an interactive query service that makes it easy to analyze data directly in Amazon S3 using standard SQL. With a few actions in the AWS Management Console, you can point Athena at your data stored in Amazon S3 and begin using standard SQL to run ad-hoc queries and get results in seconds. To learn more, see What is Amazon Athena? in the Amazon Athena User Guide.

You can query Athena databases and import the results in Data Wrangler. To use this import option, you must create at least one database in Athena. To learn how, see Getting Started in the Amazon Athena User Guide.

Note the following about the Athena import option in Data Wrangler:

- Data Wrangler supports using an Athena primary workgroup. Other workgroups are not supported.
- Data Wrangler does not support federate queries.

Data Wrangler uses the default S3 bucket in the same AWS Region in which your Studio instance is located to store Athena query results. When you import from Athena, Data Wrangler creates a new
database in your Athena account named sagemaker_data_wrangler if one does not already exist. Temporary tables are created in this database to move the query output to this S3 bucket. These tables are deleted after data has been imported, however the database, sagemaker_data_wrangler, persists. To learn more, see Imported Data Storage (p. 542).

If you use AWS Lake Formation with Athena, make sure your Lake Formation IAM permissions do not override IAM permissions for the database sagemaker_data_wrangler.

To import a dataset into Data Wrangler from Athena:

1. On the import screen, choose Amazon Athena.
2. For Catalog, choose AWSDataCatalog.
3. Use the Database dropdown list to select the database that you want to query. When you select a database, you can preview all tables in your database using the Tables listed under Details.
4. Enter a query in the code box.
5. Under Advanced configuration, Enable sampling is selected by default. If you do not uncheck this box, Data Wrangler samples and imports approximately 50% of the queried data. Unselect this check box to disable sampling.
6. Enter your query in the query editor and use the Run button to run the query. After a successful query, you can preview your result under the editor.
7. To import the queried results, select Import dataset.
8. Enter a Dataset name. If you add a Dataset name that contains spaces, these spaces are replaced with underscores when your dataset is imported.
9. Select Add.

Import data from Amazon Redshift

Amazon Redshift is a fully managed, petabyte-scale data warehouse service in the cloud. The first step to create a data warehouse is to launch a set of nodes, called an Amazon Redshift cluster. After you provision your cluster, you can upload your dataset and then perform data analysis queries.
You can connect to and query one or more Amazon Redshift clusters in Data Wrangler. To use this import option, you must create at least one cluster in Amazon Redshift. To learn how, see Getting started with Amazon Redshift.

Data Wrangler uses the default S3 bucket in the same AWS Region in which your Studio instance is located to store Amazon Redshift query results. To learn more, see Imported Data Storage (p. 542).

If you use the IAM managed policy, AmazonSageMakerFullAccess, to grant a role permission to use Data Wrangler in Studio, your Database User name must have the prefix sagemaker_access.

Use the following procedures to learn how to add a new cluster.

**Note**
Data Wrangler uses the Amazon Redshift Data API with temporary credentials. To learn more about this API, refer to Using the Amazon Redshift Data API in the Amazon Redshift Cluster Management Guide.

**To connect to a Amazon Redshift cluster:**

1. Choose **Import**.
2. Choose + under **Add data connection**.
3. Choose **Amazon Redshift**.
4. Choose **Temporary credentials (IAM)** for **Type**.
5. Enter a **Connection Name**. This is a name used by Data Wrangler to identify this connection.
6. Enter the **Cluster Identifier** to specify to which cluster you want to connect.
7. Enter the **Database Name** of the database that you want to connect.
8. Enter a **Database User** to identify the user you want to use to connect to the database.
9. For **UNLOAD IAM Role**, enter the IAM role ARN of the role that the Amazon Redshift cluster should assume to move and write data to Amazon S3. For more information about this role, see Authorizing Amazon Redshift to access other AWS services on your behalf in the Amazon Redshift Cluster Management Guide.
10. Choose **Connect**.
After your connection is successfully established, it appears as a data source under Data Import. Select this data source to query your database and import data.

To query and import data from Redshift:

1. Select the connection that you want to query from Data Sources.
2. Select a Schema. To learn more about Redshift Schemas, see Schemas in the Amazon Redshift Database Developer Guide.
3. Under Advanced configuration, Enable sampling is selected by default. If you do not uncheck this box, Data Wrangler samples and imports approximately 50% of the queried data. Unselect this check box to disable sampling.
4. Enter your query in the query editor and use the Run button to run the query. After a successful query, you can preview your result under the editor.
5. Select Import dataset to import the dataset that has been queried.
6. Enter a Dataset name. If you add a Dataset name that contains spaces, these spaces are replaced with underscores when your dataset is imported.
7. Select Add.
Imported Data Storage

When you query data from Amazon Athena or Amazon Redshift, the queried dataset is automatically stored in Amazon S3. Data is stored in the default SageMaker S3 bucket for the AWS Region in which you are using Studio.

Default S3 buckets have the following naming convention: sagemaker-region-account_number. For example, if your account number is 111122223333 and you are using Studio in us-east-1, your imported datasets are stored in sagemaker-us-east-1-111122223333.

Data Wrangler flows depend on this S3 dataset location, so you should not modify this dataset in Amazon S3 while you are using a dependent flow. If you do modify this S3 location, and you want to continue using your data flow, you must remove all objects in trained_parameters in your .flow file. To do this, download the .flow file from Studio and for each instance of trained_parameters, delete all entries. When you are done, trained_parameters should be an empty JSON object:

"trained_parameters": {}

When you export and use your data flow to process your data, the .flow file you export refers to this dataset in Amazon S3. Use the following sections to learn more.

Amazon Redshift Import Storage

Data Wrangler stores the datasets that result from your query in a Parquet file in your default SageMaker S3 bucket.

This file is stored under the following prefix (folder): redshift/uuid/data/, where uuid is a unique identifier that gets created for each query.

For example, if your default bucket is sagemaker-us-east-1-111122223333, a single dataset queried from Amazon Redshift is located in s3://sagemaker-us-east-1-111122223333/redshift/uuid/data/.
Amazon Athena Import Storage

When you query an Athena database and import a dataset, Data Wrangler stores the dataset, as well as a subset of that dataset, or preview files, in Amazon S3.

The dataset you import by selecting **Import dataset** is stored in Parquet format in Amazon S3.

Preview files are written in CSV format when you select **Run** on the Athena import screen, and contain up to 100 rows from your queried dataset.

The dataset you query is located under the prefix (folder): athena/uuid/data/, where **uuid** is a unique identifier that gets created for each query.

For example, if your default bucket is sagemaker-us-east-1-111122223333, a single dataset queried from Athena is located in s3://sagemaker-us-east-1-111122223333/redshift/uuid/data/example_dataset.parquet.

The subset of the dataset that is stored to preview dataframes in Data Wrangler is stored under the prefix: athena/.

Create and Use a Data Wrangler Flow

Use an Amazon SageMaker Data Wrangler flow, or a **data flow**, to create and modify a data preparation pipeline. The data flow connects the datasets, transformations, and analyses, or **steps**, you create and can be used to define your pipeline.

The Data Flow UI

When you import a dataset, the original dataset appears on the data flow and is named **Source**. If you enabled sampling when you imported your data, this dataset is named **Source - sampled**. Data Wrangler automatically infers the types of each column in your dataset and creates a new dataframe named **Data types**. You can select this frame to update the inferred data types. You will see results similar to those shown in the following image after you upload a single dataset:
Each time you add a transform step, you create a new dataframe. When multiple transform steps (other than Join or Concatenate) are added to the same dataset, they are stacked.

Join and Concatenate create stand-alone steps that contain the new joined or concatenated dataset.

The following diagram shows a data flow with a join between two datasets, as well as two stacks of steps. The first stack (Steps (2)) adds two transforms to the type inferred in the Data types dataset. The downstream stack, or the stack to the right, adds transforms to the dataset resulting from a join named demo-join.

The small, gray box in the bottom-right corner of the data flow provides an overview of number of stacks and steps in the flow and the layout of the flow. The lighter box inside the gray box indicates the steps that are within the UI view. You can use this box to see sections of your data flow that fall outside of the UI view. Use the fit screen icon ( ) to fill all steps and datasets into your UI view.

The bottom left navigation bar includes icons that you can use to zoom in ( ) and out ( ) of your data flow and resize the data flow to fit the screen.
Add a Step to Your Data Flow

Select + next to any dataset or previously added step and then select one of the following options:

- **Edit data types** (For Data types step only): If you have not added any transforms to a Data types step, you can select Edit data types to update the data types Data Wrangler inferred when importing your dataset.

- **Add transform**: Adds a new transform step. See Transform Data (p. 545) to learn more about the data transformations you can add.

- **Add analysis**: Adds an analysis. You can use this option to analyze your data at any point in the data flow. When you add one or more analyses to a step, an analysis icon ( altında) appears on that step. See Analyze and Visualize (p. 562) to learn more about the analyses you can add.

- **Join**: Joins two datasets and adds the resulting dataset to the data flow. To learn more, see Join Datasets (p. 550).

- **Concatenate**: Concatenates two datasets and adds the resulting dataset to the data flow. To learn more, see Concatenate Datasets (p. 550).

Delete a step from your Data Flow

To delete a step, select the step and select Delete. When you delete a step, all subsequent steps connected to that step, or downstream steps, are also deleted.

To delete a step from a stack of steps, select the stack and then select the step you want to delete.

Note
You cannot delete the Data type step directly. To remove this dataset, you must delete the corresponding Source dataset.

Transform Data

Amazon SageMaker Data Wrangler provides numerous ML data transforms to streamline cleaning, transforming, and featurizing your data. When you add a transform, it adds a step to the data flow. Each transform you add modifies your dataset and produces a new dataframe. All subsequent transforms apply to the resulting dataframe.

Data Wrangler includes built-in transforms, which you can use to transform columns without any code. You can also add custom transformations using PySpark, Pandas, and PySpark SQL. Some transforms operate in place, while others create a new output column in your dataset.

Use this page to learn more about these built-in and custom transforms.
**Transform UI**

Most of the built-in transforms are located in the **Prepare** tab of the Data Wrangler UI. The Join and Concatenate transforms are accessed through the data flow view. Use the following table to preview these two views.

**Join View**

To join two datasets, select the first dataset in your data flow and choose **Join**. When you select **Join**, you see results similar to those shown in the following image. Your left and right datasets are displayed in the left panel. The main panel displays your data flow, with the newly joined dataset added.

When you select **Configure** to configure your join, you see results similar to those shown in the following image. Your join configuration is displayed in the left panel. You can use this panel to choose the joined dataset name, join type, and columns to join on. The main panel displays three tables. The top-two tables display the Left and Right datasets on the left and right respectively. Under this table, you can preview the joined dataset.
See Join Datasets (p. 550) to learn more.

Concatenate View

To concatenate two datasets, you select the first dataset in your data flow and choose **Concatenate**. When you select **Concatenate**, you see results similar to those shown in the following image. Your left and right datasets are displayed in the left panel. The main panel displays your data flow, with the newly concatenated dataset added.

When you select **Configure** to configure your concatenation, you see results similar to those shown in the following image. Your concatenate configuration is display in the left panel. You can use this panel to choose the concatenated dataset's name, and choose to remove duplicates after...
concatenation and add columns to indicate the source dataframe. The main panel displays three tables. The top-two tables display the Left and Right datasets on the left and right respectively. Under this table, you can preview the concatenated dataset.

See Concatenate Datasets (p. 550) to learn more.

Transforms in the Prepare Tab

To access transforms on the Prepare tab, select + next to a step in your data flow, and select Add transform.

On the Prepare tab, you add steps under Add.
You can use the Previous steps tab to view and remove transformations that you have added, in sequential order.
Join Datasets

You join dataframes directly in your data flow. When you join two datasets, the resulting joined dataset appears in your flow. The following join types are supported by Data Wrangler.

- **Left Outer** – Include all rows from the left table. If the value for the column joined on in a left table row does not match any right table row values, that row contains null values for all right table columns in the joined table.
- **Left Anti** – Include rows from the left table that do not contain values in the right table for the joined column.
- **Left semi** – Include a single row from the left table for all identical rows that satisfy the criteria in the join statement. This excludes duplicate rows from the left table that match the criteria of the join.
- **Right Outer** – Include all rows from the right table. If the value for the joined column in a right table row does not match any left table row values, that row contains null values for all left table columns in the joined table.
- **Inner** – Include rows from left and right tables that contain matching values in the joined column.
- **Full Outer** – Include all rows from the left and right tables. If the row value for the joined column in either table does not match, separate rows are created in the joined table. If a row doesn't contain a value for a column in the joined table, null is inserted for that column.
- **Cartesian Cross** – Include rows which combine each row from the first table with each row from the second table. This is a Cartesian product of rows from tables in the join. The result of this product is the size of the left table times the size of the right table. Therefore, we recommend caution in using this join between very large datasets.

Use the following procedure to join two dataframes.

1. Select + next to the left dataframe that you want to join. The first dataframe you select is always the left table in your join.
2. Select Join.
3. Select the right dataframe. The second dataframe you select is always the right table in your join.
4. Select Configure to configure your join.
5. Give your joined dataset a name using the Name field.
6. Select a Join type.
7. Select a column from the left and right tables to join on.
8. Select Apply to preview the joined dataset on the right.
9. To add the joined table to your data flow, select Add in the top right corner.

Concatenate Datasets

**Concatenate two datasets:**

1. Select + next to the left dataframe that you want to concatenate. The first dataframe you select is always the left table in your concatenate.
2. Select Concatenate.
3. Select the right dataframe. The second dataframe you select is always the right table in your concatenate.
4. Select Configure to configure your concatenate.
5. Give your concatenated dataset a name using the Name field.
6. (Optional) Select the check box next to Remove duplicates after concatenation to remove duplicate columns.
7. (Optional) Select the check box next to Add column to indicate source dataframe if, for each column in the new dataset, you want to add an indicator of the column's source.
8. Select Apply to preview the new dataset.
9. Select Add to add the new dataset to your data flow.

Custom Transforms

The Custom Transforms group allows you to use Pyspark, Pandas, or Pyspark (SQL) to define custom transformations. For all three options, you use the variable df to access the dataframe to which you want to apply the transform. You do not need to include a return statement. Select Preview to preview the result of the custom transform. Select Add to add the custom transform to your list of Previous steps.

You can import the popular libraries with an import statement in the custom transform code block such as the following:

- Numpy version 1.19.0
- Scikit-learn version 0.23.2
- Scipy version 1.5.4
- Pandas version 1.0.3
- Pyspark version 3.0.0

If you include print statements in the code block, the result appears when you select Preview.

The following are examples of how you can use the custom transforms code block:

**Pyspark**

The following example extracts date and time from a timestamp.

```python
from pyspark.sql.functions import from_unixtime, to_date, date_format
df = df.withColumn('DATE_TIME', from_unixtime('TIMESTAMP'))
df = df.withColumn('EVENT_DATE', to_date('DATE_TIME')).withColumn('EVENT_TIME', date_format('DATE_TIME', 'HH:mm:ss'))
```

**Pandas**

The following example provides an overview of the dataframe to which you are adding transforms.

```python
df.info()
```

**Pyspark (SQL)**

The following creates a new dataframe with five columns: name, fare, pclass, survived.

```sql
SELECT name, fare, pclass, survived FROM df
```

**Custom Formula**

Use Custom formula to define a new column using a Spark SQL expression to query data in the current dataframe. The query must use the conventions of Spark SQL expressions.
Important

Custom formula does not support columns with spaces in the name. You can use the Rename column transform in the Manage columns transform group to remove spaces from a column's name. You can also add a Pandas Custom transform similar to the following to remove spaces from multiple columns in a single step. This example changes columns named A column and B column to A_column and B_column respectively.

```
df.rename(columns={"A column": "A_column", "B column": "B_column"})
```

You can use this transform to perform operations on columns, referencing the columns by name. For example, assuming the current dataframe contains columns named col_a and col_b, you can use the following operation to produce an Output column that is the product of these two columns with the following code:

```
col_a * col_b
```

Other common operations include the following, assuming a dataframe contains col_a and col_b columns:

- Concatenate two columns: `concat(col_a, col_b)`
- Add two columns: `col_a + col_b`
- Subtract two columns: `col_a - col_b`
- Divide two columns: `col_a / col_b`
- Take the absolute value of a column: `abs(col_a)`

For more information, see the Spark documentation on selecting data.

Encode Categorical

Categorical data is usually composed of a finite number of categories, where each category is represented with a string. For example, if you have a table of customer data, a column that indicates the country a person lives in is categorical. The categories would be Afghanistan, Albania, Algeria, and so on. Categorical data can be nominal or ordinal. Ordinal categories have an inherent order, and nominal categories do not. The highest degree obtained (High school, Bachelors, Master) is an example of ordinal categories.

Encoding categorical data is the process of creating a numerical representation for categories. For example, if your categories are Dog and Cat, you may encode this information into two vectors, [1, 0] to represent dog, and [0, 1] to represent cat.

When you encode ordinal categories, you may need to translate the natural order of categories into your encoding. For example, you can represent highest degree obtained with the following map: {"High school": 1, "Bachelors": 2, "Masters":3}.

Use categorical encoding to encode categorical data that is in string format into arrays of integers.

The Data Wrangler categorical encoders create encodings for all categories that exist in a column at the time the step is defined. If new categories have been added to a column when you start a Data Wrangler job to process your dataset at time t, and this column was the input for a Data Wrangler categorical encoding transform at time t-1, these new categories are considered missing in the Data Wrangler job. The option you select for Invalid handling strategy is applied to these missing values. Examples of when this can occur are:

- When you use a .flow file to create a Data Wrangler job to process a dataset that was updated after the creation of the data flow. For example, you may use a data flow to regularly process sales data each
month. If that sales data is updated weekly, new categories may be introduced into columns for which an encode categorical step is defined.

- When you select Sampling when you import your dataset, some categories may be left out of the sample.

In these situations, these new categories are considered missing values in the Data Wrangler job.

You can choose from and configure an ordinal and a one-hot encode. Use the following sections to learn more about these options.

Both transforms create a new column named **Output column name**. You specify the output format of this column with **Output style**:

- Select Vector to produce a single column with a sparse vector.
- Select Columns to create a column for every category with an indicator variable for whether the text in the original column contains a value that is equal to that category.

### Ordinal Encode

Select **Ordinal encode** to encode categories into an integer between 0 and the total number of categories in the **Input column** you select.

**Invalid handing strategy**: Select a method to handle invalid or missing values.

- Choose Skip if you want to omit the rows with missing values.
- Choose Keep to retain missing values as the last category.
- Choose Error if you want Data Wrangler to throw an error if missing values are encountered in the **Input column**.
- Choose Replace with NaN to replace missing with NaN. This option is recommended if your ML algorithm can handle missing values. Otherwise, the first three options in this list may produce better results.

### One-Hot Encode

Select **One-hot encode** for Transform to use one-hot encoding. Configure this transform using the following:

- **Drop last category**: If True, the last category does not have a corresponding index in the one-hot encoding. When missing values are possible, a missing category is always the last one and setting this to True means that a missing value results in an all zero vector.
- **Invalid handing strategy**: Select a method to handle invalid or missing values.
  - Choose Skip if you want to omit the rows with missing values.
  - Choose Keep to retain missing values as the last category.
  - Choose Error if you want Data Wrangler to throw an error if missing values are encountered in the **Input column**.
- **Is input ordinal encoded**: Select this option if the input vector contains ordinal encoded data. This option requires that input data contain non-negative integers. If True, input i is encoded as a vector with a non-zero in the i-th location.

### Featurize Text

Use the **Feature Text** transform group to inspect string typed columns and use text embedding to featurize these columns.
This feature group contains two features, Character statistics and Vectorize. Use the following sections to learn more about these transforms. For both options, the input column must contain text data (string type).

**Character Statistics**

Use **Character statistics** to generate statistics for each row in a column containing text data.

This transforms computes the following ratios and counts for each row, and creates a new column to report the result. The new column is named using the input column name as a prefix and a suffix that is specific to the ratio or count.

- **Number of words:** The total number of words in that row. The suffix for this output column is `-stats_word_count`.
- **Number of characters:** The total number of characters in that row. The suffix for this output column is `-stats_char_count`.
- **Ratio of upper:** The number of upper-case characters, from A to Z, divided by all characters in the column. The suffix for this output column is `-stats_capital_ratio`.
- **Ratio of lower:** The number of lower-case characters, from a to z, divided by all characters in the column. The suffix for this output column is `-stats_lower_ratio`.
- **Ratio of digits:** The ratio of digits in a single row over the sum of digits in the input column. The suffix for this output column is `-stats_digit_ratio`.
- **Special characters ratio:** The ratio of non-alphanumeric (characters like #$&%:@) characters to over the sum of all characters in the input column. The suffix for this output column is `-stats_special_ratio`.

**Vectorize**

Text embedding involves mapping words or phrases from a vocabulary to vectors of real numbers. Use the Data Wrangler text embedding transform to tokenize and vectorize text data into term frequency-inverse document frequency (TF-IDF) vectors.

When TF-IDF is calculated for a column of text data, each word in each sentence is converted to a real number that represents its semantic importance. Higher numbers are associated with less frequent words, which tend to be more meaningful.

When you define a Vectorize transform step, the count vectorizer and TF-IDF methods are defined using data available in Data Wrangler when defining this step. These same methods will be used when running a Data Wrangler job.

You configure this transform using the following:

- **Output column name:** This transform will create a new column with the text embedding. Use this field to specify a name for this output column.
- **Tokenizer:** A tokenizer converts the sentence into a list of words, or tokens.

  Choose **Standard** to use a tokenizer that splits by white space and converts each word to lowercase. For example, “Good dog” is tokenized to [“good”, “dog”].

  Choose **Custom** to use a customized tokenizer. If you choose Custom, you can use the following fields to configure the tokenizer:

  - **Minimum token length:** The minimum length, in characters, for a token to be valid. Defaults to 1. For example, if you specify 3 for minimum token length, words like a, at, in are dropped from the tokenized sentence.

  - **Should regex split on gaps:** If selected, regex splits on gaps. Otherwise, it matches tokens. Defaults to True.

  - **Regex pattern:** Regex pattern that defines the tokenization process. Defaults to `\s+`. 


• To lowercase: If chosen, all characters are converted to lowercase before tokenization. Defaults to True.

To learn more, refer to the Spark documentation on Tokenizer.

• Vectorizer: The vectorizer converts the list of tokens into a sparse numeric vector. Each token corresponds to an index in the vector and a non-zero indicates the existence of the token in the input sentence. You can choose from two vectorizer options, Count and Hashing.

• Count vectorize allows customizations that filter infrequent or too common tokens. Count vectorize parameters include the following:
  • Minimum term frequency: In each row, terms (tokens) with smaller frequency are filtered. If you specify an integer, this is an absolute threshold (inclusive). If you specify a fraction between 0 (inclusive) and 1, the threshold is relative to the total term count. Defaults to 1.
  • Minimum document frequency: Minimum number of rows in which a term (token) must appear to be included. If you specify an integer, this is an absolute threshold (inclusive). If you specify a fraction between 0 (inclusive) and 1, the threshold is relative to the total term count. Defaults to 1.
  • Maximum document frequency: Maximum number of documents (rows) in which a term (token) can appear to be included. If you specify an integer, this is an absolute threshold (inclusive). If you specify a fraction between 0 (inclusive) and 1, the threshold is relative to the total term count. Defaults to 0.999.
  • Maximum vocabulary size: Maximum size of the vocabulary. The vocabulary is made up of all terms (tokens) in all rows of the column. Defaults to 262144.
  • Binary outputs: If selected, the vector outputs do not include the number of appearances of a term in a document, but rather a binary indicator of its appearance. Defaults to False.

To learn more about this option, refer to the Spark documentation on CountVectorizer.

• Hashing is computationally faster. Hash vectorize parameters includes the following:
  • Number of features during hashing: A hash vectorizer maps tokens to a vector index according to their hash value. This feature determines the number of possible hash values. Large values result in fewer collisions between hash values but a higher dimension output vector.

To learn more about this option, refer to the Spark documentation on FeatureHasher.

• Apply IDF: When chosen, an IDF transformation is applied, which multiplies the term frequency with the standard inverse document frequency used for TF-IDF embedding. IDF parameters include the following:
  • Minimum document frequency: Minimum number of documents (rows) in which a term (token) must appear to be included. If count_vectorize is the chosen vectorizer, we recommend that you keep the default value and only modify the min_doc_freq field in Count vectorize parameters. Defaults to 5.

• Output format: The output format of each row.
  • Select Vector to produce a single column with a sparse vector.
  • Select Flattened to create a column for every category with an indicator variable for whether the text in the original column contains a value that is equal to that category. You can only choose flattened when Vectorizer is set as Count vectorizer.

Featurize Date/Time

Use Featurize date/time to create a vector embedding representing a date/time field. To use this transform, your date/time data must be in one of the following formats:

• Strings describing date/time, for example, "January 1st, 2020, 12:44pm".
• A unix timestamp. A unix timestamp describes the number of seconds, milliseconds, microseconds, or nanoseconds from 1/1/1970.
You can choose to infer datetime format and provide a datetime format. If you provide a date/time format, you must use the codes described in this Python documentation. The options you select for these two configurations have implications for the speed of the operation, and the final results:

- The most manual and computationally fastest option is to specify a Datetime format and select No for Infer datetime format.
- To reduce manual labor, you can simply choose Infer datetime format and not specify a date/time format. This is also a computationally fast operation; however, the first date/time format encountered in the input column is assumed to be the format for the entire column. Therefore, if other formats are encountered in the column, these values are NaN in the final output. Therefore, this option can result in unparsed strings.
- If you do not specify a format and you select No for Infer datetime format, you get the most robust results. All valid date/time strings are parsed. However, this operation can be an order of magnitude slower than the first two options in this list.

When you use this transform, you specify an Input column which contains date/time data in one of the formats listed above. The transform creates an output column named Output column name. The format of the output column depends on your configuration using the following:

- **Vector**: Outputs a single column with a sparse vector.
- **Columns**: Creates a new column for every feature. For example, if the output contains a year, month, and day, three separate columns are created for year, month, and day.

Additionally, you must choose an Embedding mode. For linear models and deep networks, cyclic is recommended. For tree based algorithms, ordinal is recommended.

**Format String**

The Format string transforms contain standard string formatting operations. For example, you can use these operations to remove special characters, normalize string lengths, and update string casing.

This feature group contains the following transforms. All transforms return copies of the strings in the Input column and add the result to a new, output column.

<table>
<thead>
<tr>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Left pad</td>
<td>Left-pad the string with a given Fill character to the given width. If the string is longer than width, the return value is shortened to width characters.</td>
</tr>
<tr>
<td>Right pad</td>
<td>Right-pad the string with a given Fill character to the given width. If the string is longer than width, the return value is shortened to width characters.</td>
</tr>
<tr>
<td>Center (pad on either side)</td>
<td>Center-pad the string (add padding on both sides of the string) with a given Fill character to the given width. If the string is longer than width, the return value is shortened to width characters.</td>
</tr>
<tr>
<td>Prepend zeros</td>
<td>Left-fill a numeric string with zeros, up to a given width. If the string is longer than width, the return value is shortened to width characters.</td>
</tr>
<tr>
<td>Strip left and right</td>
<td>Returns a copy of the string with the leading and trailing characters removed.</td>
</tr>
</tbody>
</table>
### Handle Outliers

Machine learning models are sensitive to the distribution and range of your feature values. Outliers, or rare values, can negatively impact model accuracy and lead to longer training times. Use this feature group to detect and update outliers in your dataset.

When you define a **Handle outliers** transform step, the statistics used to detect outliers are generated on the data available in Data Wrangler when defining this step. These same statistics are used when running a Data Wrangler job.

Use the following sections to learn more about the transforms this group contains. You specify an **Output name** and each of these transforms produces an output column with the resulting data.

#### Robust standard deviation numeric outliers

This transform detects and fixes outliers in numeric features using statistics that are robust to outliers.

You must define an **Upper quantile** and a **Lower quantile**, which are used in the statistics used to calculate outliers. You also need to specify the number of **Standard deviations** from which a value must vary from the mean to be considered an outlier. For example, if you specify 3 for **Standard deviations**, a value must fall more than 3 standard deviations from the mean to be considered an outlier.

The **Fix method** is the method used to handle outliers when they are detected. You can choose from the following:

- **Clip**: Use this option to clip the outliers to the corresponding outlier detection bound.
- **Remove**: Use this to remove rows with outliers from the dataframe.
- **Invalidate**: Use this to replace outliers with invalid values.

#### Standard Deviation Numeric Outliers

This transform detects and fixes outliers in numeric features using the mean and standard deviation.
You specify the number of **Standard deviations** a value must vary from the mean to be considered an outlier. For example, if you specify 3 for **Standard deviations**, a value must fall more than 3 standard deviations from the mean to be considered an outlier.

The **Fix method** is the method used to handle outliers when they are detected. You can choose from the following:

- **Clip**: Use this option to clip the outliers to the corresponding outlier detection bound.
- **Remove**: Use this to remove rows with outliers from the dataframe.
- **Invalidate**: Use this to replace outliers with invalid values.

**Quantile Numeric Outliers**

Use this transform to detect and fix outliers in numeric features using quantiles. You can define an **Upper quantile** and a **Lower quantile**, and all values that fall above or below those quantile-values, respectively, are considered outliers.

The **Fix method** is the method used to handle outliers when they are detected. You can choose from the following:

- **Clip**: Use this option to clip the outliers to the corresponding outlier detection bound.
- **Remove**: Use this to remove rows with outliers from the dataframe.
- **Invalidate**: Use this to replace outliers with invalid values.

**Min-Max Numeric Outliers**

This transform detects and fixes outliers in numeric features using upper and lower thresholds. Use this method if you know threshold values that demark outliers.

You specify a **Upper threshold** and a **Lower threshold**, and if values fall above or below those thresholds respectively, they are considered outliers.

The **Fix method** is the method used to handle outliers when they are detected. You can choose from the following:

- **Clip**: Use this option to clip the outliers to the corresponding outlier detection bound.
- **Remove**: Use this to remove rows with outliers from the dataframe.
- **Invalidate**: Use this to replace outliers with invalid values.

**Replace Rare**

When you use the **Replace rare** transform, you specify a threshold and Data Wrangler finds all values that meet that threshold and replaces them with a string that you specify. For example, you may want to use this transform to categorize all outliers in a column into an "Others" category.

- **Replacement string**: The string with which to replace outliers.
- **Absolute threshold**: A category is rare if the number of instances is less than or equal to this absolute threshold.
- **Fraction threshold**: A category is rare if the number of instances is less than or equal to this fraction threshold multiplied by the number of rows.
- **Max common categories**: Maximum not-rare categories that remain after the operation. If the threshold does not filter enough categories, those with the top number of appearances are classified as not rare. If set to 0 (default), there is no hard limit to the number of categories.
Handle Missing Values

Missing values are a common occurrence in machine learning datasets. In some situations, it is appropriate to impute missing data with a calculated value, such as an average or categorically common value. You can process missing values using the **Handle missing values** transform group. This group contains the following transforms.

**Fill Missing**

Use the **Fill missing** transform to replace missing values with a **Fill value** you define.

**Impute Missing**

Use the **Impute missing** transform to create a new column that contains imputed values where missing values were found in input categorical and numerical data. The configuration depends on your data type. Configure this transform using the following:

- **Imputing strategy**: The strategy used to determine the new value to impute.
  - For numeric data, the chosen statistic is computed over the present values and is used as the imputed value for all missing values. The choices are **mean** and **median**.
  - For categorical data, you can choose to impute the **Most frequent value** in the column, or you can define a custom string to impute.

**Add Indicator for Missing**

Use the **Add indicator for missing** transform to create a new indicator column, which contains a 1 if a row contains a value, and a 0 if a row contains a missing value.

**Drop Missing**

Use the **Drop missing** option to drop rows that contain missing values from the **Input column**.

**Manage Columns**

You can use the following transforms to quickly update and manage columns in your dataset:

<table>
<thead>
<tr>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Drop Column</td>
<td>Delete a column.</td>
</tr>
<tr>
<td>Duplicate Column</td>
<td>Duplicate a column.</td>
</tr>
<tr>
<td>Rename Column</td>
<td>Rename a column.</td>
</tr>
<tr>
<td>Move Column</td>
<td>Move a column's location in the dataset. Choose to move your column to the start or end of the dataset, before or after a reference column, or to a specific index.</td>
</tr>
</tbody>
</table>

**Manage Rows**

Use this transform group to quickly perform sort and shuffle operations on rows. This group contains the following:
• **Sort**: Sort the entire dataframe by a given column. Select the check box next to **Ascending order** for this option; otherwise, deselect the check box and descending order is used for the sort.

• **Shuffle**: Randomly shuffle all rows in the dataset.

### Manage Vectors

Use this transform group to combine or flatten vector columns. This group contains the following transforms.

• **Assemble**: Use this transform to combine Spark vectors and numeric data into a single column. For example, you can combine three columns: two containing numeric data and one containing vectors. Add all the columns you want to combine in **Input columns** and specify a **Output column name** for the combined data.

• **Flatten**: Use this transform to flatten a single column containing vector data. The input column must contain PySpark vectors, or array-like objects. You can control the number of columns that get created by specifying a **Method to detect number of outputs**. For example, if you select **Length of first vector**, the number of elements in the first valid vector or array found in the column determines the number of output columns that gets created. All other input vectors with too many items are truncated. Inputs with too few items are filled with NaNs.

You also specify an **Output prefix**, which is used as the prefix for each output column.

### Process Numeric

Use the **Process Numeric** feature group to process numeric data. Each scalar in this group is defined using the Spark library. The following scalars are supported:

• **Standard Scaler**: Standardize the input column by subtracting the mean from each value and scaling to unit variance. To learn more, refer to the Spark documentation for `StandardScaler`.

• **Robust Scaler**: Scale the input column using statistics that are robust to outliers. To learn more, refer to the Spark documentation for `RobustScaler`.

• **Min Max Scaler**: Transform the input column by scaling each feature to a given range. To learn more, refer to the Spark documentation for `MinMaxScaler`.

• **Max Absolute Scaler**: Scale the input column by dividing each value by the maximum absolute value. To learn more, refer to the Spark documentation for `MaxAbsScaler`.

### Search and Edit

Use this section to search for and edit specific patterns within strings. For example, you can find and update strings within sentences or documents, split strings by delimiters, and find occurrences of specific strings.

The following transforms are supported under **Search and edit**. All transforms return copies of the strings in the **Input column** and add the result to a new output column.

<table>
<thead>
<tr>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Find substring</td>
<td>Returns the index of the first occurrence of the <strong>Substring</strong> for which you searched optionally, starting and ending the search at <strong>Start</strong> and <strong>End</strong> respectively.</td>
</tr>
<tr>
<td>Find substring (from right)</td>
<td>Returns the index of the last occurrence of the <strong>Substring</strong> for which you searched, optionally,</td>
</tr>
<tr>
<td>Name</td>
<td>Function</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>starting and ending the search at Start and End respectively.</td>
<td>Matches prefix Returns a Boolean if the string contains a given Pattern. A pattern can be a character sequence or regular expression. Optionally, you can make the pattern case sensitive.</td>
</tr>
<tr>
<td></td>
<td>Find all occurrences Returns an array with all occurrences of a given pattern. A pattern can be a character sequence or regular expression.</td>
</tr>
<tr>
<td></td>
<td>Extract using regex Returns a string that matches a given Regex pattern.</td>
</tr>
<tr>
<td></td>
<td>Extract between delimiters Returns a string with all characters found between Left delimiter and Right delimiter.</td>
</tr>
<tr>
<td></td>
<td>Extract from position Returns a string, starting from Start position in the input string, that contains all characters up to the start position plus Length.</td>
</tr>
<tr>
<td></td>
<td>Find and replace substring Returns a string with all matches of a given Pattern (regular expression) replaced by Replacement string.</td>
</tr>
<tr>
<td></td>
<td>Replace between delimiters Returns a string with the substring found between the first appearance of a Left delimiter and the last appearance of a Right delimiter replaced by Replacement string. If no match is found, nothing is replaced.</td>
</tr>
<tr>
<td></td>
<td>Replace from position Returns a string with the substring between Start position and Start position plus Length replaced by Replacement string. If Start position plus Length is greater than the length of the replacement string, the output contains ....</td>
</tr>
<tr>
<td></td>
<td>Convert regex to missing Converts a string to None if invalid and returns the result. Validity is defined with a regular expression in Pattern.</td>
</tr>
<tr>
<td></td>
<td>Split string by delimiter Returns an array of strings from the input string, split by Delimiter, with up to Max number of splits (optional). Delimiter defaults to white space.</td>
</tr>
</tbody>
</table>

**Parse Value as Type**

Use this transform to cast a column to a new type. The supported Data Wrangler data types are:

- Long
- Float
- Boolean
- Date, in the format dd-MM-yyyy, representing day, month, and year respectively.
• String

**Validate String**

Use the **Validate string** transforms to create a new column that indicates that a row of text data meets a specified condition. For example, you can use a **Validate string** transform to verify that a string only contains lowercase characters. The following transforms are supported under **Validate string**.

The following transforms are included in this transform group. If a transform outputs a Boolean value, **True** is represented with a 1 and **False** is represented with a 0.

<table>
<thead>
<tr>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>String length</td>
<td>Returns <strong>True</strong> if a string length equals specified length. Otherwise, returns <strong>False</strong>.</td>
</tr>
<tr>
<td>Starts with</td>
<td>Returns <strong>True</strong> if a string starts will a specified prefix. Otherwise, returns <strong>False</strong>.</td>
</tr>
<tr>
<td>Ends with</td>
<td>Returns <strong>True</strong> if a string length equals specified length. Otherwise, returns <strong>False</strong>.</td>
</tr>
<tr>
<td>Is alphanumeric</td>
<td>Returns <strong>True</strong> if a string only contains numbers and letters. Otherwise, returns <strong>False</strong>.</td>
</tr>
<tr>
<td>Is alpha (letters)</td>
<td>Returns <strong>True</strong> if a string only contains letters. Otherwise, returns <strong>False</strong>.</td>
</tr>
<tr>
<td>Is digit</td>
<td>Returns <strong>True</strong> if a string only contains digits. Otherwise, returns <strong>False</strong>.</td>
</tr>
<tr>
<td>Is space</td>
<td>Returns <strong>True</strong> if a string only contains numbers and letters. Otherwise, returns <strong>False</strong>.</td>
</tr>
<tr>
<td>Is title</td>
<td>Returns <strong>True</strong> if a string contains any white spaces. Otherwise, returns <strong>False</strong>.</td>
</tr>
<tr>
<td>Is lowercase</td>
<td>Returns <strong>True</strong> if a string only contains lower case letters. Otherwise, returns <strong>False</strong>.</td>
</tr>
<tr>
<td>Is uppercase</td>
<td>Returns <strong>True</strong> if a string only contains upper case letters. Otherwise, returns <strong>False</strong>.</td>
</tr>
<tr>
<td>Is numeric</td>
<td>Returns <strong>True</strong> if a string only contains numbers. Otherwise, returns <strong>False</strong>.</td>
</tr>
<tr>
<td>Is decimal</td>
<td>Returns <strong>True</strong> if a string only contains decimal numbers. Otherwise, returns <strong>False</strong>.</td>
</tr>
</tbody>
</table>

**Analyze and Visualize**

Amazon SageMaker Data Wrangler includes built-in analyses that help you generate visualizations and data analyses in a few clicks. You can also create custom analyses using your own code.

You add an analysis to a dataframe by selecting a step in your data flow, and then choosing **Add analysis**. To access an analysis you've created, select the step that contains the analysis, and select the analysis.
All analyses are generated using 100,000 rows of your dataset.

You can add the following analysis to a dataframe:

- Data visualizations, including histograms and scatter plots.
- A quick summary of your dataset, including number of entries, minimum and maximum values (for numeric data), and most and least frequent categories (for categorical data).
- A quick model of the dataset, which can be used to generate an importance score for each feature.
- A target leakage report, which you can use to determine if one or more features are strongly correlated with your target feature.
- A custom visualization using your own code.

Use the following sections to learn more about these options.

**Histogram**

Use histograms to see the counts of feature values for a specific feature. You can inspect the relationships between features using the **Color by** option. For example, the following histogram charts the distribution of user ratings of the best-selling books on Amazon from 2009–2019, colored by genre.

You can use the **Facet by** feature to create histograms of one column, for each value in another column. For example, the following diagram shows histograms of user reviews of best-selling books on Amazon if faceted by year.
Use the **Scatter Plot** feature to inspect the relationship between features. To create a scatter plot, select a feature to plot on the **X axis** and the **Y axis**. Both of these columns must be numeric typed columns.

You can color scatter plots by an additional column. For example, the following example shows a scatter plot comparing the number of reviews against user ratings of top-selling books on Amazon between 2009 and 2019. The scatter plot is colored by book genre.
Additionally, you can facet scatter plots by features. For example, the following shows an example of the same review versus user rating scatter plot, faceted by year.
Table Summary

Use the **Table Summary** analysis to quickly summarize your data.

For columns with numerical data, including log and float data, a table summary reports the number of entries (count), minimum (min), maximum (max), mean, and standard deviation (stddev) for each column.

For columns with non-numerical data, including columns with string, Boolean, or date/time data, a table summary reports the number of entries (count), least frequent value (min), and most frequent value (max).

Quick Model

Use the **Quick Model** visualization to quickly evaluate your data and produce importance scores for each feature. A **feature importance score** score indicates how useful a feature is at predicting a target label. The feature importance score is between [0, 1] and a higher number indicates that the feature is more important to the whole dataset. On the top of the quick model chart, there is a model score. A classification problem shows an F1 score. A regression problem has a mean squared error (MSE) score.

When you create a quick model chart, you select a dataset you want evaluated, and a target label against which you want feature importance to be compared. Data Wrangler does the following:

- Infers the data types for the target label and each feature in the dataset selected.
- Determines the problem type. Based on the number of distinct values in the label column, Data Wrangler determines if this is a regression or classification problem type. Data Wrangler sets a
categorical threshold to 100. If there are more than 100 distinct values in the label column, Data Wrangler classifies it as a regression problem; otherwise, it is classified as a classification problem.

- Pre-process features and label data for training. The algorithm used requires encoding features to vector type and encoding labels to double type.
- Trains a random forest algorithm with 70% of data. Spark’s RandomForestRegressor is used to train a model for regression problems. The RandomForestClassifier is used to train a model for classification problems.
- Evaluates a random forest model with the remaining 30% of data. Data Wrangler evaluates classification models using an F1 score and evaluates regression models using an MSE score.
- Calculates feature importance for each feature using the Gini importance method.

Below is the user interface for the Quick Model feature.

![User Interface](image)

**Target Leakage**

Target leakage occurs when there is data in a machine learning training dataset that is strongly correlated with the target label, but is not available in real-world data. For example, you may have a column in your dataset that serves as a proxy for the column you want to predict with your model.

When you use the **Target Leakage** analysis, you specify the following:

- **Target:** This is the feature about which you want your ML model to be able to make predictions.
- **Problem type:** This is the ML problem type on which you are working. Problem type can either be **classification** or **regression**.
- (Optional) **Max features:** This is the maximum number of features to present in the visualization, which shows features ranked by their risk of being target leakage.

For classification, the Target Leakage analysis uses the area under the receiver operating characteristic, or AUC - ROC curve for each column, up to **Max features**. For regression, it uses a coefficient of determination, or R2 metric.

The AUC - ROC curve provides a predictive metric, computed individually for each column using cross-validation, on a sample of up to around 1000 rows. A score of 1 indicates perfect predictive abilities, which often indicates target leakage. A score of 0.5 or lower indicates that the information on the
column could not provide, on its own, any useful information towards predicting the target. Although it can happen that a column is uninformative on its own but is useful in predicting the target when used in tandem with other features, a low score could indicate the feature is redundant.

For example, the following image shows a target leakage report for a diabetes classification problem, that is, predicting if a person has diabetes or not. An AUC - ROC curve is used to calculate the predictive ability of five features, and all are determined to be safe from target leakage.

Bias Report

You can use the bias report in Data Wrangler to uncover potential biases in your data. To generate a bias report, you must specify the target column, or Label, that you want to predict and a Facet, or the column that you want to inspect for biases.

**Label** – The feature about which you want a model to make predictions. For example, if you are predicting customer conversion, you may select a column containing data on whether or not a customer has placed an order. You must also specify whether this feature is a label or a threshold. If you specify a label, you must specify what a positive outcome looks like in your data. In the customer conversion example, a positive outcome may be a 1 in the orders column, representing the positive outcome of a customer placing an order within the last three months. If you specify a threshold, you must specify a lower bound defining a positive outcome. For example, if your customer orders columns contains the number of orders placed in the last year, you may want to specify 1.

**Facet** – The column that you want to inspect for biases. For example, if you are trying to predict customer conversion, your facet may be the age of the customer. You may choose this facet because you believe that your data is biased toward a certain age group. You must identify whether the facet is measured as a value or threshold. For example, if you wanted to inspect one or more specific ages, you select Value and specify those ages. If you want to look at an age group, you select Threshold and specify the threshold of ages you want to inspect.

After you select your feature and label, you select the types of bias metrics you want to calculate.

To learn more, see Generate reports for bias in pre-training data.
Create Custom Visualizations

Use the **Code** tab to create custom visualizations. Your dataset, after undergoing the most recently added transformation, is available as a **Pandas DataFrame** in this code block via the `df` variable.

You must provide an output variable, `chart`, to store an **Altair** output chart. For example, the following is an example of a code block that can be used to generate a custom histogram using the titanic dataset.

```python
import altair as alt
df = df.iloc[:30]
df = df.rename(columns={"Age": "value"})
df = df.assign(count=df.groupby('value').value.transform('count'))
df = df["value", "count"]
base = alt.Chart(df)
base = base.mark_bar().encode(x=alt.X('value', bin=True, axis=None), y=alt.Y('count'))
base = base.mark_rule(color='red').encode(
    x=alt.value("mean(value)"),
    size=alt.value(5))
chart = bar + rule
```

To create a custom visualization:

1. In the **Visualize** area of Data Wrangler, choose the **Code** tab.
2. Give your visualization a **Name**.
3. Enter your code in the code box.
4. Choose **Preview** to preview your visualization.
5. Choose **Add Visualization** to add your visualization.

Export

When you create a Data Wrangler data flow, you can choose from four export options to easily integrate that data flow into your data processing pipeline. Data Wrangler offers export options to SageMaker Data Wrangler Job, Pipeline, **Python code** and **Feature Store**.
The following options create a Jupyter Notebook to execute your data flow and integrate with the respective SageMaker feature.

- **Data Wrangler Job**
- **Pipeline**
- **Feature Store**

For these options, you choose one or more steps in your data flow to export. When you select a step, all downstream steps are also exported. For example, if you have defined seven consecutive steps in your data flow, and you choose to export the 7th step, code is exported to execute steps 1 through 7. The Jupyter Notebooks automatically open when you select one of these export options, and you can run the notebook directly in Studio using a **Python 3 (Data science)** kernel.

If you select **Python code**, a Python file is created containing all steps in your data flow.

When you choose an export option that creates a Jupyter Notebook and you execute the notebook, it exports your data flow (.flow file) to the default SageMaker S3 bucket for the AWS Region in which the data flow was created, under the prefix `data_wrangler_flows`. Default S3 buckets have the following naming convention: `sagemaker-region-account number`. For example, if your account number is 111122223333 and you are using Studio in us-east-1, your imported datasets are stored in `sagemaker-us-east-1-111122223333`. In this example, your .flow files created in us-east-1 are stored in `s3://sagemaker-region-account number/data_wrangler_flows/`.

**Important**

If you do not use the IAM managed policy, `AmazonSageMakerFullAccess`, to grant AWS roles permission to use Data Wrangler, make sure you grant these roles permission to access this bucket. See Security and Permissions (p. 573) for an example IAM policy you can use to grant these permissions.

Use the following procedure to export a data flow. Use the sections on this page to learn more about each export option.

**To export your Data Wrangler flow:**

1. Navigate to the **Export** tab.
2. Select the last step in your Data Wrangler flow.
3. Choose **Export step**.
4. Select the export option you want.
Export to a Data Wrangler Job

If you export a Data Wrangler job Jupyter Notebook, we recommend that you select **Python 3 (Data Science)** for the **Kernel** and run it directly in Studio to execute your data flow and process your data. Follow the instructions in the notebook to launch your Data Wrangler job.

Data Wrangler jobs use processing jobs to process your data. You can run these processing jobs using ml.m5.4xlarge, ml.m5.12xlarge, and ml.m5.24xlarge instances and support one instance count. By default, the notebook that is exported from Data Wrangler sets the following instance_count and instance_type:

```
instance_count = 1
instance_type = "ml.m5.xlarge"
```

You can monitor your Data Wrangler job status in the **SageMaker console** in the **Processing** tab. The processing job is named `data-wrangler-flow-processing-<flow_id>`. The `<flow_id>` is defined in the notebook using the day and time the notebook is executed.

Additionally, you can monitor your Data Wrangler job using Amazon CloudWatch. For additional information, see [Monitor Amazon SageMaker Processing Jobs with CloudWatch Logs and Metrics](#).

Export to SageMaker Pipelines

When you want to build and deploy large-scale machine learning workflows, you can use SageMaker ML Pipelines to create end-to-end workflows that manage and deploy SageMaker jobs. ML Pipelines allows you to build workflows that manage your SageMaker data preparation, model training, and model deployment jobs. Because of this, you can take advantage of the first-party algorithms that SageMaker offers. To learn more about SageMaker Pipelines, see [SageMaker Pipelines](#).

When you export one or more steps from your data flow to SageMaker Pipelines, Data Wrangler creates a Jupyter Notebook that you can use to define, instantiate, run, and manage a pipeline.

**Use A Jupyter Notebook to Create a Pipeline**

The Jupyter Notebook that Data Wrangler produces can be used to define a pipeline. The pipeline includes a data processing step that is defined by your data flow.

You can add additional steps to your pipeline by adding steps to the `steps` list in the following code in the notebook:

```
pipeline = Pipeline(
    name=pipeline_name,
    parameters=[[instance_type, instance_count],
                [step_process]],  #Add more steps to this list to run in your Pipeline
    steps=[step_process],
)
```

To learn more about defining pipelines, see [Define SageMaker Pipeline](#).

Export to Python Code

To export all steps in your data flow to a Python file that you can manually integrate into any data processing workflow, choose the **Export to Code** option.

You may need to configure the Python script to make it runnable. For example, you may need to modify your Spark environment, and ensure you are running the script from an environment that has permission to access AWS resources.
Export to the SageMaker Feature Store

The SageMaker Feature Store can be used to create, share, and manage curated data for machine learning (ML) development. You can configure an online and offline feature store to be a centralized store for features and associated metadata so features can be easily discovered and reused. To learn more about the Data Wrangler feature store, see Amazon SageMaker Feature Store.

Use A Jupyter Notebook to Add Features to a Feature Store

The Jupyter Notebook that SageMaker produces can be used to process your dataset using a SageMaker Data Wrangler job, and then ingest the data into an online and offline feature store.

Important
The IAM role you use to run this notebook must have the following AWS managed policies attached: AmazonSageMakerFullAccess and AmazonSageMakerFeatureStoreAccess.

You only need to enable one online or offline feature store when you create a feature group. Optionally, you can enable both. To disable online store creation, set EnableOnlineStore to False:

```python
# Online Store Configuration
online_store_config = {
    "EnableOnlineStore": False
}
```

The notebook uses the column names and types of the dataframe you export to create a feature group schema, which is used to create a feature group. A feature group is a group of features defined in the feature store to describe a record. The feature group defines the schema and features contained in the feature group. A feature group definition is composed of a list of features, a record identifier feature name, an event time feature name, and configurations for its online store and offline store.

Each feature in a feature group can have one of the following types: String, Fractional, or Integral. If a column in your exported dataframe is not one of these types, it defaults to String.

The following is an example of a feature group schema.

```python
column_schema = [
    {
        "name": "Height",
        "type": "long"
    },
    {
        "name": "Input",
        "type": "string"
    },
    {
        "name": "Output",
        "type": "string"
    },
    {
        "name": "Sum",
        "type": "string"
    },
    {
        "name": "Time",
        "type": "string"
    }
]
```

Additionally, you must specify a record identifier name and event time feature name:
The record identifier name is the name of the feature whose value uniquely identifies a record defined in the feature store. Only the latest record per identifier value is stored in the online store. The record identifier feature name must be one of feature definitions' names.

- The event time feature name is the name of the feature that stores the EventTime of a record in a feature group. An EventTime is a point in time when a new event occurs that corresponds to the creation or update of a record in a feature. All records in the feature group must have a corresponding EventTime.

The notebook uses these configurations to create a feature group, process your data at scale, and then ingest the processed data into your online and offline feature stores. To learn more, see Data Sources and Ingestion.

Security and Permissions

When you query data from Athena or Amazon Redshift, the queried dataset is automatically stored in the default SageMaker S3 bucket for the AWS Region in which you are using Studio. Additionally, when you export a Jupyter Notebook from Amazon SageMaker Data Wrangler and execute it, your data flows, or .flow files, are saved to the same default bucket, under the prefix data_wrangler_flows.

For high-level security needs, you can configure a bucket policy that restricts the AWS roles that have access to this default SageMaker S3 bucket. Use the following section to add this type of policy to an S3 bucket. To follow the instructions on this page, use the AWS Command Line Interface (AWS CLI). To learn how, see Configuring the AWS CLI in the IAM User Guide.

Additionally, you need to grant each IAM role that uses Data Wrangler permissions to access required resources. If you do not require granular permissions for the IAM role you use to access Data Wrangler, you can add the IAM managed policy, AmazonSageMakerFullAccess, to an IAM role that you use to create your Studio user. This policy grants you full permission to use Data Wrangler. If you require more granular permissions, refer to the section, Grant an IAM Role Permission to Use Data Wrangler (p. 574).

Add a Bucket Policy To Restrict Access to Datasets Imported to Data Wrangler

You can add a policy to the S3 bucket that contains your Data Wrangler resources using an Amazon S3 bucket policy. Resources that Data Wrangler uploads to your default SageMaker S3 bucket in the AWS Region you are using Studio in include the following:

- Queried Amazon Redshift results. These are stored under the redshift/prefix.
- Queried Athena results. These are stored under the athena/prefix.
- The .flow files uploaded to Amazon S3 when you execute an exported Jupyter Notebook Data Wrangler produces. These are stored under the data_wrangler_flows/prefix.

Use the following procedure to create an S3 bucket policy that you can add to restrict IAM role access to that bucket. To learn how to add a policy to an S3 bucket, see How do I add an S3 Bucket policy?.

**To set up a bucket policy on the S3 bucket that stores your Data Wrangler resources:**

1. Configure one or more IAM roles that you want to be able to access Data Wrangler.
2. Open a command prompt or shell. For each role that you create, replace role-name with the name of the role and run the following:

   ```bash
   $ aws iam get-role --role-name role-name
   ```

   In the response, you'll see a RoleId string which begins will AROA. Copy this string.
3. Add the following policy to the SageMaker default bucket in the AWS Region in which you are using Data Wrangler. Replace region with the AWS Region in which the bucket is located, and account-id with your AWS account ID. Replace userIds starting with AROAEXAMPLEID with the IDs of an AWS roles to which you want to grant permission to use Data Wrangler.

```json
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Deny",
      "Principal": "*",
      "Action": "s3:*",
      "Resource": [
        "arn:aws:s3:::sagemaker-region-account-id/data_wrangler_flows/",
        "arn:aws:s3:::sagemaker-region-account-id/data_wrangler_flows/**",
        "arn:aws:s3:::sagemaker-region-account-id/athena",
        "arn:aws:s3:::sagemaker-region-account-id/athena/**",
        "arn:aws:s3:::sagemaker-region-account-id/redshift",
        "arn:aws:s3:::sagemaker-region-account-id/redshift/**"
      ],
      "Condition": {
        "StringNotLike": {
          "aws:userId": [
            "AROAEXAMPLEID_1:*",
            "AROAEXAMPLEID_2:*"
          ]
        }
      }
    }
  ]
}
```

**Grant an IAM Role Permission to Use Data Wrangler**

You can grant an IAM role permission to use Data Wrangler with the general IAM managed policy, `AmazonSageMakerFullAccess`. This is a general policy that includes permissions required to use all SageMaker services. This policy grants an IAM role full access to Data Wrangler. You should be aware of the following when using `AmazonSageMakerFullAccess` to grant access to Data Wrangler:

- If you import data from Amazon Redshift, the **Database User** name must have the prefix `sagemaker_access`.
- This managed policy only grants permission to access buckets with one of the following in the name: SageMaker, Sagemaker, sagemaker, or aws-glue. If want to use Data Wrangler to import from an S3 bucket without these phrases in the name, refer to the last section on this page to learn how to grant permission to an IAM entity to access your S3 buckets.

If you have high-security needs, you can attach the policies in this section to an IAM entity to grant permissions required to use Data Wrangler.

If you have datasets in Amazon Redshift or Athena that an IAM role needs to import from Data Wrangler, you must add a policy to that entity to access these resources. The following policies are the most restrictive policies you can use to give an IAM role permission to import data from Amazon Redshift and Athena.

To learn how to attach a custom policy to an IAM role, refer to Managing IAM policies in the IAM User Guide.

**Policy example to grant access to an Athena dataset import**
The following policy assumes that the IAM role has permission to access the underlying S3 bucket where data is stored through a separate IAM policy.

```json
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "athena:ListDataCatalogs",
                "athena:ListDatabases",
                "athena:ListTableMetadata",
                "athena:GetQueryExecution",
                "athena:GetQueryResults",
                "athena:StartQueryExecution",
                "athena:StopQueryExecution"
            ],
            "Resource": ["*"]
        },
        {
            "Effect": "Allow",
            "Action": ["glue:CreateTable"],
            "Resource": [
                "arn:aws:glue:*:*:table/*/sagemaker_tmp_*",
                "arn:aws:glue:*:*:table/sagemaker_featurestore/*",
                "arn:aws:glue:*:*:catalog",
                "arn:aws:glue:*:*:database/*"
            ]
        },
        {
            "Effect": "Allow",
            "Action": ["glue:DeleteTable"],
            "Resource": [
                "arn:aws:glue:*:*:table/*/sagemaker_tmp_*",
                "arn:aws:glue:*:*:catalog",
                "arn:aws:glue:*:*:database/*"
            ]
        },
        {
            "Effect": "Allow",
            "Action": ["glue:GetDatabases", "glue:GetTable", "glue:GetTables"],
            "Resource": [
                "arn:aws:glue:*:*:table/*/",
                "arn:aws:glue:*:*:catalog",
                "arn:aws:glue:*:*:database/*"
            ]
        },
        {
            "Effect": "Allow",
            "Action": ["glue:CreateDatabase", "glue:GetDatabase"],
            "Resource": ["arn:aws:glue:*:*:catalog"]
        }
    ]
}
```
Policy example to grant access to an Amazon Redshift dataset import

The following policy grants permission to set up an Amazon Redshift connection to Data Wrangler using database users that have the prefix `sagemaker_access` in the name. To grant permission to connect using additional database users, add additional entries under "Resources" in the following policy. The following policy assumes that the IAM role has permission to access the underlying S3 bucket where data is stored through a separate IAM policy, if applicable.

```json
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "redshift-data:ExecuteStatement",
        "redshift-data:DescribeStatement",
        "redshift-data:CancelStatement",
        "redshift-data:GetStatementResult",
        "redshift-data:ListSchemas",
        "redshift-data:ListTables"
      ],
      "Resource": ["*"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "redshift:GetClusterCredentials"
      ],
      "Resource": [
        "arn:aws:redshift:*:*:dbuser:*/sagemaker_access*",
        "arn:aws:redshift:*:*:dbname:*"
      ]
    }
  ]
}
```

Policy to grant access to an S3 bucket

If your dataset is stored in Amazon S3, you can grant an IAM role permission to access this bucket with a policy similar to the following. This example grants programmatic read-write access to the bucket named `test`:

```json
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": ["s3:ListBucket"],
      "Resource": ["arn:aws:s3:::test"]
    },
    {
```
To import data from Athena and Amazon Redshift, you must grant an IAM role permission to access the following prefixes under the default Amazon S3 bucket in the AWS Region Data Wrangler is being used in: \texttt{athena/}, \texttt{redshift/}. If a default Amazon S3 bucket does not already exist in the AWS Region, you must also give the IAM role permission to create a bucket in this region.

Additionally, if you want the IAM role to be able to use the SageMaker Feature Store, SageMaker Pipeline, and Data Wrangler job export options, you must grant access to the prefix \texttt{data\_wrangler\_flows/} in this bucket.

Data Wrangler uses the \texttt{athena/} and \texttt{redshift/} prefixes to store preview files and imported datasets. To learn more, see Imported Data Storage (p. 542).

Data Wrangler uses the \texttt{data\_wrangler\_flows/} prefix to store .flow files when you run a Jupyter Notebook exported from Data Wrangler. To learn more, see Export (p. 569).

Use a policy similar to the following to grant the permissions described above:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "s3:GetObject",
                "s3:PutObject"
            ],
            "Resource": [
                "arn:aws:s3:::sagemaker-region-account-id/data\_wrangler\_flows/",
                "arn:aws:s3:::sagemaker-region-account-id/data\_wrangler\_flows/**",
                "arn:aws:s3:::sagemaker-region-account-id/athena",
                "arn:aws:s3:::sagemaker-region-account-id/athena/**",
                "arn:aws:s3:::sagemaker-region-account-id/redshift",
                "arn:aws:s3:::sagemaker-region-account-id/redshift/**"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "s3:CreateBucket",
                "s3:ListBucket"
            ],
            "Resource": "arn:aws:s3:::sagemaker-region-account-id"
        },
        {
            "Effect": "Allow",
            "Action": [
                "s3:ListAllMyBuckets",
                "s3:GetBucketLocation"
            ],
            "Resource": "*"
        }
    ]
}
```
Policy example to grant access to use SageMaker Studio

Use a policy like the following to create an IAM execution role that can be used to set up a Studio instance.

```json
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "sagemaker:CreatePresignedDomainUrl",
        "sagemaker:DescribeDomain",
        "sagemaker:ListDomains",
        "sagemaker:DescribeUserProfile",
        "sagemaker:ListUserProfiles",
        "sagemaker:*App",
        "sagemaker:ListApps"
      ],
      "Resource": "*"
    }
  ]
}
```

Troubleshoot

If an issue arises when using Amazon SageMaker Data Wrangler, we recommend you do the following:

- If an error message is provided, read the message and resolve the issue it reports if possible.
- Make sure the IAM role of your Studio user has the required permissions to perform the action. Reference Security and Permissions (p. 573).
- If the issue occurs when you are trying to import from another AWS service, such as Amazon Redshift or Athena, make sure that you have configured the necessary permissions and resources to perform the data import. Reference Import (p. 536).

As a last resort, you can try restarting the kernel on which Data Wrangler is running.

1. Save and exit out of the .flow file for which you want to restart the kernel.
2. Select the Running Terminals and Kernels icon, as shown in the following image.
3. Select the **Stop** icon to the right of the `.flow` file for which you want to terminate the kernel, as shown in the following image.
4. Refresh the browser.
5. Reopen the .flow file on which you were working.

Process Data and Evaluate Models

To analyze data and evaluate machine learning models on Amazon SageMaker, use Amazon SageMaker Processing. With Processing, you can use a simplified, managed experience on SageMaker to run your data processing workloads, such as feature engineering, data validation, model evaluation, and model interpretation. You can also use the Amazon SageMaker Processing APIs during the experimentation phase and after the code is deployed in production to evaluate performance.
The preceding diagram shows how Amazon SageMaker spins up a Processing job. Amazon SageMaker takes your script, copies your data from Amazon Simple Storage Service (Amazon S3), and then pulls a processing container. The processing container image can either be an Amazon SageMaker built-in image or a custom image that you provide. The underlying infrastructure for a Processing job is fully managed by Amazon SageMaker. Cluster resources are provisioned for the duration of your job, and cleaned up when a job completes. The output of the Processing job is stored in the Amazon S3 bucket you specified.

**Note**
Your data must be stored in an Amazon S3 bucket.

**Topics**

- Use Amazon SageMaker Processing Sample Notebooks (p. 581)
- Monitor Amazon SageMaker Processing Jobs with CloudWatch Logs and Metrics (p. 582)
- Data Processing with Apache Spark (p. 582)
- Data Processing with scikit-learn (p. 583)
- Detect Pretraining Data Bias (p. 583)
- Use Your Own Processing Code (p. 599)

**Use Amazon SageMaker Processing Sample Notebooks**

We provide two sample Jupyter notebooks that show how to perform data preprocessing, model evaluation, or both.

For a sample notebook that shows how to run scikit-learn scripts to perform data preprocessing and model training and evaluation with the SageMaker Python SDK for Processing, see scikit-learn Processing. This notebook also shows how to use your own custom container to run processing workloads with your Python libraries and other specific dependencies.

For a sample notebook that shows how to use Amazon SageMaker Processing to perform distributed data preprocessing with Spark, see Distributed Processing (Spark). This notebook also shows how to train a regression model using XGBoost on the preprocessed dataset.

For instructions on how to create and access Jupyter notebook instances that you can use to run these samples in SageMaker, see Use Amazon SageMaker Notebook Instances (p. 124). After you have created a notebook instance and opened it, choose the SageMaker Examples tab to see a list of all the SageMaker samples. To open a notebook, choose its Use tab and choose Create copy.
Monitor Amazon SageMaker Processing Jobs with CloudWatch Logs and Metrics

Amazon SageMaker Processing provides Amazon CloudWatch logs and metrics to monitor processing jobs. CloudWatch provides CPU, GPU, memory, GPU memory, and disk metrics, and event logging. For more information, see Monitor Amazon SageMaker with Amazon CloudWatch (p. 1704) and Log Amazon SageMaker Events with Amazon CloudWatch (p. 1712).

Data Processing with Apache Spark

Apache Spark is a unified analytics engine for large-scale data processing. Amazon SageMaker provides prebuilt Docker images that include Apache Spark and other dependencies needed to run distributed data processing jobs. With the Amazon SageMaker Python SDK, you can easily apply data transformations and extract features (feature engineering) using the Spark framework. For information about using the SageMaker Python SDK to run Spark processing jobs, see Data Processing with Spark in the Amazon SageMaker Python SDK.

A code repository that contains the source code and Dockerfiles for the Spark images is available on GitHub.

Running a Spark Processing Job

You can use the `sagemaker.spark.PySparkProcessor` or `sagemaker.spark.SparkJarProcessor` class to run your Spark application inside of a processing job.

The following code example shows how to run a processing job that invokes your PySpark script `preprocess.py`.

```python
from sagemaker.spark.processing import PySparkProcessor

spark_processor = PySparkProcessor(
    base_job_name="spark-preprocessor",
    framework_version="2.4",
    role=role,
    instance_count=2,
    instance_type="ml.m5.xlarge",
    max_runtime_in_seconds=1200,
)

spark_processor.run(
    submit_app="preprocess.py",
    arguments=['s3_input_bucket', bucket,
              's3_input_key_prefix', input_prefix,
              's3_output_bucket', bucket,
              's3_output_key_prefix', output_prefix]
)
```

For an in-depth look, see the Distributed Data Processing with Apache Spark and SageMaker Processing example notebook.

If you are not using the Amazon SageMaker Python SDK and one of its Processor classes to retrieve the pre-built images, you can retrieve these images yourself. The SageMaker prebuilt Docker images are stored in Amazon Elastic Container Registry (Amazon ECR). For a complete list of the available pre-built Docker images, see the available images document.

To learn more about using the SageMaker Python SDK with Processing containers, see Amazon SageMaker Python SDK.
Data Processing with scikit-learn

For a sample notebook that shows how to run scikit-learn scripts using a Docker image provided and maintained by SageMaker to preprocess data and evaluate models, see scikit-learn Processing. To use this notebook, you need to install the SageMaker Python SDK for Processing.

This notebook runs a processing job using SKLearnProcessor class from the the SageMaker Python SDK to run a scikit-learn script that you provide. The script preprocesses data, trains a model using a SageMaker training job, and then runs a processing job to evaluate the trained model. The processing job estimates how the model is expected to perform in production.

To learn more about using the SageMaker Python SDK with Processing containers, see the SageMaker Python SDK.

The following code example shows how the notebook uses SKLearnProcessor to run your own scikit-learn script using a Docker image provided and maintained by SageMaker, instead of your own Docker image.

```python
from sagemaker.sklearn.processing import SKLearnProcessor
from sagemaker.processing import ProcessingInput, ProcessingOutput

sklearn_processor = SKLearnProcessor(framework_version='0.20.0',
                                  role=role,
                                  instance_type='ml.m5.xlarge',
                                  instance_count=1)

sklearn_processor.run(code='preprocessing.py',
                      inputs=[ProcessingInput(
                              source='s3://path/to/my/input-data.csv',
                              destination='/opt/ml/processing/input')],
                      outputs=[ProcessingOutput(source='/opt/ml/processing/output/train'),
                               ProcessingOutput(source='/opt/ml/processing/output/validation'),
                               ProcessingOutput(source='/opt/ml/processing/output/test')]
)
```

To process data in parallel using Scikit-Learn on Amazon SageMaker Processing, you can shard input objects by S3 key by setting `s3_data_distribution_type='ShardedByS3Key'` inside a ProcessingInput so that each instance receives about the same number of input objects.

Detect Pretraining Data Bias

Algorithmic bias, discrimination, fairness, and related topics have been studied across disciplines such as law, policy, and computer science. A computer system might be considered biased if it discriminates against certain individuals or groups of individuals. The machine learning models powering these applications learn from data and this data could reflect disparities or other inherent biases. For example, the training data may not have sufficient representation of various demographic groups or may contain biased labels. The machine learning models trained on datasets that exhibit these biases could end up learning them and then reproduce or even exacerbate those biases in their predictions. The field of machine learning provides an opportunity to address biases by detecting them and measuring them at each stage of the ML lifecycle. You can use Amazon SageMaker Clarify to determine whether data used for training models encodes any bias.

Bias can be measured before training and after training, and monitored against baselines after deploying models to endpoints for inference. Pretraining bias metrics are designed to detect and measure bias in the raw data before it is used to train a model. The metrics used are model-agnostic because they do not depend on any model outputs. However, there are different concepts of fairness that require distinct measures of bias. Amazon SageMaker Clarify provides bias metrics to quantify various fairness criteria.
For additional information about bias metrics, see Fairness Measures for Machine Learning in Finance.

Amazon SageMaker Clarify Terms for Bias and Fairness

SageMaker Clarify uses the following terminology to discuss bias and fairness.

**Feature**
An individual measurable property or characteristic of a phenomenon being observed, contained in a column for tabular data.

**Label**
Feature that is the target for training a machine learning model. Referred to as the *observed label* or *observed outcome*.

**Predicted label**
The label as predicted by the model. Also referred to as the *predicted outcome*.

**Sample**
An observed entity described by feature values and label value, contained in a row for tabular data.

**Dataset**
A collection of samples.

**Bias**
An imbalance in the training data or the prediction behavior of the model across different groups, such as age or income bracket. Biases can result from the data or algorithm used to train your model. For instance, if an ML model is trained primarily on data from middle-aged individuals, it may be less accurate when making predictions involving younger and older people.

**Bias metric**
A function that returns numerical values indicating the level of a potential bias.

**Bias report**
A collection of bias metrics for a given dataset, or a combination of a dataset and a model.

**Positive label values**
Label values that are favorable to a demographic group observed in a sample. In other words, designates a sample as having a *positive result*.

**Negative label values**
Label values that are unfavorable to a demographic group observed in a sample. In other words, designates a sample as having a *negative result*.

**Group variable**
Categorical column of the dataset that is used to form subgroups for the measurement of Conditional Demographic Disparity (CDD). Required only for this metric with regards to Simpson's paradox.

**Facet**
A column or feature that contains the attributes with respect to which bias is measured.

**Facet value**
The feature values of attributes that bias might favor or disfavor.

**Predicted probability**
The probability, as predicted by the model, of a sample having a positive or negative outcome.
Sample Notebooks

Amazon SageMaker Clarify provides the following sample notebook for bias detection:

- Explainability and bias detection with Amazon SageMaker Clarify – Use SageMaker Clarify to create a processing job for the detecting bias and explaining model predictions with feature attributions.

This notebook has been verified to run in Amazon SageMaker Studio only. If you need instructions on how to open a notebook in Amazon SageMaker Studio, see Create or Open an Amazon SageMaker Studio Notebook (p. 87). If you're prompted to choose a kernel, choose Python 3 (Data Science).

Topics
- Measure Pretraining Bias (p. 585)
- Generate Reports for Bias in Pretraining Data in SageMaker Studio (p. 594)

Measure Pretraining Bias

Measuring bias in ML models is a first step to mitigating bias. Each measure of bias corresponds to a different notion of fairness. Even considering simple concepts of fairness leads to many different measures applicable in various contexts. For example, consider fairness with respect to age, and, for simplicity, that middle-aged and rest of the age groups are the two relevant demographics, referred to as facets. In the case of an ML model for lending, we may want small business loans to be issued to equal numbers of both demographics. Or, when processing job applicants, we may want to see equal numbers of members of each demographic hired. However, this approach may assume that equal numbers of both age groups apply to these jobs, so we may want to condition on the number that apply. Further, we may want to consider not whether equal numbers apply, but whether we have equal numbers of qualified applicants. Or, we may consider fairness to be an equal acceptance rate of qualified applicants across both age demographics, or, an equal rejection rate of applicants, or both. You might use datasets with different proportions of data on the attributes of interest. This imbalance can confound the bias measures you choose. The models might be more accurate in classifying one facet than in the other. Thus, you need to choose bias metrics that are conceptually appropriate for the application and the situation.

We use the following notation to discuss the bias metrics. The conceptual model described here is for binary classification, where events are labeled as having only two possible outcomes in their sample space, referred to as positive (with value 1) and negative (with value 0). This framework is usually extensible to multiclassification in a straightforward way or to cases involving continuous valued outcomes when needed. In the binary classification case, positive and negative labels are assigned to outcomes recorded in a raw dataset for a favored facet $a$ and for a disfavored facet $d$. These labels $y$ are referred to as observed labels to distinguish them from the predicted labels $y'$ that are assigned by a machine learning model during the training or inferences stages of the ML lifecycle. These labels are used to define probability distributions $P_a(y)$ and $P_d(y)$ for their respective facet outcomes.

- labels:
  - $y$ represents the $n$ observed labels for event outcomes in a training dataset.
  - $y'$ represents the predicted labels for the $n$ observed labels in the dataset by a trained model.

- outcomes:
  - A positive outcome (with value 1) for a sample, such as an application acceptance.
    - $n^{(1)}$ is the number of observed labels for positive outcomes (acceptances).
    - $n^{(1)}$ is the number of predicted labels for positive outcomes (acceptances).
  - A negative outcome (with value 0) for a sample, such as an application rejection.
    - $n^{(0)}$ is the number of observed labels for negative outcomes (rejections).
    - $n^{(0)}$ is the number of predicted labels for negative outcomes (rejections).

- facet values:
Detect Pretraining Data Bias

- facet $a$ – The feature value that defines a demographic that bias favors.
  - $n_a$ is the number of observed labels for the favored facet value: $n_a = n_a^{(1)} + n_a^{(0)}$ the sum of the positive and negative observed labels for the value facet $a$.
  - $n'_a$ is the number of predicted labels for the favored facet value: $n'_a = n'_a^{(1)} + n'_a^{(0)}$ the sum of the positive and negative predicted outcome labels for the facet value $a$. Note that $n'_a = n_a$.
- facet $d$ – The feature value that defines a demographic that bias disfavors.
  - $n_d$ is the number of observed labels for the disfavored facet value: $n_d = n_d^{(1)} + n_d^{(0)}$ the sum of the positive and negative observed labels for the facet value $d$.
  - $n'_d$ is the number of predicted labels for the disfavored facet value: $n'_d = n'_d^{(1)} + n'_d^{(0)}$ the sum of the positive and negative predicted labels for the facet value $d$. Note that $n'_d = n_d$.

- probability distributions for outcomes of the labeled facet data outcomes:
  - $P_a(y)$ is the probability distribution of the observed labels for facet $a$. For binary labeled data, this distribution is given by the ratio of the number of samples in facet $a$ labeled with positive outcomes to the total number, $P_a(y^1) = n_a^{(1)}/n_a$ and the ratio of the number of samples with negative outcomes to the total number, $P_a(y^0) = n_a^{(0)}/n_a$.
  - $P_d(y)$ is the probability distribution of the observed labels for facet $d$. For binary labeled data, this distribution is given by the number of samples in facet $d$ labeled with positive outcomes to the total number, $P_d(y^1) = n_d^{(1)}/n_d$ and the ratio of the number of samples with negative outcomes to the total number, $P_d(y^0) = n_d^{(0)}/n_d$.

Models trained on data biased by demographic disparities might learn and even exacerbate them. To identify bias in the data before expending resources to train models on it, SageMaker Clarify provides data bias metrics that you can compute on raw datasets before training. All of the pretraining metrics are model-agnostic because they do not depend on model outputs and so are valid for any model. The first bias metric examines facet imbalance, but not outcomes. It determines the extent to which the amount of training data is representative across different facets, as desired for the application. The remaining bias metrics compare the distribution of outcome labels in various ways for facets $a$ and $d$ in the data. The metrics that range over negative values can detect negative bias. The following table contains a cheat sheet for quick guidance and links to the pretraining bias metrics.

### Pretraining Bias Metrics

<table>
<thead>
<tr>
<th>Bias metric</th>
<th>Description</th>
<th>Example question</th>
<th>Interpreting metric values</th>
</tr>
</thead>
</table>
| **Class Imbalance (CI) (p. 589)** | Measures the imbalance in the number of members between different facet values. | Could there be age-based biases due to not having enough data for the demographic outside a middle-aged facet? | Normalized range: [-1, +1] Interpretation:  
  - Positive values indicate the facet $a$ has more training samples in the dataset.  
  - Values near zero indicate the facets are balanced in the number of training samples in the dataset.  
  - Negative values indicate the facet $d$ has more training samples than the data available for other facets. |
<table>
<thead>
<tr>
<th>Bias metric</th>
<th>Description</th>
<th>Example question</th>
<th>Interpreting metric values</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Difference in Proportions of Labels (DPL) (p. 590)</strong></td>
<td>Measures the imbalance of positive outcomes between different facet values.</td>
<td>Could there be age-based biases in ML predictions due to biased labeling of facet values in the data?</td>
<td>Range for normalized binary &amp; multicategory facet labels: [-1, +1] Range for continuous labels: (-∞, +∞)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Interpretation:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Positive values indicate facet a has a higher proportion of positive outcomes.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Values near zero indicate a more equal proportion of positive outcomes between facets.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Negative values indicate facet d has a higher proportion of positive outcomes.</td>
</tr>
<tr>
<td><strong>Kullback-Leibler Divergence (KL) (p. 590)</strong></td>
<td>Measures how much the outcome distributions of different facets diverge from each other entropically.</td>
<td>How different are the distributions for loan application outcomes for different demographic groups?</td>
<td>Range for binary, multicategory, continuous: [0, +∞)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Interpretation:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Values near zero indicate the labels are similarly distributed.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Positive values indicate the label distributions diverge, the more positive the larger the divergence.</td>
</tr>
<tr>
<td><strong>Jensen-Shannon Divergence (JS) (p. 591)</strong></td>
<td>Measures how much the outcome distributions of different facets diverge from each other entropically.</td>
<td>How different are the distributions for loan application outcomes for different demographic groups?</td>
<td>Range for binary, multicategory, continuous: [0, +∞)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Interpretation:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Values near zero indicate the labels are similarly distributed.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Positive values indicate the label distributions diverge, the more positive the larger the divergence.</td>
</tr>
<tr>
<td>Bias metric</td>
<td>Description</td>
<td>Example question</td>
<td>Interpreting metric values</td>
</tr>
<tr>
<td>-----------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>----------------------------------------------------------------------------------</td>
<td>------------------------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| $L_p$-norm (LP) (p. 591)                | Measures a p-norm difference between distinct demographic distributions of the outcomes associated with different facets in a dataset. | How different are the distributions for loan application outcomes for different demographics? | Range for binary, multicategory, continuous: $[0, +\infty)$<br>Interpretation:  
  - Values near zero indicate the labels are similarly distributed.  
  - Positive values indicate the label distributions diverge, the more positive the larger the divergence. |
| Total Variation Distance (TVD) (p. 592) | Measures half of the $L_1$-norm difference between distinct demographic distributions of the outcomes associated with different facets in a dataset. | How different are the distributions for loan application outcomes for different demographics? | Range for binary, multicategory, and continuous outcomes: $[0, +\infty)$<br>Interpretation:  
  - Values near zero indicates the labels are similarly distributed.  
  - Positive values indicates the label distributions diverge, the more positive the larger the divergence. |
| Kolmogorov-Smirnov (KS) (p. 592)       | Measures maximum divergence between outcomes in distributions for different facets in a dataset. | Which college application outcomes manifest the greatest disparities by demographic group? | Range of LP values for binary, multicategory, and continuous outcomes: $[0, +1]$
  - Values near zero indicate the labels were evenly distributed between facets in all outcome categories.  
  - Values near one indicate the labels for one category were all in one facet, so very imbalanced.  
  - Intermittent values indicate relative degrees of maximum label imbalance. |
<table>
<thead>
<tr>
<th>Bias metric</th>
<th>Description</th>
<th>Example question</th>
<th>Interpreting metric values</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Conditional Demographic Disparity (CDD) (p. 593)</strong></td>
<td>Measures the disparity of outcomes between different facets as a whole, but also by subgroups.</td>
<td>Do some groups have a larger proportion of rejections for college admission outcomes than their proportion of acceptances?</td>
<td>Range of CDD: [-1, +1] - Positive values indicate a outcomes where facet d is rejected more than accepted. - Near zero indicates no demographic disparity on average. - Negative values indicate a outcomes where facet a is rejected more than accepted.</td>
</tr>
</tbody>
</table>

For additional information about bias metrics, see *Fairness Measures for Machine Learning in Finance*.

**Topics**

- Class Imbalance (CI) (p. 589)
- Difference in Proportions of Labels (DPL) (p. 590)
- Kullback-Leibler Divergence (KL) (p. 590)
- Jensen-Shannon Divergence (JS) (p. 591)
- Lp-norm (LP) (p. 591)
- Total Variation Distance (TVD) (p. 592)
- Kolmogorov-Smirnov (KS) (p. 592)
- Conditional Demographic Disparity (CDD) (p. 593)

**Class Imbalance (CI)**

Class imbalance (CI) bias occurs when a facet value d has fewer training samples when compared with another facet a in the dataset. This is because models preferentially fit the larger facets at the expense of the smaller facets and so can result in a higher training error for facet d. Models are also at higher risk of overfitting the smaller data sets, which can cause a larger test error for facet d. Consider the example where a machine learning model is trained primarily on data from middle-aged individuals, it might be less accurate when making predictions involving younger and older people.

The formula for the (normalized) facet imbalance measure:

\[
CI = \frac{(n_a - n_d)}{(n_a + n_d)}
\]

Where \(n_a\) is the number of members of facet a and \(n_d\) the number for facet d. Its values range over the interval [-1, 1].

- Positive CI values indicate the facet a has more training samples in the dataset and a value of 1 indicates the data only contains members of the facet a.
- Values of CI near zero indicate a more equal distribution of members between facets and a value of zero indicates a perfectly equal partition between facets and represents a balanced distribution of samples in the training data.
• Negative CI values indicate the facet d has more training samples in the dataset and a value of -1 indicates the data only contains members of the facet d.
• CI values near either of the extremes values of -1 or 1 are very imbalanced and are at a substantial risk of making biased predictions.

If a significant facet imbalance is found to exist among the facets, you might want to rebalance the sample before proceeding to train models on it.

**Difference in Proportions of Labels (DPL)**

The difference in proportions of labels (DPL) compares the proportion of observed outcomes with positive labels for facet d with the proportion of observed outcomes with positive labels of facet a in a training dataset. For example, you could use it to compare the proportion of middle-aged individuals (facet a) and other age groups (facet d) approved for financial loans. Machine learning models try to mimic the training data decisions as closely as possible. So a machine learning model trained on a dataset with a high DPL is likely to reflect the same imbalance in its future predictions.

The formula for the difference in proportions of labels is as follows:

\[ \text{DPL} = (q_a - q_d) \]

Where:

• \( q_a = n_a^{(1)}/n_a \) is the proportion of facet a who have an observed label value of 1. For example, the proportion of a middle-aged demographic who get approved for loans. Here \( n_a^{(1)} \) represents the number of members of facet a who get a positive outcome and \( n_a \) the is number of members of facet a.
• \( q_d = n_d^{(1)}/n_d \) is the proportion of facet d who have an observed label value of 1. For example, the proportion of people outside the middle-aged demographic who get approved for loans. Here \( n_d^{(1)} \) represents the number of members of the facet d who get a positive outcome and \( n_d \) the is number of members of the facet d.

If DPL is close enough to 0, then we say that *demographic parity* has been achieved.

For binary and multicategory facet labels, the DPL values range over the interval (-1, 1). For continuous labels, the values vary over the interval \((-\infty, +\infty)\).

• Positive DPL values indicate that facet a is has a higher proportion of positive outcomes when compared with facet d.
• Values of DPL near zero indicate a more equal proportion of positive outcomes between facets and a value of zero indicates perfect demographic parity.
• Negative DPL values indicate that facet d has a higher proportion of positive outcomes when compared with facet a.

Whether or not a high magnitude of DPL is problematic varies from one situation to another. In a problematic case, a high-magnitude DPL might be a signal of underlying issues in the data. For example, a dataset with high DPL might reflect historical biases or prejudices against age-based demographic groups that would be undesirable for a model to learn.

**Kullback-Leibler Divergence (KL)**

The Kullback-Leibler divergence (KL) measures how much the observed label distribution of facet a, \( P_a(y) \), diverges from distribution of facet d, \( P_d(y) \). It is also known as the relative entropy of \( P_a(y) \) with respect to \( P_d(y) \) and quantifies the amount of information lost when moving from \( P_a(y) \) to \( P_d(y) \).

The formula for the Kullback-Leibler divergence is as follows:
KL(P \parallel P_d) = \sum_y P_d(y) \cdot \log[P_a(y)/P_d(y)]

It is the expectation of the logarithmic difference between the probabilities P_a(y) and P_d(y), where the expectation is weighted by the probabilities P_d(y). This is not a true distance between the distributions as it is asymmetric and does not satisfy the triangle inequality. The implementation uses natural logarithms, giving KL in units of nats. Using different logarithmic bases gives proportional results but in different units. For example, using base 2 gives KL in units of bits.

For example, assume that a group of applicants for loans have a 30% approval rate (facet d) and that the approval rate for other applicants (facet a) is 80%. The Kullback-Leibler formula gives you the label distribution divergence of facet a from facet d as follows:

KL = 0.8*ln(0.8/0.3) + 0.2*ln(0.2/0.7) = 0.53

There are two terms in the formula here because labels are binary in this example. This measure can be applied to multiple labels in addition to binary ones. For example, in a college admissions scenario, assume an applicant may be assigned one of three category labels: y_i = \{y_0, y_1, y_2\} = \{rejected, waitlisted, accepted\}.

Range of values for the KL metric for binary, multicategory, and continuous outcomes is \([0, +\infty)\).

- Values near zero mean the outcomes are similarly distributed for the different facets.
- Positive values mean the label distributions diverge, the more positive the larger the divergence.

### Jensen-Shannon Divergence (JS)

The Jensen-Shannon divergence (JS) measures how much the label distributions of different facets diverge from each other entropically. It is based on the Kullback-Leibler divergence, but it is symmetric.

The formula for the Jensen-Shannon divergence is as follows:

JS = \frac{1}{2} \cdot [KL(P_a \parallel P) + KL(P_d \parallel P)]

Where P = \frac{1}{2} (P_a + P_d), the average label distribution across facets a and d.

The range of JS values for binary, multicategory, continuous outcomes is \([0, +\infty)\).

- Values near zero mean the labels are similarly distributed.
- Positive values mean the label distributions diverge, the more positive the larger the divergence.

This metric indicates whether there is a big divergence in one of the labels across facets.

### L_p-norm (LP)

The L_p-norm (LP) measures the p-norm distance between the facet distributions of the observed labels in a training dataset. This metric is non-negative and so cannot detect reverse bias.

The formula for the L_p-norm is as follows:

L_p(P_a, P_d) = (\sum |P_a - P_d|^p)^{1/p}

Where the p-norm distance between the points x and y is defined as follows:

L_p(x, y) = (|x_1-y_1|^p + |x_2-y_2|^p + ... + |x_n-y_n|^p)^{1/p}

The 2-norm is the Euclidean norm. Assume you have an outcome distribution with three categories, for example, y_i = \{y_0, y_1, y_2\} = \{accepted, waitlisted, rejected\} in a college admissions multicategory scenario. You take the sum of the squares of the differences between the outcome counts for facets a and d. The resulting Euclidean distance is calculated as follows:
Detect Pretraining Data Bias

\[ L_2(P_a, P_d) = \left[ (n_a^{(0)} - n_d^{(0)})^2 + (n_a^{(1)} - n_d^{(1)})^2 + (n_a^{(2)} - n_d^{(2)})^2 \right]^{1/2} \]

Where:

- \( n_a^{(i)} \) is number of the ith category outcomes in facet \( a \): for example \( n_a^{(0)} \) is number of facet \( a \) acceptances.
- \( n_d^{(i)} \) is number of the ith category outcomes in facet \( d \): for example \( n_d^{(2)} \) is number of facet \( d \) rejections.

The range of \( L_2 \) values for binary, multicategory, and continuous outcomes is \([0, +\infty)\), where:

- Values near zero mean the labels are similarly distributed.
- Positive values mean the label distributions diverge, the more positive the larger the divergence.

**Total Variation Distance (TVD)**

The total variation distance data bias metric (TVD) is half the \( L_1 \)-norm. The TVD is the largest possible difference between the probability distributions for label outcomes of facets \( a \) and \( d \). The \( L_1 \)-norm is the Hamming distance, a metric used compare two binary data strings by determining the minimum number of substitutions required to change one string into another. If the strings were to be copies of each other, it determines the number of errors that occurred when copying. In the bias detection context, TVD quantifies how many outcomes in facet \( a \) would have to be changed to match the outcomes in facet \( d \).

The formula for the Total variation distance is as follows:

\[ \text{TVD} = \frac{1}{2} \cdot L_1(P_a, P_d) \]

For example, assume you have an outcome distribution with three categories, \( y_i = \{y_0, y_1, y_2\} = \{\text{accepted, waitlisted, rejected}\} \), in a college admissions multicategory scenario. You take the differences between the counts of facets \( a \) and \( d \) for each outcome to calculate TVD. The result is as follows:

\[ L_1(P_a, P_d) = |n_a^{(0)} - n_d^{(0)}| + |n_a^{(1)} - n_d^{(1)}| + |n_a^{(2)} - n_d^{(2)}| \]

Where:

- \( n_a^{(i)} \) is number of the ith category outcomes in facet \( a \): for example \( n_a^{(0)} \) is number of facet \( a \) acceptances.
- \( n_d^{(i)} \) is number of the ith category outcomes in facet \( d \): for example \( n_d^{(2)} \) is number of facet \( d \) rejections.

The range of \( L_1 \) values for binary, multicategory, and continuous outcomes is \([0, +\infty)\), where:

- Values near zero mean the labels are similarly distributed.
- Positive values mean the label distributions diverge, the more positive the larger the divergence.

**Kolmogorov-Smirnov (KS)**

The Kolmogorov-Smirnov bias metric (KS) is equal to the maximum divergence between labels in the distributions for facets \( a \) and \( d \) of a dataset. The two-sample KS test implemented by SageMaker Clarify complements the other measures of label imbalance by finding the most imbalanced label.

The formula for the Kolmogorov-Smirnov metric is as follows:

\[ \text{KS} = \max(|P_a(y) - P_d(y)|) \]

For example, assume a group of applicants (facet \( a \)) to college are rejected, waitlisted, or accepted at 40%, 40%, 20% respectively and that these rates for other applicants (facet \( d \)) are 20%, 10%, 70%. Then the Kolmogorov-Smirnov bias metric value is as follows:

\[ \text{KS} = \max(|0.4-0.2|, |0.4-0.3|, |0.2-0.7|) = 0.5 \]
This tells us the maximum divergence between facet distributions is 0.5 and occurs in the acceptance rates. There are three terms in the equation because labels are multiclass of cardinality three.

The range of LP values for binary, multcategory, and continuous outcomes is \([0, +1]\), where:

- Values near zero indicate the labels were evenly distributed between facets in all outcome categories. For example, both facets applying for a loan got 50% of the acceptances and 50% of the rejections.
- Values near one indicate the labels for one outcome were all in one facet. For example, facet \(a\) got 100% of the acceptances and facet \(d\) got none.
- Intermittent values indicate relative degrees of maximum label imbalance.

**Conditional Demographic Disparity (CDD)**

The demographic disparity metric (DD) determines whether facet \(d\) has a larger proportion of the rejected outcomes in the dataset than of the accepted outcomes. For example, in the case of college admissions, if women applicants comprised 60% of the rejected applicants and comprised only 50% of the accepted applicants, we say that there is demographic disparity because the rate at which women were rejected exceeds the rate at which they are accepted.

The formula for the demographic disparity for the less favored facet \(d\) is as follows:

\[
DD_d = \frac{n_d(0)}{n(0)} - \frac{n_d(1)}{n(1)} = P_d^R(y^0) - P_d^A(y^1)
\]

Where:

- \(n(0) = n_a(0) + n_d(0)\) is the number of rejected outcomes in the dataset.
- \(n(1) = n_a(1) + n_d(1)\) is the number of accepted outcomes in the dataset.
- \(P_d^R(y^0)\) is the proportion of rejected outcomes (with value 0) in facet \(d\).
- \(P_d^A(y^1)\) is the proportion of accepted outcomes (value 1) in facet \(d\).

For the college admission example, the demographic disparity is \(DD = 0.6 - 0.5 = 0.1\).

A conditional demographic disparity (CDD) metric that conditions DD on attributes that define a strata of subgroups on the dataset is needed to rule out Simpson's paradox. The regrouping can provide insights into the cause of apparent demographic disparities for less favored facets. The classic case arose in the case of Berkeley admissions where men were accepted at a higher rate overall than women. However, when departmental subgroups were examined, women were shown to have higher admission rates than men by department. The explanation was that women had applied to departments with lower acceptance rates than men had. Examining the subgrouped acceptance rates revealed that women were actually accepted at a higher rate than men for the departments with lower acceptance rates.

The CDD metric gives a single measure for all of the disparities found in the subgroups defined by an attribute of a dataset by averaging them. It is defined as the weighted average of demographic disparities (DD\(_i\)) for each of the subgroups, with each subgroup disparity weighted in proportion to the number of observations in contains. The formula for the conditional demographic disparity is as follows:

\[
CDD = \frac{1}{n} \cdot \sum_i n_i \cdot DD_i
\]

Where:

- \(\sum_i n_i = n\) is the total number of observations and \(n_i\) is the number of observations for each subgroup.
- \(DD_i = \frac{n_i(0)}{n(0)} - \frac{n_i(1)}{n(1)} = P_i^R(y^0) - P_i^A(y^1)\) is the demographic disparity for the \(i\)th subgroup.

The demographic disparity for a subgroup (DD\(_i\)) are the difference between the proportion of rejected outcomes and the proportion of accepted outcomes for each subgroup.

The range of DD values for binary outcomes is \((-1, +1)\).
Detect Pretraining Data Bias

+1: when there are no rejections in facet \( a \) or subgroup and no acceptances in facet \( d \) or subgroup

-1: when there are no rejections in facet \( d \) or subgroup and no acceptances in facet \( a \) or subgroup

Positive values indicate there is a demographic disparity as facet \( d \) or subgroup has a larger proportion of the rejected outcomes in the dataset than of the accepted outcomes. The higher the value the greater the disparity.

Negative values indicate there is a demographic disparity as facet \( a \) or subgroup has a larger proportion of the rejected outcomes in the dataset than of the accepted outcomes. The lower the value the greater the disparity.

If you don’t condition on anything then CDD is zero if and only if DPL is zero.

This metric is useful for exploring the concepts of direct and indirect discrimination and of objective justification in EU and UK non-discrimination law and jurisprudence. For additional information, see Why Fairness Cannot Be Automated.

Generate Reports for Bias in Pretraining Data in SageMaker Studio

SageMaker Clarify is integrated with Amazon SageMaker Data Wrangler, which can help you identify bias during data preparation without having to write your own code. Data Wrangler provides an end-to-end solution to import, prepare, transform, featureize, and analyze data with Amazon SageMaker Studio. For an overview of the Data Wrangler data prep workflow, see Prepare ML Data with Amazon SageMaker Data Wrangler (p. 526). You specify attributes of interest, such as gender or age, and SageMaker Clarify runs a set of algorithms to detect the presence of bias in those attributes. After the algorithm runs, SageMaker Clarify provides a visual report with a description of the sources and severity of possible bias so that you can plan steps to mitigate. For example, in a financial dataset that contains few examples of business loans to one age group as compared to others, SageMaker flags the imbalance so that you can avoid a model that disfavors that age group.

To analyze and report on data bias

To get started with Data Wrangler, see Get Started with Data Wrangler (p. 527).

1. Open Amazon SageMaker Studio and choose Create Data Flow from the Import and prepare your data tile.
2. From the **Import data** tab, choose **Amazon S3** and then specify your data source on the **Data sources/S3 source** page.

3. After you have imported your data, choose the plus sign on the **Data flow** page and then choose **Add Analysis**.

4. On the **Create Analysis** page, go to the **Configure** panel and then choose **Bias Report** from the **Chart** menu.
5. Configure the bias report by providing the **Name**, the column to predict and whether it is a value or threshold, the column to analyze for bias (the facet) and whether it is a value or threshold.

6. Continue configuring the bias report by choosing the bias metrics.
7. Choose **Check for bias** to generate and view the bias report. Scroll down to view all of the reports.

8. Choose the carrot to the right of the bias metric description to see documentation that can help you interpret the significance of the metric values.
9. To view a table summary of the bias metric values, choose the table. You can save the report for export by choosing Create in the lower-right corner of the page.

10. On the page where your data bias reports are stored, choose the Export tab to download the reports.
Use Your Own Processing Code

You can install libraries to run your scripts in your own processing container or, in a more advanced scenario, you can build your own processing container that satisfies the contract to run in Amazon SageMaker. For a formal specification that defines the contract for an Amazon SageMaker Processing container, see Build Your Own Processing Container (Advanced Scenario) (p. 600).

Topics

- Run Scripts with Your Own Processing Container (p. 599)
- Build Your Own Processing Container (Advanced Scenario) (p. 600)

Run Scripts with Your Own Processing Container

You can use scikit-learn scripts to preprocess data and evaluate your models. To see how to run scikit-learn scripts to perform these tasks see the scikit-learn Processing sample notebook. This notebook uses the ScriptProcessor class from the Amazon SageMaker Python SDK for Processing,

The following example shows how to use a ScriptProcessor class to run a Python script with your own image that runs a processing job that processes input data, and saves the processed data in Amazon Simple Storage Service (Amazon S3).

The notebook shows the general workflow for using a ScriptProcessor class.

1. Create a Docker directory and add the Dockerfile used to create the processing container. Install pandas and scikit-learn into it. (You could also install your own dependencies with a similar RUN command.)

```bash
mkdir docker
%%writefile docker/Dockerfile
```
Use Your Own Processing Code

FROM python:3.7-slim-buster

RUN pip3 install pandas==0.25.3 scikit-learn==0.21.3
ENV PYTHONUNBUFFERED=TRUE
ENTRYPOINT ["python3"]

2. Build the container using the docker command, create an Amazon Elastic Container Registry (Amazon ECR) repository, and push the image to Amazon ECR.

   ```python
   import boto3

   account_id = boto3.client('sts').get_caller_identity().get('Account')
   ecr_repository = 'sagemaker-processing-container'
   tag = ':latest'
   processing_repository_uri = '{}.dkr.ecr.{}.amazonaws.com/{}'.format(account_id, region, ecr_repository + tag)

   # Create ECR repository and push docker image
   !docker build -t $ecr_repository docker
   !aws ecr get-login-password --region region | docker login --username AWS --password-stdin aws_account_id.dkr.ecr.region.amazonaws.com
   !aws ecr create-repository --repository-name $ecr_repository
   !docker tag {ecr_repository + tag} $processing_repository_uri
   !docker push $processing_repository_uri
   ```

3. Set up the ScriptProcessor from the SageMaker Python SDK to run the script.

   ```python
   from sagemaker.processing import ScriptProcessor, ProcessingInput, ProcessingOutput

   script_processor = ScriptProcessor(command=['python3'],
                                       image_uri='<image_uri>',
                                       role='<role_arn>',
                                       instance_count=1,
                                       instance_type='ml.m5.xlarge')
   ```

4. Run the script.

   ```python
   script_processor.run(code='preprocessing.py',
                        inputs=[ProcessingInput(
                            source='s3://path/to/my/input-data.csv',
                            destination='/opt/ml/processing/input')],
                        train='),
                       outputs=[ProcessingOutput(source='/opt/ml/processing/output/
                       train'),
                       validation'),
                       ProcessingOutput(source='/opt/ml/processing/output/
                       train'),
                       test')]]
   ```

You can use the same procedure with any other library or system dependencies. You can also use existing Docker images. This includes images that you run on other platforms such as Kubernetes.

Build Your Own Processing Container (Advanced Scenario)

You can provide Amazon SageMaker Processing with a Docker image that has your own code and dependencies to run your data processing, feature engineering, and model evaluation workloads.

The following example of a Dockerfile builds a container with the Python libraries scikit-learn and pandas, which you can run as a processing job.

FROM python:3.7-slim-buster
Use Your Own Processing Code

# Install scikit-learn and pandas
RUN pip3 install pandas==0.25.3 scikit-learn==0.21.3

# Add a Python script and configure Docker to run it
ADD processing_script.py /
ENTRYPOINT ["python3", "/processing_script.py"]

Build and push this Docker image to an Amazon Elastic Container Registry (Amazon ECR) repository and ensure that your SageMaker IAM role can pull the image from Amazon ECR. Then you can run this image on Amazon SageMaker Processing.

How Amazon SageMaker Processing Runs Your Processing Container Image

Amazon SageMaker Processing runs your processing container image in a similar way as the following command, where AppSpecification.ImageUri is the Amazon ECR image URI that you specify in a CreateProcessingJob operation.

```
docker run [AppSpecification.ImageUri]
```

This command runs the ENTRYPOINT command configured in your Docker image.

You can also override the entrypoint command in the image or give command-line arguments to your entrypoint command using the AppSpecification.ContainerEntrypoint and AppSpecification.ContainerArgument parameters in your CreateProcessingJob request. Specifying these parameters configures Amazon SageMaker Processing to run the container similar to the way that the following command does.

```
```

For example, if you specify the ContainerEntrypoint to be [python3, -v, /processing_script.py] in your CreateProcessingJob request, and ContainerArguments to be [data-format, csv], Amazon SageMaker Processing runs your container with the following command.

```
python3 -v /processing_script.py data-format csv
```

When building your processing container, consider the following details:

- Amazon SageMaker Processing decides whether the job completes or fails depending on the exit code of the command run. A processing job completes if all of the processing containers exit successfully with an exit code of 0, and fails if any of the containers exits with a non-zero exit code.
- Amazon SageMaker Processing lets you override the processing container's entrypoint and set command-line arguments just like you can with the Docker API. Docker images can also configure the entrypoint and command-line arguments using the ENTRYPOINT and CMD instructions. The way CreateProcessingJob's ContainerEntrypoint and ContainerArgument parameters configure a Docker image's entrypoint and arguments mirrors how Docker overrides the entrypoint and arguments through the Docker API:
  - If neither ContainerEntrypoint nor ContainerArguments are provided, Processing uses the default ENTRYPOINT or CMD in the image.
  - If ContainerEntrypoint is provided, but not ContainerArguments, Processing runs the image with the given entrypoint, and ignores the ENTRYPOINT and CMD in the image.
  - If ContainerArguments is provided, but not ContainerEntrypoint, Processing runs the image with the default ENTRYPOINT in the image and with the provided arguments.
• If both ContainerEntrypoint and ContainerArguments are provided, Processing runs the image with the given entrypoint and arguments, and ignores the ENTRYPOINT and CMD in the image.

• You must use the exec form of the ENTRYPOINT instruction in your Dockerfile (ENTRYPOINT ["executable", "param1", "param2"]) instead of the shell form (ENTRYPOINT command param1 param2). This lets your processing container receive SIGINT and SIGHUP signals, which Processing uses to stop processing jobs with the StopProcessingJob API.

• /opt/ml and all its subdirectories are reserved by SageMaker. When building your Processing Docker image, don't place any data required by your processing container in these directories.

• If you plan to use GPU devices, make sure that your containers are nvidia-docker compatible. Include only the CUDA toolkit in containers. Don't bundle NVIDIA drivers with the image. For more information about nvidia-docker, see NVIDIA/nvidia-docker.

How Amazon SageMaker Processing Configures Input and Output For Your Processing Container

When you create a processing job using the CreateProcessingJob operation, you can specify multiple ProcessingInput and ProcessingOutput values.

You use the ProcessingInput parameter to specify an Amazon Simple Storage Service (Amazon S3) URI to download data from, and a path in your processing container to download the data to. The ProcessingOutput parameter configures a path in your processing container from which to upload data, and where in Amazon S3 to upload that data to. For both ProcessingInput and ProcessingOutput, the path in the processing container must begin with /opt/ml/processing/.

For example, you might create a processing job with one ProcessingInput parameter that downloads data from s3://your-data-bucket/path/to/input/csv/data into /opt/ml/processing/csv in your processing container, and a ProcessingOutput parameter that uploads data from /opt/ml/processing/processed_csv to s3://your-data-bucket/path/to/output/csv/data. Your processing job would read the input data, and write output data to /opt/ml/processing/processed_csv. Then it uploads the data written to this path to the specified Amazon S3 output location.

Important

Symbolic links (symlinks) can not be used to upload output data to Amazon S3. Symlinks are not followed when uploading output data.

How Amazon SageMaker Processing Provides Logs and Metrics for Your Processing Container

When your processing container writes to stdout or stderr, Amazon SageMaker Processing saves the output from each processing container and puts it in Amazon CloudWatch logs. For information about logging, see Log Amazon SageMaker Events with Amazon CloudWatch (p. 1712).

Amazon SageMaker Processing also provides CloudWatch metrics for each instance running your processing container. For information about metrics, see Monitor Amazon SageMaker with Amazon CloudWatch (p. 1704).

How Amazon SageMaker Processing Configures Your Processing Container

Amazon SageMaker Processing provides configuration information to your processing container through environment variables and two JSON files—/opt/ml/config/processingjobconfig.json and /opt/ml/config/resourceconfig.json—at predefined locations in the container.

When a processing job starts, it uses the environment variables that you specified with the Environment map in the CreateProcessingJob request. The /opt/ml/config/
processingjobconfig.json file contains information about the hostnames of your processing containers, and is also specified in the CreateProcessingJob request.

The following example shows the format of the /opt/ml/config/processingjobconfig.json file.

```
{
  "ProcessingJobArn": "<processing_job_arn>",
  "ProcessingJobName": "<processing_job_name>",
  "AppSpecification": {
    "ImageUri": "<image_uri>",
    "ContainerEntrypoint": null,
    "ContainerArguments": null
  },
  "Environment": {
    "KEY": "VALUE"
  },
  "ProcessingInputs": [
    {
      "InputName": "input-1",
      "S3Input": {
        "LocalPath": "/opt/ml/processing/input/dataset",
        "S3Uri": "<s3_uri>",
        "S3DataDistributionType": "FullyReplicated",
        "S3DataType": "S3Prefix",
        "S3InputMode": "File",
        "S3CompressionType": "None",
        "S3DownloadMode": "StartOfJob"
      }
    }
  ],
  "ProcessingOutputConfig": {
    "Outputs": [
      {
        "OutputName": "output-1",
        "S3Output": {
          "LocalPath": "/opt/ml/processing/output/dataset",
          "S3Uri": "<s3_uri>",
          "S3UploadMode": "EndOfJob"
        }
      }
    ],
    "KmsKeyId": null
  },
  "ProcessingResources": {
    "ClusterConfig": {
      "InstanceCount": 1,
      "InstanceType": "ml.m5.xlarge",
      "VolumeSizeInGB": 30,
      "VolumeKmsKeyId": null
    }
  },
  "RoleArn": "<IAM role>",
  "StoppingCondition": {
    "MaxRuntimeInSeconds": 86400
  }
}
```

The /opt/ml/config/resourceconfig.json file contains information about the hostnames of your processing containers. Use the following hostnames when creating or running distributed processing code.

```
{
  "current_host": "algo-1",
```
"hosts": ["algo-1","algo-2","algo-3"]
}

Don't use the information about hostnames contained in /etc/hostname or /etc/hosts because it might be inaccurate.

Hostname information might not be immediately available to the processing container. We recommend adding a retry policy on hostname resolution operations as nodes become available in the cluster.

**Save and Access Metadata Information About Your Processing Job**

To save metadata from the processing container after exiting it, containers can write UTF-8 encoded text to the /opt/ml/output/message file. After the processing job enters any terminal status ("Completed", "Stopped", or "Failed"), the "ExitMessage" field in DescribeProcessingJob contains the first 1 KB of this file. Access that initial part of file with a call to DescribeProcessingJob, which returns it through the ExitMessage parameter. For failed processing jobs, you can use this field to communicate information about why the processing container failed.

**Important**

Don't write sensitive data to the /opt/ml/output/message file.

If the data in this file isn't UTF-8 encoded, the job fails and returns a ClientError. If multiple containers exit with an ExitMessage, the content of the ExitMessage from each processing container is concatenated, then truncated to 1 KB.

**Run Your Processing Container Using the SageMaker Python SDK**

You can use the SageMaker Python SDK to run your own processing image by using the Processor class. The following example shows how to run your own processing container with one input from Amazon Simple Storage Service (Amazon S3) and one output to Amazon S3.

```python
from sagemaker.processing import Processor, ProcessingInput, ProcessingOutput

processor = Processor(image_uri='<your_ecr_image_uri>',
                       role=role,
                       instance_count=1,
                       instance_type="ml.m5.xlarge")

processor.run(inputs=[ProcessingInput(
    source='<s3_uri or local path>',
    destination='/opt/ml/processing/input_data')],
               outputs=[ProcessingOutput(
    source='/opt/ml/processing/processed_data',
    destination='<s3_uri>')] )
```

Instead of building your processing code into your processing image, you can provide a ScriptProcessor with your image and the command that you want to run, along with the code that you want to run inside that container. For an example, see Run Scripts with Your Own Processing Container (p. 599).

You can also use the scikit-learn image that Amazon SageMaker Processing provides through SKLearnProcessor to run scikit-learn scripts. For an example, see Data Processing with scikit-learn (p. 583).
Amazon SageMaker Developer Guide
How Feature Store Works

Create, Store, and Share Features with Amazon SageMaker Feature Store

The machine learning (ML) development process often begins with extracting data signals also known as features from data to train ML models. Amazon SageMaker Feature Store makes it easy for data scientists, machine learning engineers, and general practitioners to create, share, and manage features for machine learning (ML) development. Feature Store accelerates this process by reducing repetitive data processing and curation work required to convert raw data into features for training an ML algorithm.

Further, the processing logic for your data is authored only once, and features generated are used for both training and inference, reducing the training-serving skew. Feature Store is a centralized store for features and associated metadata so features can be easily discovered and reused. You can create an online or an offline store. The online store is used for low latency real-time inference use cases, and the offline store is used for training and batch inference.

The following diagram shows how you can use Amazon SageMaker Feature Store as part of your machine learning pipeline. First, you read in your raw data and process it. You can ingest data via streaming to the online and offline store, or in batches directly to the offline store. You first create a FeatureGroup and configure it to an online or offline store, or both. Then, you can ingest data into your FeatureGroup and store it in your store. A FeatureGroup is a group of features that is defined via a schema in Feature Store to describe a record.

Online store is primarily designed for supporting real-time predictions that need low millisecond latency reads and high throughput writes. Offline store is primarily intended for batch predictions and model training. Offline store is an append only store and can be used to store and access historical feature data. The offline store can help you store and serve features for exploration and model training. The online store retains only the latest feature data. Feature group definitions are immutable after they are created.

How Feature Store Works

In Feature Store, features are stored in a collection called a feature group. You can visualize a feature group as a table in which each column is a feature, with a unique identifier for each row. In principle, a
A feature group is composed of features and values specific to each feature. A Record is a collection of values for features that correspond to a unique RecordIdentifier. Altogether, a FeatureGroup is a group of features defined in your FeatureStore to describe a Record.

You can use Feature Store in the following modes:

- **Online** – In online mode, features are read with low latency (milliseconds) reads and used for high throughput predictions. This mode requires a feature group to be stored in an online store.
- **Offline** – In offline mode, large streams of data are fed to an offline store, which can be used for training and batch inference. This mode requires a feature group to be stored in an offline store. The offline store uses your S3 bucket for storage and can also fetch data using Athena queries.
- **Online and Offline** – This includes both online and offline modes.

You can ingest data into feature groups in Feature Store in two ways: streaming or in batches. When you ingest data through streaming, a collection of records are pushed to Feature Store by calling a synchronous PutRecord API call. This API enables you to maintain the latest feature values in Feature Store and to push new feature values as soon an update is detected.

Alternatively, Feature Store can process and ingest data in batches. You can author features using Amazon SageMaker Data Wrangler, create feature groups in Feature Store and ingest features in batches using a SageMaker Processing job with a notebook exported from Data Wrangler. This mode allows for batch ingestion into the offline store. It also supports ingestion into the online store if the feature group is configured for both online and offline use.

### Create Feature Groups

To ingest features into Feature Store, you must first define the feature group and the feature definitions (feature name and data type) for all features that belong to the feature group. After they are created, feature groups are immutable. Feature group names are unique within an AWS Region and AWS account. When creating a feature group, you can also create the metadata for the feature group, such as a short description, storage configuration, features for identifying each record, and the event time, as well as tags to store information such as the author, data source, version, and more.

### Find, Discover, and Share Features

After you create a feature group in Feature Store, other authorized users of the feature store can share and discover it. Users can browse through a list of all feature groups in Feature Store or discover existing feature groups by searching by feature group name, description, record identifier name, creation date, and tags.

### Real-Time Inference for Features Stored in the Online Store

With Feature Store, you can enrich your features stored in the online store in real time with data from a streaming source (clean stream data from another application) and serve the features with low millisecond latency for real-time inference.

You can also perform joins across different FeatureGroups for real-time inference by querying two different FeatureGroups in the client application.
Offline Store for Model Training and Batch Inference

Feature Store provides offline storage for feature values in your S3 bucket. Your data is stored in your S3 bucket using a prefixing scheme based on event time. The offline store is an append-only store, enabling Feature Store to maintain a historical record of all feature values. Data is stored in the offline store in Parquet format for optimized storage and query access.

You can query, explore, and visualize features using Data Wrangler from Amazon SageMaker Studio. Feature Store supports combining data to produce, train, validate, and test data sets, and allows you to extract data at different points in time.

Feature Data Ingestion

Feature generation pipelines can be created to process large batches (1 million rows of data or more) or small batches, and to write feature data to the offline or online store. Streaming sources such as Amazon Managed Streaming for Apache Kafka or Amazon Kinesis can also be used as data sources from which features are extracted and directly fed to the online store for training, inference, or feature creation.

You can push records to Feature Store by calling the synchronous `PutRecord` API call. Since this is a synchronous API call, it allows small batches of updates to be pushed in a single API call. This enables you to maintain high freshness of the feature values and publish values as soon as an update is detected. These are also called streaming features.

When feature data is ingested and updated, Feature Store stores historical data for all features in the offline store. For batch ingest, you can pull feature values from your S3 bucket or use Athena to query. You can also use Data Wrangler to process and engineer new features that can then be exported to a chosen S3 bucket to be accessed by Feature Store. For batch ingestion, you can configure a processing job to batch ingest your data into Feature Store, or you can pull feature values from your S3 bucket using Athena.

Access Control

Feature Store allows you to grant or deny access to individuals at the feature group-level and enables cross-account access to Feature Store. For example, you can set up developer accounts to access the offline store for model training and exploration that do not have write access to production accounts. You can set up production accounts to access both online and offline stores. Feature Store uses unique customer AWS KMS CMKs for offline and online store data at-rest encryption. Access control is enabled through both API and KMS CMK access. You can also create feature group-level access control.

Get started with Amazon SageMaker Feature Store

To get started using Amazon SageMaker Feature Store, review the basic concepts, learn how to ingest data for your feature store, and then walk through a Feature Store example. The following sections explain how to create feature groups, ingest data into the groups, and how to manage security for your feature store.

Topics

- Feature Store Concepts (p. 608)
- Create Feature Groups (p. 608)
Feature Store Concepts

The following list of terms are key to understanding the capabilities of Amazon SageMaker Feature Store:

- **Feature store** – Serves as the single source of truth to store, retrieve, remove, track, share, discover, and control access to features.
- **Feature** – A measurable property or characteristic that encapsulates an observed phenomenon. In the Amazon SageMaker Feature Store API, a feature is an attribute of a record. You can define a name and type for every feature stored in Feature Store. Name uniquely identifies a feature within a feature group. Type identifies the datatype for the values of the feature. Supported datatypes are: String, Integral and Fractional.
- **Feature group** – A FeatureGroup is the main Feature Store resource that contains the metadata for all the data stored in Amazon SageMaker Feature Store. A feature group is a logical grouping of features, defined in the feature store, to describe records. A feature group's definition is composed of a list of feature definitions, a record identifier name, and configurations for its online and offline store.
- **Feature definition** – A FeatureDefinition consists of a name and one of the following data types: an Integral, String or Fractional. A FeatureGroup contains a list of feature definitions.
- **Record identifier name** – Each feature group is defined with a record identifier name. The record identifier name must refer to one of the names of a feature defined in the feature group's feature definitions.
- **Record** – A Record is a collection of values for features for a single record identifier value. A combination of record identifier name and a timestamp uniquely identify a record within a feature group.
- **Event time** – a point in time when a new event occurs that corresponds to the creation or update of a record in a feature group. All records in the feature group must have a corresponding EventTime. It can be used to track changes to a record over time. The online store contains the record corresponding to the last EventTime for a record identifier name, whereas the offline store contains all historic records.
- **Online Store** – the low latency, high availability cache for a feature group that enables real-time lookup of records. The online store allows quick access to the latest value for a Record via the GetRecord API. A feature group contains an OnlineStoreConfig controlling where the data is stored.
- **Offline store** – the OfflineStore, stores historical data in your S3 bucket. It is used when low (sub-second) latency reads are not needed. For example, when you want to store and serve features for exploration, model training, and batch inference. A feature group contains an OfflineStoreConfig controlling where the data is stored.
- **Ingestion** – The act of populating feature groups in the feature store.

Create Feature Groups

A FeatureGroup is the main Feature Store resource that contains the metadata for all the data stored in Amazon SageMaker Feature Store. A feature group is a logical grouping of features, defined in the feature store, to describe records. A feature group's definition is composed of a list of feature definitions, a record identifier name, and configurations for its online and offline store. The example code in this topic uses the SageMaker Python SDK. The underlying APIs are available for developers using other languages.

Prior to using a feature store you typically load your dataset, run transformations, and set up your features for ingestion. This process has a lot of variation and is highly dependent on your data.
example code in this topic refer to the Fraud Detection with Amazon SageMaker FeatureStore example notebook. We recommend that you run this notebook in Amazon SageMaker Studio because the code in this guide is conceptual and not fully functional if copied.

Feature Store supports the following data types: String, Fractional, and Integral. The default type is set to String. This means that, if a column in your dataset is not a float or long type, it defaults to String in your feature store.

You may use a schema to describe your data's columns and data types. You pass this schema into FeatureDefinitions, a required parameter for a FeatureGroup. You can use the SageMaker Python SDK, which has automatic data type detection when you use the load_feature_definitions function.

Topics
- Step 1: Set Up Feature Store (p. 609)
- Step 2: Load Datasets and Partition Data into Feature Groups (p. 609)
- Step 3: Set Up Feature Groups (p. 610)
- Step 4: Set Up Record Identifier and Event Time Features (p. 611)
- Step 5: Load Feature Definitions (p. 611)
- Step 6: Create a Feature Group (p. 611)
- Step 7: Work with Feature Groups (p. 612)

Step 1: Set Up Feature Store

To start using Feature Store, create a SageMaker session, boto3 session, and a Feature Store session. Also, set up the S3 bucket you want to use for your features. This is your offline store. The following code uses the SageMaker default bucket and adds a custom prefix to it.

Note
The role that you use must have the following managed policies attached to it:
AmazonSageMakerFullAccess and AmazonSageMakerFeatureStoreAccess.

```python
import boto3
import sagemaker
from sagemaker.session import Session

boto_session = boto3.Session(region_name=region)
role = sagemaker.get_execution_role()
sagemaker_session = sagemaker.Session()
region = sagemaker_session.boto_region_name
default_bucket = sagemaker_session.default_bucket()
prefix = 'sagemaker-featurestore'
offline_feature_store_bucket = 's3://{}{}'.format(default_bucket, prefix)
sagemaker_client = boto_session.client(service_name='sagemaker', region_name=region)
featurestore_runtime = boto_session.client(service_name='sagemaker-featurestore-runtime', region_name=region)

feature_store_session = Session(
    boto_session=boto_session,
    sagemaker_client=sagemaker_client,
    sagemaker_featurestore_runtime_client=featurestore_runtime
)
```

Step 2: Load Datasets and Partition Data into Feature Groups

Load your data into data frames for each of your features. You use these data frames after you set up the feature group. In the fraud detection example, you can see these steps in the following code.
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import io

fraud_detection_bucket_name = 'sagemaker-featurestore-fraud-detection'
identity_file_key = 'sampled_identity.csv'
transaction_file_key = 'sampled_transactions.csv'

identity_data_object = s3_client.get_object(Bucket=fraud_detection_bucket_name,
                                        Key=identity_file_key)
transaction_data_object = s3_client.get_object(Bucket=fraud_detection_bucket_name,
                                        Key=transaction_file_key)

identity_data = pd.read_csv(io.BytesIO(identity_data_object['Body'].read()))
transaction_data = pd.read_csv(io.BytesIO(transaction_data_object['Body'].read()))

identity_data = identity_data.round(5)
transaction_data = transaction_data.round(5)

identity_data = identity_data.fillna(0)
transaction_data = transaction_data.fillna(0)

# Feature transformations for this dataset are applied before ingestion into FeatureStore.
# One hot encode card4, card6
encoded_card_bank = pd.get_dummies(transaction_data['card4'], prefix = 'card_bank')
encoded_card_type = pd.get_dummies(transaction_data['card6'], prefix = 'card_type')

transformed_transaction_data = pd.concat([transaction_data, encoded_card_type,
                                            encoded_card_bank], axis=1)
transformed_transaction_data = transformed_transaction_data.rename(columns={
                            "card_bank_american_express": "card_bank_american_express"})

Step 3: Set Up Feature Groups

When you set up your feature groups, you need to customize the feature names with a unique name and
set up each feature group with the FeatureGroup class.

from sagemaker.feature_store.feature_group import FeatureGroup
feature_group_name = "some string for a name"
feature_group = FeatureGroup(name=feature_group_name,
sagemaker_session=feature_store_session)

For example, in the fraud detection example, the two feature groups are identity and transaction.
In the following code you can see how the names are customized with a timestamp, and then each group
is set up by passing in the name and the session.

import time
from time import gmtime, strftime, sleep
from sagemaker.feature_store.feature_group import FeatureGroup

identity_feature_group_name = 'identity-feature-group-' + strftime('%d-%H-%M-%S', gmtime())
transaction_feature_group_name = 'transaction-feature-group-' + strftime('%d-%H-%M-%S',
gmtime())

identity_feature_group = FeatureGroup(name=identity_feature_group_name,
sagemaker_session=feature_store_session)
transaction_feature_group = FeatureGroup(name=transaction_feature_group_name,
sagemaker_session=feature_store_session)
Step 4: Set Up Record Identifier and Event Time Features

In this step, you specify a record identifier name and an event time feature name. This name maps to the column of the corresponding features in your data. For example, in the fraud detection example, the column of interest is TransactionID. EventTime can be appended to your data when no timestamp is available. In the following code, you can see how these variables are set, and then EventTime is appended to both feature's data.

```python
record_identifier_name = "TransactionID"
event_time_feature_name = "EventTime"
current_time_sec = int(round(time.time()))
identity_data[event_time_feature_name] = pd.Series([current_time_sec]*len(identity_data),
                                                 dtype="float64")
transformed_transaction_data[event_time_feature_name] =
                                 pd.Series([current_time_sec]*len(transaction_data), dtype="float64")
```

Step 5: Load Feature Definitions

You can now load the feature definitions by passing a data frame containing the feature data. In the following code for the fraud detection example, the identity feature and transaction feature are each loaded by using `load_feature_definitions`, and this function automatically detects the data type of each column of data. For developers using a schema rather than automatic detection, see the Export Feature Groups from Data Wrangler example for code that shows how to load the schema, map it, and add it as a FeatureDefinition that you can use to create the FeatureGroup. This example also covers a boto3 implementation, which you can use instead of the SageMaker Python SDK.

```python
identity_feature_group.load_feature_definitions(data_frame=identity_data); # output is suppressed
transaction_feature_group.load_feature_definitions(data_frame=transformed_transaction_data);
# output is suppressed
```

Step 6: Create a Feature Group

In this step, you use the `create` function to create the feature group. The following code shows all of the available parameters. The online store is not created by default, so you must set this as True if you want to enable it. The `s3_uri` is the S3 bucket location of your offline store.

```python
# create a FeatureGroup
feature_group.create(
    description = "Some info about the feature group",
    feature_group_name = feature_group_name,
    record_identifier_name = record_identifier_name,
    event_time_feature_name = event_time_feature_name,
    feature_definitions = feature_definitions,
    role_arn = role,
    s3_uri = offline_feature_store_bucket,
    enable_online_store = True,
    online_store_kms_key_id = None,
    offline_store_kms_key_id = None,
    disable_glue_table_creation = False,
    data_catalog_config = None,
    tags = ["tag1","tag2"]
)
```

The following code from the fraud detection example shows a minimal `create` call for each of the two features groups being created.

```python
identity_feature_group.create(
    s3_uri=offline_feature_store_bucket,
)
When you create a feature group, it takes time to load the data, and you need to wait until the feature group is created before you can use it. You can check status using the following method.

```python
status = feature_group.describe().get("FeatureGroupStatus")
```

While the feature group is being created, you receive `Creating` as a response. When this step has finished successfully, the response is `Created`. Other possible statuses are `CreateFailed`, `Deleting`, or `DeleteFailed`.

**Step 7: Work with Feature Groups**

Now that you've set up your feature group, you can perform any of the following tasks:

**Topics**
- Describe a Feature Group (p. 612)
- List Feature Groups (p. 612)
- Put Records in a Feature Group (p. 612)
- Get Records from a Feature Group (p. 613)
- Generate Hive DDL Commands (p. 613)
- Build a Training Dataset (p. 613)
- Write and Execute an Athena Query (p. 614)
- Delete a Feature Group (p. 614)

**Describe a Feature Group**

You can retrieve information about your feature group with the `describe` function.

```python
feature_group.describe()
```

**List Feature Groups**

You can list all of your feature groups with the `list_feature_groups` function.

```python
sagemaker_client.list_feature_groups()
```

**Put Records in a Feature Group**

You can use the `ingest` function to load your feature data. You pass in a data frame of feature data, set the number of workers, and choose to wait for it to return or not. The following example demonstrates using the `ingest` function.
For each feature group you have, run the `ingest` function on the feature data you want to load.

### Get Records from a Feature Group

You can use the `get_record` function to retrieve the data for a specific feature by its record identifier. The following example uses an example identifier to retrieve the record.

```python
record_identifier_value = str(2990130)
featurestore_runtime.get_record(FeatureGroupName=transaction_feature_group_name,
RecordIdentifierValueAsString=record_identifier_value)
```

An example response from the fraud detection example:

...  
'Record': [{'FeatureName': 'TransactionID', 'ValueAsString': '2990130'},
{'FeatureName': 'isFraud', 'ValueAsString': '0'},
{'FeatureName': 'TransactionDT', 'ValueAsString': '152647'},
{'FeatureName': 'TransactionAmt', 'ValueAsString': '75.0'},
{'FeatureName': 'ProductCD', 'ValueAsString': 'H'},
{'FeatureName': 'card1', 'ValueAsString': '4577'},
...

### Generate Hive DDL Commands

The SageMaker Python SDK's `FeatureStore` class also provides the functionality to generate Hive DDL commands. The schema of the table is generated based on the feature definitions. Columns are named after feature name and data-type are inferred based on feature type.

```python
print(feature_group.as_hive_ddl())
```

Example output:

```
CREATE EXTERNAL TABLE IF NOT EXISTS sagemaker_featurestore.identity-feature-group-27-19-33-00 (
    TransactionID INT,
    id_01 FLOAT,
    id_02 FLOAT,
    id_03 FLOAT,
    id_04 FLOAT,
    ...
```

### Build a Training Dataset

Feature Store automatically builds an AWS Glue data catalog when you create feature groups and you can turn this off if you want. The following describes how to create a single training dataset with feature values from both identity and transaction feature groups created earlier in this topic. Also, the following describes how to run an Amazon Athena query to join data stored in the offline store from both identity and transaction feature groups.

To start, create an Athena query using `athena_query()` for both identity and transaction feature groups. The `table_name` is the AWS Glue table that is autogenerated by Feature Store.

```python
identity_query = identity_feature_group.athena_query()
```
Write and Execute an Athena Query

You write your query using SQL on these feature groups, and then execute the query with the `.run()` command and specify your S3 bucket location for the data set to be saved there.

```python
# Athena query
query_string = 'SELECT * FROM "' + transaction_table + '" LEFT JOIN "' + identity_table + '" ON "' + transaction_table + '".transactionid = "' + identity_table + '".transactionid'

# run Athena query. The output is loaded to a Pandas dataframe.
dataset = pd.DataFrame()
identity_query.run(query_string=query_string,
output_location='s3://' + default_s3_bucket_name + '/query_results/')
identity_query.wait()
dataset = identity_query.as_dataframe()
```

From here you can train a model using this data set and then perform inference.

Delete a Feature Group

You can delete a feature group with the `delete` function.

```python
feature_group.delete()
```

The following code example is from the fraud detection example.

```python
identity_feature_group.delete()
transaction_feature_group.delete()
```

For more information, see the Delete a feature group API

Use Amazon SageMaker Feature Store with Amazon SageMaker Studio

You can use Studio to create and view details about your feature groups.

Topics
- Create a Feature Group in Studio (p. 614)
- View Feature Group Details in Studio (p. 619)

Create a Feature Group in Studio

The create feature group process in Studio has four steps: enter details, definitions, required features, and tags.

Consider the following options before you start:
- If you plan to only use an online store, you just need the schema for your features. This is your columns and each column’s data type.
- If you plan to use an offline store, you need an S3 bucket URI and a Role ARN.
• If you plan to use encryption, you need a KMS key. You can use the same one for both the online store and your offline store, or have a unique key for each.

• If you plan to use AWS Glue integration, be prepared to provide a data catalog name, database name, and table name.

**To create a feature group in Studio**

1. Sign in to Studio. For more information, see [Onboard to Amazon SageMaker Studio](#).

2. In the left navigation pane, choose the **Components and registries** icon.

3. In the file and resource browser, choose **Feature Store**.

![Feature Store](image-url)
4. The **Feature Store** tab lists your feature groups.

5. The **Feature Store** tab, choose **Create Feature Group**.

6. Enter the feature group details.

7. (Optional) If you’re using Glue, enter the data catalog details.
8. Enter feature definitions. You have two options for providing a schema for your features: a JSON editor, or a Table editor.

9. The table editor accepts a column name and a data type. In a minimal example, you will need at least two for the next step: one for a record identifier and one for a time event feature. You can have up to 2,500 feature definitions.
10. Set a record identifier and a time feature to use for this feature group.

11. (Optional) Enter tags as key value pairs.
12. Choose **Create feature group**.

13. In the **Actions** column, choose **Open feature store**.

14. When the feature group is finished being created, it appears in your feature groups list. Choose the refresh button to refresh the list.

**View Feature Group Details in Studio**

You can view details about your feature groups, and get sample queries to run against your data sources to gather the data for features you have defined.

**To view feature group details in Studio**

1. Double-click or right-click a feature group from your list, and then choose **Open feature group detail**.
2. In the details view, you can review your feature group summary, feature definitions, feature group tags, and sample query.

3. On the Feature definitions tab, you can search for features by name.
4. On the **Feature group tags** tab, you can add and remove tags.

5. On the **Sample query** tab, you can see a variety of sample queries: interactive exploration, time travel, remove tombstone, and remove duplicates.

---

**Data Sources and Ingestion**

There are multiple ways to bring your data into Amazon SageMaker Feature Store. Feature Store offers a single API call for data ingestion called `GetRecord` that enables you to ingest data in batches or from
streaming sources. You can also use Amazon SageMaker Data Wrangler to engineer features and then ingest your features into your Feature Store.

**Topics**
- Stream Ingestion (p. 622)
- Data Wrangler with Feature Store (p. 622)
- Athena and AWS Glue with Feature Store (p. 624)

**Stream Ingestion**

You can use streaming sources such as Kafka or Kinesis as a data source where features are extracted from there and directly fed to the online feature store for training, inference or feature creation. Records can be pushed into the feature store by calling the synchronous `PutRecord` API call. Since this is a synchronous API call it allows small batches of updates to be pushed in a single API call. This enables you to maintain high freshness of the feature values and publish values as soon an update is detected. These are also called *streaming features.*

**Data Wrangler with Feature Store**

Data Wrangler is a feature of Studio that provides an end-to-end solution to import, prepare, transform, featurize, and analyze data. Data Wrangler enables you to engineer your features and ingest them into a feature store.

In Studio, after interacting with Data Wrangler, choose the **Export** tab, choose **Export Step**, and the choose **Feature Store**, as shown in the following screenshot. This exports a Jupyter notebook that has all the source code in it to create a Feature Store feature group that adds yours features from Data Wrangler to an offline or online feature store.
After the feature group has been created, you can also select and join data across multiple feature groups to create new engineered features in Data Wrangler and then export your data set to an S3 bucket.
Athena and AWS Glue with Feature Store

After a Feature Store feature group has been created in an offline feature store, you can choose to run queries using Amazon Athena on an AWS Glue catalog. This requires data to be registered in a data catalog with other catalog details which is auto-registered for you in Feature Store. In other words, Feature Store automatically builds an AWS Glue data catalog when feature groups are created and you can turn them off. This is particularly useful when you want to build a dataset by executing SQL queries and then train a model for inference.

After your FeatureStore has been created and populated with your data in the offline store, you have the capability to write SQL queries to join data stored in the offline store from different FeatureGroups. To do this, you can use Amazon Athena to write and execute SQL queries. You can set up a AWS Glue crawler to run on a schedule to ensure your catalog is always up to date as well.

If you want to do this please define a role which can used by the AWS Glue crawler to access the offline store's S3 buckets. For more information, see Create an IAM role.

For more information on how to use AWS Glue and Athena to build a training dataset for model training and inference, see Build Training Dataset: Create Feature Groups.

Sample Athena Queries

Below we provide some sample queries that act as a template for you to quickly write queries using Athena.

Interactive Exploration

This query selects the first 1000 records.

```
SELECT *
FROM <FeatureGroup.DataCatalogConfig.DatabaseName>.<FeatureGroup.DataCatalogConfig.TableName>
LIMIT 1000
```

Latest snapshot without duplicates

This query selects the latest non-duplicate records.

```
SELECT *
FROM
(SELECT *,
    row_number()
    OVER (PARTITION BY <RecordIdentifierFeatureName>
         ORDER BY <EventTimeFeatureName> desc, Api_Invocation_Time DESC, write_time DESC) AS row_num
FROM
<FeatureGroup.DataCatalogConfig.DatabaseName>.<FeatureGroup.DataCatalogConfig.TableName>)
WHERE row_num = 1;
```

Latest snapshot without duplicates and deleted records in the offline store

This query selects non-duplicate records and delete records from the offline store.

```
SELECT *
FROM
(SELECT *,
    row_number()
    OVER (PARTITION BY <RecordIdentifierFeatureName>
         ORDER BY <EventTimeFeatureName> desc, Api_Invocation_Time DESC, write_time DESC) AS row_num
FROM
<FeatureGroup.DataCatalogConfig.DatabaseName>.<FeatureGroup.DataCatalogConfig.TableName>)
```
Security and Access Control

Amazon SageMaker Feature Store enables you to create two types of stores: an online store or offline store. The online store is used for low latency real-time inference use cases whereas the offline store is used for training and batch inference use cases. When you create a feature group for online or offline use you can provide a KMS customer master key (CMK) to encrypt all your data at rest. In case you do not provide a CMK then we ensure that your data is encrypted on the server side using an AWS owned CMK or AWS managed CMK. While creating a feature group, you can select storage type and optionally provide a CMK for encrypting data, then you can call various APIs for data management such as PutRecord, GetRecord, DeleteRecord.

For more information about CMKs, see Customer Managed Keys. For more information about KMS, please see KMS.

Using KMS Permissions for Amazon SageMaker Feature Store

Encryption at rest protects Feature Store under an AWS KMS customer master key (CMK). By default, it uses an AWS owned CMK for OnlineStore and AWS managed CMK for OfflineStore. Feature Store supports an option to encrypt your online or offline store under customer managed CMKs. You can select the CMK for Feature Store when you create your online or offline store, and they can be different for each store.

Feature Store supports only symmetric CMKs. You cannot use an asymmetric CMK to encrypt your data in your online or offline store. For help determining whether a CMK is symmetric or asymmetric, see Identifying symmetric and asymmetric CMKs.

When you use a customer managed CMK, you can take advantage of the following features:

- You create and manage the CMK, including setting the key policies, IAM policies and grants to control access to the CMK. You can enable and disable the CMK, enable and disable automatic key rotation, and delete the CMK when it is no longer in use.
• You can use a customer managed CMK with imported key material or a customer managed CMK in a custom key store that you own and manage.

• You can audit the encryption and decryption of your online or offline store by examining the API calls to AWS KMS in AWS CloudTrail logs.

You do not pay a monthly fee for AWS owned CMKs. Customer managed CMKs incur a charge for each API call and AWS KMS quotas apply to these CMKs.

Authorizing Use of Your CMK for Your Online Store

If you use a customer managed CMK to protect your online store, the policies on that CMK must give Feature Store permission to use it on your behalf. You have full control over the policies and grants on a customer managed CMK.

Feature Store does not need additional authorization to use the default AWS owned CMK to protect your online or offline stores in your AWS account.

Customer Managed CMK Key Policy

When you select a customer managed CMK to protect your Online Store, Feature Store must have permission to use the CMK on behalf of the principal who makes the selection. That principal, a user or role, must have the permissions on the CMK that Feature Store requires. You can provide these permissions in a key policy, an IAM policy, or a grant. At a minimum, Feature Store requires the following permissions on a customer managed CMK:


For example, the following example key policy provides only the required permissions. The policy has the following effects:

• Allows Feature Store to use the CMK in cryptographic operations and create grants, but only when it is acting on behalf of principals in the account who have permission to use your Feature Store. If the principals specified in the policy statement don't have permission to use your Feature Store, the call fails, even when it comes from the Feature Store service.

• The kms:ViaService condition key allows the permissions only when the request comes from FeatureStore on behalf of the principals listed in the policy statement. These principals can't call these operations directly. The value for kms:ViaService should be sagemaker.*.amazonaws.com.

   Note
   The kms:ViaService condition key can only be used for the online store customer managed KMS key, and cannot be used for the offline store. If you add this special condition to your customer managed key, and use the same KMS key for both the online and offline store, then it will fail the CreateFeatureGroup API operation.

• Gives the CMK administrators read-only access to the CMK and permission to revoke grants, including the grants that Feature Store uses to protect your data.

• Gives Feature Store read-only access to the CMK. In this case, Feature Store can call these operations directly. It does not have to act on behalf of an account principal.

Before using an example key policy, replace the example principals with actual principals from your AWS account.

{"Id": "key-policy-feature-store", "Version":"2012-10-17"}
"Statement": [
{"Sid": "Allow access through Amazon SageMaker Feature Store for all principals in the
account that are authorized to use Amazon SageMaker Feature Store",
"Effect": "Allow",
"Principal": {"AWS": "arn:aws:iam::111122223333:user/featurestore-user"},
"Action": [
"kms:Encrypt",
"kms:Decrypt",
"kms:DescribeKey",
"kms:CreateGrant",
"kms:RetireGrant",
"kms:ReEncryptFrom",
"kms:ReEncryptTo",
"kms:GenerateDataKey",
"kms:ListAliases",
"kms:ListGrants"
],
"Resource": "*",
"Condition": {"StringLike": {"kms:ViaService": "sagemaker.*.amazonaws.com"}}
},
{"Sid": "Allow administrators to view the CMK and revoke grants",
"Effect": "Allow",
"Principal": {"AWS": "arn:aws:iam::111122223333:role/featurestore-admin"},
"Action": [
"kms:Describe*",
"kms:Get*",
"kms:List*",
"kms:RevokeGrant"
],
"Resource": "*
"},
{"Sid": "Allow Feature Store to get information about the CMK",
"Effect": "Allow",
"Principal": {"Service": ["sagemaker.amazonaws.com"]},
"Action": [
"kms:Describe*",
"kms:Get*",
"kms:List*"
],
"Resource": "*
"},
{"Sid": "Enable IAM User Permissions",
"Effect": "Allow",
"Principal": {"AWS": "arn:aws:iam::123456789:root"},
"Action": "kms:*",
"Resource": "*
"}]
}

**Using Grants to Authorize Feature Store**

In addition to key policies, Feature Store uses grants to set permissions on the customer managed CMK.
To view the grants on a CMK in your account, use the ListGrants operation. Feature Store does not need
grants, or any additional permissions, to use the AWS owned CMK to protect your online store.

Feature Store uses the grant permissions when it performs background system maintenance and
continuous data protection tasks.
Each grant is specific to an online store. If the account includes multiple stores encrypted under the same CMK, there will be unique grants per FeatureGroup using the same CMK.

The key policy can also allow the account to revoke the grant on the CMK. However, if you revoke the grant on an active encrypted online store, Feature Store won't be able to protect and maintain the store.

Monitoring Feature Store interaction with AWS KMS

If you use a customer managed CMK to protect your online or offline store, you can use AWS CloudTrail logs to track the requests that Feature Store sends to AWS KMS on your behalf.

Accessing Data in Your Online Store

The caller (either IAM user or IAM role) to ALL DataPlane operations (Put, Get, DeleteRecord) must have below permissions on the customer managed CMK:

"kms:Decrypt"

Authorizing Use of Your CMK for Offline Store

The roleArn that is passed as a parameter to createFeatureGroup must have below permissions to the OfflineStore KmsKeyId:

"kms:GenerateDataKey"

Note
The key policy for the online store also works for the offline store, only when the kms:ViaService condition is not specified.

Amazon SageMaker Feature Store Offline Store

Data Format

Amazon SageMaker Feature Store offline store data is stored in an Amazon S3 bucket within your account. When you call PutRecord, your data is buffered, batched, and written into Amazon S3 within 15 minutes. Feature Store supports both CSV and Parquet file formats. Each file can contain multiple Records.

Files are organized with the following naming convention:


For example:

The following additional fields are added by Feature Store to each `Record` when they persist in the offline store:

- **write_time** – The timestamp when data was written into the offline store. Can be used for constructing time-travel related queries.
- **event_time** – The timestamp provided in the `PutRecord` API call.
- **is_deleted** – `False` by default. If `DeleteRecord` is called, a new `Record` is inserted into `RecordIdentifierValue` and set to `True` in the offline store.

## Amazon SageMaker Feature Store Notebook Examples

To get started using Amazon SageMaker Feature Store, you can use an example Jupyter notebook that demonstrates the key functionalities of Feature Store. This notebook shows how to create and configure a feature group, how to ingest data using `PutRecord` API, shows how the offline data replication is performed, uses data from Feature Store for inference and lastly performs deletion and clean up steps.

To access an example Jupyter notebook that demonstrates key Feature Store capabilities, see [Fraud Detection with Amazon SageMaker Feature Store](#).
Train Models

For an overview on training models with Amazon SageMaker, see Train a Model with Amazon SageMaker (p. 8).

SageMaker provides features to monitor and manage the training and validation of machine learning models. For guidance on metrics available, incremental training, automatic model tuning, and the use of augmented manifest files to label training data, see the following topics.

- For guidance on choosing a machine learning algorithm and its implementation for your task or problem, see Choose an Algorithm (p. 630).
- For guidance on debugging the training of machine learning models, see Amazon SageMaker Debugger (p. 871).
- For guidance on metrics used to monitor and train models, see Monitor and Analyze Training Jobs Using Metrics (p. 1127).
- For guidance on metrics used to detect model post-processing bias, see Detect Posttraining Data and Model Bias (p. 1086).
- For guidance on model explainability, see Model Explainability (p. 1115).
- For guidance on incremental training in SageMaker, see Incremental Training in Amazon SageMaker (p. 1118).
- For guidance on using managed spot training in SageMaker, see Managed Spot Training in Amazon SageMaker (p. 1122).
- For guidance on using training checkpoints in SageMaker, see Use Checkpoints in Amazon SageMaker (p. 1123).
- For guidance on automatic model tuning, also known as hyperparameter tuning, see Perform Automatic Model Tuning (p. 1023).
- For guidance on using an augmented manifest file to label training data, see Provide Dataset Metadata to Training Jobs with an Augmented Manifest File (p. 1124).

Topics

- Choose an Algorithm (p. 630)
- Manage Machine Learning with Amazon SageMaker Experiments (p. 845)
- Amazon SageMaker Debugger (p. 871)
- Perform Automatic Model Tuning (p. 1023)
- Distributed Training (p. 1047)
- Detect Posttraining Data and Model Bias (p. 1086)
- Model Explainability (p. 1115)
- Incremental Training in Amazon SageMaker (p. 1118)
- Managed Spot Training in Amazon SageMaker (p. 1122)
- Use Checkpoints in Amazon SageMaker (p. 1123)
- Provide Dataset Metadata to Training Jobs with an Augmented Manifest File (p. 1124)
- Monitor and Analyze Training Jobs Using Metrics (p. 1127)

Choose an Algorithm

Machine learning can help you accomplish empirical tasks that require some sort of inductive inference. This task involves induction as it uses data to train algorithms to make generalizable inferences. This
means that the algorithms can make statistically reliable predictions or decisions, or complete other
tasks when applied to new data that was not used to train them.

To help you select the best algorithm for your task, we classify these tasks on various levels of
abstraction. At the highest level of abstraction, machine learning attempts to find patterns or
relationships between features or less structured items, such as text in a data set. Pattern recognition
techniques can be classified into distinct machine learning paradigms, each of which address specific
problem types. There are currently three basic paradigms for machine learning used to address various
problem types:

- Supervised learning (p. 633)
- Unsupervised learning (p. 634)
- Reinforcement learning (p. 634)

The types of problems that each learning paradigm can address are identified by considering the
inferences (or predictions, decisions, or other tasks) you want to make from the type of data that you
have or could collect. Machine learning paradigms use algorithmic methods to address their various
problem types. The algorithms provide recipes for solving these problems.

However, many algorithms, such as neural networks, can be deployed with different learning paradigms
and on different types of problems. Multiple algorithms can also address a specific problem type. Some
algorithms are more generally applicable and others are quite specific for certain kinds of objectives and
data. So the mapping between machine learning algorithms and problem types is many-to-many. Also,
there are various implementation options available for algorithms.

The following sections provide guidance concerning implementation options, machine learning
paradigms, and algorithms appropriate for different problem types.

Topics
- Choose an algorithm implementation (p. 631)
- Problem types for the basic machine learning paradigms (p. 633)
- Use Amazon SageMaker Built-in Algorithms (p. 635)
- Use reinforcement learning with Amazon SageMaker (p. 838)

Choose an algorithm implementation

After choosing an algorithm, you must decide which implementation of it you want to use. Amazon
SageMaker supports three implementation options that require increasing levels of effort.

- **Built-in algorithms** require the least effort and scale if the data set is large and significant resources
  are needed to train and deploy the model.
- If there is no built-in solution that works, try to develop one that uses pre-made images for machine
  and deep learning frameworks for supported frameworks such as Scikit-Learn, TensorFlow, PyTorch,
  MXNet, or Chainer.
- If you need to run custom packages or use any code which isn’t a part of a supported framework or
  available via PyPi, then you need to build your own custom Docker image that is configured to install
  the necessary packages or software. The custom image must also be pushed to an online repository
  like the Amazon Elastic Container Service.

Topics
- Use a built-in algorithm (p. 632)
- Use script mode in a supported framework (p. 632)
- Use a custom Docker image (p. 633)
Choose an algorithm implementation

Algorithm implementation guidance

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Requires code</th>
<th>Pre-coded algorithms</th>
<th>Support for third party packages</th>
<th>Support for custom code</th>
<th>Level of effort</th>
</tr>
</thead>
<tbody>
<tr>
<td>Built-in</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Low</td>
</tr>
<tr>
<td>Scikit-learn</td>
<td>Yes</td>
<td>Yes</td>
<td>PyPi only</td>
<td>Yes</td>
<td>Medium</td>
</tr>
<tr>
<td>Spark ML</td>
<td>Yes</td>
<td>Yes</td>
<td>PyPi only</td>
<td>Yes</td>
<td>Medium</td>
</tr>
<tr>
<td>XGBoost (open source)</td>
<td>Yes</td>
<td>Yes</td>
<td>PyPi only</td>
<td>Yes</td>
<td>Medium</td>
</tr>
<tr>
<td>TensorFlow</td>
<td>Yes</td>
<td>No</td>
<td>PyPi only</td>
<td>Yes</td>
<td>Medium-high</td>
</tr>
<tr>
<td>PyTorch</td>
<td>Yes</td>
<td>No</td>
<td>PyPi only</td>
<td>Yes</td>
<td>Medium-high</td>
</tr>
<tr>
<td>MXNet</td>
<td>Yes</td>
<td>No</td>
<td>PyPi only</td>
<td>Yes</td>
<td>Medium-high</td>
</tr>
<tr>
<td>Chainer</td>
<td>Yes</td>
<td>No</td>
<td>PyPi only</td>
<td>Yes</td>
<td>Medium-high</td>
</tr>
<tr>
<td>Custom image</td>
<td>Yes</td>
<td>No</td>
<td>Yes, from any source</td>
<td>Yes</td>
<td>High</td>
</tr>
</tbody>
</table>

Use a built-in algorithm

When choosing an algorithm for your type of problem and data, the easiest option is to use one of Amazon SageMaker's built-in algorithms. These built-in algorithms come with two major benefits.

- The built-in algorithms require no coding to start running experiments. The only inputs you need to provide are the data, hyperparameters, and compute resources. This allows you to run experiments more quickly, with less overhead for tracking results and code changes.
- The built-in algorithms come with parallelization across multiple compute instances and GPU support right out of the box for all applicable algorithms (some algorithms may not be included due to inherent limitations). If you have a lot of data with which to train your model, most built-in algorithms can easily scale to meet the demand. Even if you already have a pre-trained model, it may still be easier to use its corollary in SageMaker and input the hyper-parameters you already know than to port it over, using script mode on a supported framework.

For more information on the built-in algorithms provided by SageMaker, see Use Amazon SageMaker Built-in Algorithms (p. 635).

For important information about docker registry paths, data formats, recommended EC2 instance types, and &CW; logs common to all of the built-in algorithms provided by SageMaker, see Common Information About Built-in Algorithms (p. 640).

Use script mode in a supported framework

If the algorithm you want to use for your model is not supported by a built-in choice and you are comfortable coding your own solution, then you should consider using an Amazon SageMaker supported framework. This is referred to as "script mode" because you write your custom code (script) in a text file with a .py extension. As the table above indicates, SageMaker supports most of the popular machine learning frameworks. These frameworks come preloaded with the corresponding framework and some additional Python packages, such as Pandas and NumPy, so you can write your own code for training an algorithm. These frameworks also allow you to install any Python package hosted on PyPi by including a
requirements.txt file with your training code or to include your own code directories. R is also supported natively in SageMaker notebook kernels. Some frameworks, like scikit-learn and Spark ML, have pre-coded algorithms you can use easily, while other frameworks like TensorFlow and PyTorch may require you to implement the algorithm yourself. The only limitation when using a supported framework image is that you cannot import any software packages that are not hosted on PyPI or that are not already included with the framework’s image.

For more information on the frameworks supported by SageMaker, see Use Machine Learning Frameworks, Python, and R with Amazon SageMaker (p. 17).

Use a custom Docker image

Amazon SageMaker’s built-in algorithms and supported frameworks should cover most use cases, but there are times when you may need to use an algorithm from a package not included in any of the supported frameworks. You may also already have a pre-trained model pickled or persisted somewhere which you need to deploy. SageMaker uses Docker images to host the training and serving of all models, so you can supply your own custom Docker image if the package or software you need is not included in a supported framework. This may be your own Python package or an algorithm coded in a language like Stan or Julia. For these images you must also configure the training of the algorithm and serving of the model properly in your Dockerfile. This requires intermediate knowledge of Docker and is not recommended unless you are comfortable writing your own machine learning algorithm. Your Docker image must be uploaded to an online repository, such as the Amazon Elastic Container Service (ECS) before you can train and serve your model properly.

For more information on custom Docker images in SageMaker, see Using Docker containers with SageMaker (p. 1356).

Problem types for the basic machine learning paradigms

The following three sections describe the main problem types addressed by the three basic paradigms for machine learning. For a list of the built-in algorithms that SageMaker provides to address these problem types, see Use Amazon SageMaker Built-in Algorithms (p. 635).

Topics

• Supervised learning (p. 633)
• Unsupervised learning (p. 634)
• Reinforcement learning (p. 634)

Supervised learning

If your data set consists of features or attributes (inputs) that contain target values (outputs), then you have a supervised learning problem. If your target values are categorical (mathematically discrete), then you have a classification problem. It is a standard practice to distinguish binary from multiclass classification.

• Binary classification is a type of supervised learning that assigns an individual to one of two predefined and mutually exclusive classes based on the individual’s attributes. It is supervised because the models are trained using examples in which the attributes are provided with correctly labeled objects. A medical diagnosis for whether an individual has a disease or not based on the results of diagnostic tests is an example of binary classification.

• Multiclass classification is a type of supervised learning that assigns an individual to one of several classes based on the individual’s attributes. It is supervised because the models are trained using examples in which the attributes are provided with correctly labeled objects. An example is the
prediction of the topic most relevant to a text document. A document may be classified as being about religion, politics, or finance, or as about one of several other predefined topic classes.

If the target values you are trying to predict are mathematically continuous, then you have a **regression** problem. Regression estimates the values of a dependent target variable based on one or more other variables or attributes that are correlated with it. An example is the prediction of house prices using features like the number of bathrooms and bedrooms and the square footage of the house and garden. Regression analysis can create a model that takes one or more of these features as an input and predicts the price of a house.

For more information on the built-in supervised learning algorithms provided by SageMaker, see Supervised Learning (p. 638).

### Unsupervised learning

If your data set consists of features or attributes (inputs) that do not contain labels or target values (outputs), then you have an unsupervised learning problem. In this type of problem, the output must be predicted based on the pattern discovered in the input data. The goal in unsupervised learning problems is to discover patterns such groupings within the data. There are a large variety of tasks or problem types to which unsupervised learning can be applied. Principal component and cluster analyses are two of the main methods commonly deployed for preprocessing data. Here is a short list of problem types that can be addressed by unsupervised learning:

- **Dimension reduction** is typically part of a data exploration step used to determine the most relevant features to use for model construction. The idea is to transform data from a high-dimensional, sparsely populated space into a low-dimensional space that retains most significant properties of the original data. This provides relief for the curse of dimensionality that can arise with sparsely populated, high-dimensional data on which statistical analysis becomes problematic. It can also be used to help understand data, reducing high-dimensional data to a lower dimensionality that can be visualized.

- **Cluster analysis** is a class of techniques that are used to classify objects or cases into groups called clusters. It attempts to find discrete groupings within data, where members of a group are as similar as possible to one another and as different as possible from members of other groups. You define the features or attributes that you want the algorithm to use to determine similarity, select a distance function to measure similarity, and specify the number of clusters to use in the analysis.

- **Anomaly detection** is the identification of rare items, events, or observations in a data set which raise suspicions because they differ significantly from the rest of the data. The identification of anomalous items can be used, for example, to detect bank fraud or medical errors. Anomalies are also referred to as outliers, novelties, noise, deviations, and exceptions.

- **Density estimation** is the construction of estimates of unobservable underlying probability density functions based on observed data. A natural use of density estimates is for data exploration. Density estimates can discover features such as skewness and multimodality in the data. The most basic form of density estimation is a rescaled histogram.

SageMaker provides several built-in machine learning algorithms that you can use for these unsupervised learning tasks. For more information on the built-in unsupervised algorithms provided by SageMaker, see Unsupervised Learning (p. 638).

### Reinforcement learning

Reinforcement learning is a type of learning that is based on interaction with the environment. This type of learning is used by an agent that must learn behavior through trial-and-error interactions with a dynamic environment in which the goal is to maximize the long-term rewards that the agent receives as a result of its actions. Rewards are maximized by trading off exploring actions that have uncertain rewards with exploiting actions that have known rewards.
Use Amazon SageMaker Built-in Algorithms

Amazon SageMaker provides a suite of built-in algorithms to help data scientists and machine learning practitioners get started on training and deploying machine learning models quickly. For someone that is new to SageMaker, choosing the right algorithm for your particular use case can be a challenging task. The following table provides a quick cheat sheet that shows how you can start with an example problem or use case and find an appropriate built-in algorithm offered by SageMaker that is valid for that problem type. Additional guidance organized by learning paradigms (supervised and unsupervised) and important data domains (text and images) is provided in the sections following the table.

Table: Mapping use cases to built-in algorithms

<table>
<thead>
<tr>
<th>Example problems and use cases</th>
<th>Learning paradigm or domain</th>
<th>Problem types</th>
<th>Data input format</th>
<th>Built-in algorithms</th>
</tr>
</thead>
<tbody>
<tr>
<td>Predict if an item belongs to a category: an email spam filter</td>
<td>Supervised Learning (p. 638)</td>
<td>Binary/multi-class classification</td>
<td>Tabular</td>
<td>Factorization Machines Algorithm (p. 681), K-Nearest Neighbors (k-NN) Algorithm (p. 719), Linear Learner Algorithm (p. 732), XGBoost Algorithm (p. 813)</td>
</tr>
<tr>
<td>Predict a numeric/continuous value: estimate the value of a house</td>
<td>Regression</td>
<td>Tabular</td>
<td>Factorization Machines Algorithm (p. 681), K-Nearest Neighbors (k-NN) Algorithm (p. 719), Linear Learner Algorithm (p. 732), XGBoost Algorithm (p. 813)</td>
<td></td>
</tr>
<tr>
<td>Based on historical data for a behavior, predict future behavior: predict sales on a new product based on previous sales data.</td>
<td>Time-series forecasting</td>
<td>Tabular</td>
<td>DeepAR Forecasting Algorithm (p. 666)</td>
<td></td>
</tr>
<tr>
<td>Drop those columns from a dataset that have a weak relation with the label/target variable: the color of a car</td>
<td>Unsupervised Learning (p. 638)</td>
<td>Feature engineering: dimensionality reduction</td>
<td>Tabular</td>
<td>Principal Component Analysis (PCA) Algorithm (p. 780)</td>
</tr>
</tbody>
</table>
Use built-in algorithms

<table>
<thead>
<tr>
<th>Example problems and use cases</th>
<th>Learning paradigm or domain</th>
<th>Problem types</th>
<th>Data input format</th>
<th>Built-in algorithms</th>
</tr>
</thead>
<tbody>
<tr>
<td>when predicting its mileage.</td>
<td></td>
<td>Anomaly detection</td>
<td>Tabular</td>
<td>Random Cut Forest (RCF) Algorithm (p. 785)</td>
</tr>
<tr>
<td>Detect abnormal behavior in application: spot when an IoT sensor is sending abnormal readings</td>
<td></td>
<td>IP anomaly detection</td>
<td>Tabular</td>
<td>IP Insights (p. 701)</td>
</tr>
<tr>
<td>Protect your application from suspicious users: detect if an IP address accessing a service might be from a bad actor</td>
<td></td>
<td>Embeddings: convert high-dimensional objects into low-dimensional space.</td>
<td>Tabular</td>
<td>Object2Vec Algorithm (p. 754)</td>
</tr>
<tr>
<td>Improve the data embeddings of the high-dimensional objects: identify duplicate support tickets or find the correct routing based on similarity of text in the tickets</td>
<td></td>
<td>Clustering or grouping</td>
<td>Tabular</td>
<td>K-Means Algorithm (p. 711)</td>
</tr>
<tr>
<td>Group similar objects/data together: find high-, medium-, and low-spending customers from their transaction histories</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Organize a set of documents into topics (not known in advance): tag a document as belonging to a medical category based on the terms used in the document.</td>
<td></td>
<td>Topic modeling</td>
<td>Text</td>
<td>Latent Dirichlet Allocation (LDA) Algorithm (p. 727), Neural Topic Model (NTM) Algorithm (p. 748)</td>
</tr>
</tbody>
</table>
## Example problems and use cases

<table>
<thead>
<tr>
<th>Learning paradigm or domain</th>
<th>Problem types</th>
<th>Data input format</th>
<th>Built-in algorithms</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assign pre-defined categories to documents in a corpus; categorize books in a library into academic disciplines</td>
<td>Text classification</td>
<td>Text</td>
<td>BlazingText algorithm (p. 656)</td>
</tr>
<tr>
<td>Convert text from one language to other: Spanish to English</td>
<td>Machine translation algorithm</td>
<td>Text</td>
<td>Sequence-to-Sequence Algorithm (p. 801)</td>
</tr>
<tr>
<td>Summarize a long text corpus: an abstract for a research paper</td>
<td>Text summarization</td>
<td>Text</td>
<td>Sequence-to-Sequence Algorithm (p. 801)</td>
</tr>
<tr>
<td>Convert audio files to text: transcribe call center conversations for further analysis</td>
<td>Speech-to-text</td>
<td>Text</td>
<td>Sequence-to-Sequence Algorithm (p. 801)</td>
</tr>
<tr>
<td>Label/tag an image based on the content of the image: alerts about adult content in an image</td>
<td>Image and multi-label classification</td>
<td>Image</td>
<td>Image Classification Algorithm (p. 690)</td>
</tr>
<tr>
<td>Detect people and objects in an image: police review a large photo gallery for a missing person</td>
<td>Object detection and classification</td>
<td>Image</td>
<td>Object Detection Algorithm (p. 770)</td>
</tr>
<tr>
<td>Tag every pixel of an image individually with a category: self-driving cars prepare to identify objects in their way</td>
<td>Computer vision</td>
<td>Image</td>
<td>Semantic Segmentation Algorithm (p. 792)</td>
</tr>
</tbody>
</table>

For important information about Docker registry paths, data formats, recommended Amazon EC2 instance types, and CloudWatch logs common to all of the built-in algorithms provided by SageMaker, see Common Information About Built-in Algorithms (p. 640).
The following sections provide additional guidance for the Amazon SageMaker built-in algorithms grouped by the supervised and unsupervised learning paradigms to which they belong. For descriptions of these learning paradigms and their associated problem types, see Choose an Algorithm (p. 630). Sections are also provided for the SageMaker built-in algorithms available to address two important machine learning domains: textual analysis and image processing.

- Supervised Learning (p. 638)
- Unsupervised Learning (p. 638)
- Textual Analysis (p. 639)
- Image Processing (p. 639)

**Supervised Learning**

Amazon SageMaker provides several built-in general purpose algorithms that can be used for either classification or regression problems.

- **Linear Learner Algorithm** (p. 732)—learns a linear function for regression or a linear threshold function for classification.
- **Factorization Machines Algorithm** (p. 681)—an extension of a linear model that is designed to economically capture interactions between features within high-dimensional sparse datasets.
- **XGBoost Algorithm** (p. 813)—implementation of the gradient-boosted trees algorithm that combines an ensemble of estimates from a set of simpler and weaker models.
- **K-Nearest Neighbors (k-NN) Algorithm** (p. 719)—a non-parametric method that uses the k nearest labeled points to assign a label to a new data point for classification or a predicted target value from the average of the k nearest points for regression.

Amazon SageMaker also provides several built-in supervised learning algorithms that are used for more specialized tasks during feature engineering and forecasting from time series data.

- **Object2Vec Algorithm** (p. 754)—a new highly customizable multi-purpose algorithm used for feature engineering. It can learn low-dimensional dense embeddings of high-dimensional objects to produce features that improve training efficiencies for downstream models. While this is a supervised algorithm, as it requires labeled data for training, there are many scenarios in which the relationship labels can be obtained purely from natural clusterings in data, without any explicit human annotation.
- **DeepAR Forecasting Algorithm** (p. 666)—a supervised learning algorithm for forecasting scalar (one-dimensional) time series using recurrent neural networks (RNN).

**Unsupervised Learning**

Amazon SageMaker provides several built-in algorithms that can be used for a variety of unsupervised learning tasks such as clustering, dimension reduction, pattern recognition, and anomaly detection.

- **Principal Component Analysis (PCA) Algorithm** (p. 780)—reduces the dimensionality (number of features) within a dataset by projecting data points onto the first few principal components. The objective is to retain as much information or variation as possible. For mathematicians, principal components are eigenvectors of the data’s covariance matrix.
- **K-Means Algorithm** (p. 711)—finds discrete groupings within data, where members of a group are as similar as possible to one another and as different as possible from members of other groups.
- **IP Insights** (p. 701)—learns the usage patterns for IPv4 addresses. It is designed to capture associations between IPv4 addresses and various entities, such as user IDs or account numbers.
- **Random Cut Forest (RCF) Algorithm** (p. 785)—detects anomalous data points within a data set that diverge from otherwise well-structured or patterned data.
Textual Analysis

SageMaker provides algorithms that are tailored to the analysis of textual documents used in natural language processing, document classification or summarization, topic modeling or classification, and language transcription or translation.

- **BlazingText algorithm (p. 656)**—a highly optimized implementation of the Word2vec and text classification algorithms that scale to large datasets easily. It is useful for many downstream natural language processing (NLP) tasks.
- **Sequence-to-Sequence Algorithm (p. 801)**—a supervised algorithm commonly used for neural machine translation.
- **Latent Dirichlet Allocation (LDA) Algorithm (p. 727)**—an algorithm suitable for determining topics in a set of documents. It is an unsupervised algorithm, which means that it doesn't use example data with answers during training.
- **Neural Topic Model (NTM) Algorithm (p. 748)**—another unsupervised technique for determining topics in a set of documents, using a neural network approach.

Image Processing

SageMaker also provides image processing algorithms that are used for image classification, object detection, and computer vision.

- **Image Classification Algorithm (p. 690)**—uses example data with answers (referred to as a supervised algorithm). Use this algorithm to classify images.
- **Semantic Segmentation Algorithm (p. 792)**—provides a fine-grained, pixel-level approach to developing computer vision applications.
- **Object Detection Algorithm (p. 770)**—detects and classifies objects in images using a single deep neural network. It is a supervised learning algorithm that takes images as input and identifies all instances of objects within the image scene.

Topics

- Common Information About Built-in Algorithms (p. 640)
- BlazingText algorithm (p. 656)
- DeepAR Forecasting Algorithm (p. 666)
- Factorization Machines Algorithm (p. 681)
- Image Classification Algorithm (p. 690)
- IP Insights (p. 701)
- K-Means Algorithm (p. 711)
- K-Nearest Neighbors (k-NN) Algorithm (p. 719)
- Latent Dirichlet Allocation (LDA) Algorithm (p. 727)
- Linear Learner Algorithm (p. 732)
- Neural Topic Model (NTM) Algorithm (p. 748)
- Object2Vec Algorithm (p. 754)
- Object Detection Algorithm (p. 770)
- Principal Component Analysis (PCA) Algorithm (p. 780)
- Random Cut Forest (RCF) Algorithm (p. 785)
- Semantic Segmentation Algorithm (p. 792)
- Sequence-to-Sequence Algorithm (p. 801)
- XGBoost Algorithm (p. 813)
Common Information About Built-in Algorithms

The following topics provide information about Docker registry paths, data formats, recommended Amazon EC2 instance types, and CloudWatch logs common to all of the built-in algorithms provided by Amazon SageMaker.

Topics
- Docker Registry Paths for SageMaker Built-in Algorithms (p. 640)
- Common Data Formats for Built-in Algorithms (p. 646)
- Instance Types for Built-in Algorithms (p. 655)
- Logs for Built-in Algorithms (p. 655)

Docker Registry Paths for SageMaker Built-in Algorithms

The following table lists parameters for each of the algorithms provided by Amazon SageMaker.

<table>
<thead>
<tr>
<th>Algorithm name</th>
<th>Channel name</th>
<th>Training image and inference image registry path</th>
<th>Training input mode</th>
<th>File type</th>
<th>Instance class</th>
<th>Parallelizable</th>
</tr>
</thead>
<tbody>
<tr>
<td>BlazingText</td>
<td>Train</td>
<td><code>&lt;ecr_path&gt;/blazingtext:&lt;tag&gt;</code></td>
<td>File or Pipe</td>
<td>Text file (one sentence per line with space-separated tokens)</td>
<td>GPU (single instance only) or CPU</td>
<td>No</td>
</tr>
<tr>
<td>DeepAR Forecasting</td>
<td>Train and (optionally) test</td>
<td><code>&lt;ecr_path&gt;/forecasting-deepar:&lt;tag&gt;</code></td>
<td>File</td>
<td>JSON Lines or Parquet</td>
<td>GPU or CPU</td>
<td>Yes</td>
</tr>
<tr>
<td>Factorization Machines</td>
<td>Train and (optionally) test</td>
<td><code>&lt;ecr_path&gt;/factorization-machines:&lt;tag&gt;</code></td>
<td>File or Pipe</td>
<td>recordIO-protobuf</td>
<td>CPU (GPU for dense data)</td>
<td>Yes</td>
</tr>
<tr>
<td>Image Classification</td>
<td>Train and validation, (optionally) train_lst, validation_lst, and model</td>
<td><code>&lt;ecr_path&gt;/image-classification:&lt;tag&gt;</code></td>
<td>File or Pipe</td>
<td>recordIO or image files (.jpg or .png)</td>
<td>GPU</td>
<td>Yes</td>
</tr>
<tr>
<td>IP Insights</td>
<td>Train and (optionally) validation</td>
<td><code>&lt;ecr_path&gt;/ipinsights:&lt;tag&gt;</code></td>
<td>File</td>
<td>CSV</td>
<td>CPU or GPU</td>
<td>Yes</td>
</tr>
<tr>
<td>k-means</td>
<td>Train and (optionally) test</td>
<td><code>&lt;ecr_path&gt;/kmeans:&lt;tag&gt;</code></td>
<td>File or Pipe</td>
<td>recordIO-protobuf or CSV</td>
<td>CPU or GPUCommon (single GPU device on one)</td>
<td>No</td>
</tr>
<tr>
<td>Algorithm name</td>
<td>Channel name</td>
<td>Training image and inference image registry path</td>
<td>Training input mode</td>
<td>File type</td>
<td>Instance class</td>
<td>Parallelizable</td>
</tr>
<tr>
<td>------------------------------</td>
<td>----------------------------</td>
<td>--------------------------------------------------</td>
<td>---------------------</td>
<td>----------------------------</td>
<td>----------------</td>
<td>----------------</td>
</tr>
<tr>
<td>k-nearest-neighbor (k-NN)</td>
<td>Train and (optionally) test</td>
<td><code>&lt;ecr_path&gt;/knn:&lt;tag&gt;</code></td>
<td>File or Pipe</td>
<td>recordIO-protobuf or CSV</td>
<td>CPU or GPU (single GPU device on one or more instances)</td>
<td>Yes</td>
</tr>
<tr>
<td>LDA</td>
<td>Train and (optionally) test</td>
<td><code>&lt;ecr_path&gt;/lda:&lt;tag&gt;</code></td>
<td>File or Pipe</td>
<td>recordIO-protobuf or CSV</td>
<td>CPU (single instance only)</td>
<td>No</td>
</tr>
<tr>
<td>Linear Learner</td>
<td>Train and (optionally) validation, test, or both</td>
<td><code>&lt;ecr_path&gt;/linear-learner:&lt;tag&gt;</code></td>
<td>File or Pipe</td>
<td>recordIO-protobuf or CSV</td>
<td>CPU or GPU</td>
<td>Yes</td>
</tr>
<tr>
<td>Neural Topic Model</td>
<td>Train and (optionally) validation, test, or both</td>
<td><code>&lt;ecr_path&gt;/ntm:&lt;tag&gt;</code></td>
<td>File or Pipe</td>
<td>recordIO-protobuf or CSV</td>
<td>GPU or CPU</td>
<td>Yes</td>
</tr>
<tr>
<td>Object2Vec</td>
<td>Train and (optionally) validation, test, or both</td>
<td><code>&lt;ecr_path&gt;/object2vec:&lt;tag&gt;</code></td>
<td>File</td>
<td>JSON Lines</td>
<td>GPU or CPU (single instance only)</td>
<td>No</td>
</tr>
<tr>
<td>Object Detection</td>
<td>Train and validation, (optionally) train annotation, validation annotation, and model</td>
<td><code>&lt;ecr_path&gt;/object-detection:&lt;tag&gt;</code></td>
<td>File or Pipe</td>
<td>recordIO or image files (.jpg or .png)</td>
<td>GPU</td>
<td>Yes</td>
</tr>
<tr>
<td>PCA</td>
<td>Train and (optionally) test</td>
<td><code>&lt;ecr_path&gt;/pca:&lt;tag&gt;</code></td>
<td>File or Pipe</td>
<td>recordIO-protobuf or CSV</td>
<td>GPU or CPU</td>
<td>Yes</td>
</tr>
<tr>
<td>Random Cut Forest</td>
<td>Train and (optionally) test</td>
<td><code>&lt;ecr_path&gt;/randomcutforest:&lt;tag&gt;</code></td>
<td>File or Pipe</td>
<td>recordIO-protobuf or CSV</td>
<td>CPU</td>
<td>Yes</td>
</tr>
</tbody>
</table>
## Use built-in algorithms

<table>
<thead>
<tr>
<th>Algorithm name</th>
<th>Channel name</th>
<th>Training image and inference image registry path</th>
<th>Training input mode</th>
<th>File type</th>
<th>Instance class</th>
<th>Parallelizable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Semantic Segmentation</td>
<td>Train and validation, train_annotation, validation_annotation, and (optionally) label_map and model</td>
<td>&lt;ecr_path&gt;/semantic-segmentation:&lt;tag&gt;</td>
<td>File or Pipe</td>
<td>Image files</td>
<td>GPU (single instance only)</td>
<td>No</td>
</tr>
<tr>
<td>Seq2Seq Modeling</td>
<td>Train, validation, and vocab</td>
<td>&lt;ecr_path&gt;/seq2seq:&lt;tag&gt;</td>
<td>File</td>
<td>recordIO-protobuf</td>
<td>GPU (single instance only)</td>
<td>No</td>
</tr>
<tr>
<td>XGBoost (0.90-1, 0.90-2, 1.0-1, 1.2-1)</td>
<td>Train and (optionally) validation</td>
<td>&lt;ecr_path&gt;/sagemaker-xgboost:1.0-1-cpu-py3 &lt;ecr_path&gt;/sagemaker-xgboost:1.2-1</td>
<td>File or Pipe</td>
<td>CSV, LibSVM, recordIO-protobuf, or Parquet</td>
<td>CPU (or GPU for 1.2-1)</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Note**

For XGBoost, do not use :latest or :1. Use the specific version you require, such as :0.90-1-cpu-py3, :0.90-2-cpu-py3, :1.0-1-cpu-py3, or :1.2-1.

Algorithms that are *parallelizable* can be deployed on multiple compute instances for distributed training. For the **Training Image and Inference Image Registry Path** column, use the :1 version tag to ensure that you are using a stable version of the algorithm. You can reliably host a model trained using an image with the :1 tag on an inference image that has the :1 tag. Using the :latest tag in the registry path provides you with the most up-to-date version of the algorithm, but might cause problems with backward compatibility. Avoid using the :latest tag for production purposes.

For the **Training Image and Inference Image Registry Path** column, depending on algorithm and region, use one of the following values for `<ecr_path>`.

**Algorithms:** BlazingText, Image Classification, Object Detection, Semantic Segmentation, and Seq2Seq

<table>
<thead>
<tr>
<th>AWS Region</th>
<th>Training image and inference image registry path</th>
</tr>
</thead>
<tbody>
<tr>
<td>us-west-1</td>
<td>632565934929.dkr.ecr.us-west-1.amazonaws.com</td>
</tr>
<tr>
<td>us-west-2</td>
<td>433757028032.dkr.ecr.us-west-2.amazonaws.com</td>
</tr>
<tr>
<td>us-east-1</td>
<td>811284229777.dkr.ecr.us-east-1.amazonaws.com</td>
</tr>
<tr>
<td>us-east-2</td>
<td>825641698319.dkr.ecr.us-east-2.amazonaws.com</td>
</tr>
<tr>
<td>ap-east-1</td>
<td>286214385809.dkr.ecr.ap-east-1.amazonaws.com</td>
</tr>
<tr>
<td>ap-northeast-1</td>
<td>501404015308.dkr.ecr.ap-northeast-1.amazonaws.com</td>
</tr>
<tr>
<td>ap-northeast-2</td>
<td>306986355934.dkr.ecr.ap-northeast-2.amazonaws.com</td>
</tr>
</tbody>
</table>
### Use built-in algorithms

<table>
<thead>
<tr>
<th>AWS Region</th>
<th>Training image and inference image registry path</th>
</tr>
</thead>
<tbody>
<tr>
<td>ap-south-1</td>
<td>991648021394.dkr.ecr.ap-south-1.amazonaws.com</td>
</tr>
<tr>
<td>ap-southeast-1</td>
<td>475088953585.dkr.ecr.ap-southeast-1.amazonaws.com</td>
</tr>
<tr>
<td>ap-southeast-2</td>
<td>544295431143.dkr.ecr.ap-southeast-2.amazonaws.com</td>
</tr>
<tr>
<td>ca-central-1</td>
<td>469771592824.dkr.ecr.ca-central-1.amazonaws.com</td>
</tr>
<tr>
<td>cn-north-1</td>
<td>390948362332.dkr.ecr.cn-north-1.amazonaws.com</td>
</tr>
<tr>
<td>cn-northwest-1</td>
<td>387376663083.dkr.ecr.cn-northwest-1.amazonaws.com</td>
</tr>
<tr>
<td>eu-central-1</td>
<td>813361260812.dkr.ecr.eu-central-1.amazonaws.com</td>
</tr>
<tr>
<td>eu-north-1</td>
<td>669576153137.dkr.ecr.eu-north-1.amazonaws.com</td>
</tr>
<tr>
<td>eu-west-1</td>
<td>685385470294.dkr.ecr.eu-west-1.amazonaws.com</td>
</tr>
<tr>
<td>eu-west-2</td>
<td>644912444149.dkr.ecr.eu-west-2.amazonaws.com</td>
</tr>
<tr>
<td>eu-west-3</td>
<td>749696950732.dkr.ecr.eu-west-3.amazonaws.com</td>
</tr>
<tr>
<td>me-south-1</td>
<td>249704162688.dkr.ecr.me-south-1.amazonaws.com</td>
</tr>
<tr>
<td>sa-east-1</td>
<td>855470959533.dkr.ecr.sa-east-1.amazonaws.com</td>
</tr>
<tr>
<td>us-gov-west-1</td>
<td>226302683700.dkr.ecr.us-gov-west-1.amazonaws.com</td>
</tr>
</tbody>
</table>

**Algorithms:** DeepAR Forecasting

<table>
<thead>
<tr>
<th>AWS Region</th>
<th>Training image and inference image registry path</th>
</tr>
</thead>
<tbody>
<tr>
<td>us-west-1</td>
<td>632365934929.dkr.ecr.us-west-1.amazonaws.com</td>
</tr>
<tr>
<td>us-west-2</td>
<td>156387875391.dkr.ecr.us-west-2.amazonaws.com</td>
</tr>
<tr>
<td>us-east-1</td>
<td>522234722520.dkr.ecr.us-east-1.amazonaws.com</td>
</tr>
<tr>
<td>us-east-2</td>
<td>566113047672.dkr.ecr.us-east-2.amazonaws.com</td>
</tr>
<tr>
<td>ap-east-1</td>
<td>286214385809.dkr.ecr.ap-east-1.amazonaws.com</td>
</tr>
<tr>
<td>ap-northeast-1</td>
<td>633353088612.dkr.ecr.ap-northeast-1.amazonaws.com</td>
</tr>
<tr>
<td>ap-northeast-2</td>
<td>204372634319.dkr.ecr.ap-northeast-2.amazonaws.com</td>
</tr>
<tr>
<td>ap-south-1</td>
<td>991648021394.dkr.ecr.ap-south-1.amazonaws.com</td>
</tr>
<tr>
<td>ap-southeast-1</td>
<td>475088953585.dkr.ecr.ap-southeast-1.amazonaws.com</td>
</tr>
<tr>
<td>ap-southeast-2</td>
<td>514117268639.dkr.ecr.ap-southeast-2.amazonaws.com</td>
</tr>
<tr>
<td>ca-central-1</td>
<td>469771592824.dkr.ecr.ca-central-1.amazonaws.com</td>
</tr>
<tr>
<td>cn-north-1</td>
<td>390948362332.dkr.ecr.cn-north-1.amazonaws.com</td>
</tr>
<tr>
<td>cn-northwest-1</td>
<td>387376663083.dkr.ecr.cn-northwest-1.amazonaws.com</td>
</tr>
<tr>
<td>eu-north-1</td>
<td>669576153137.dkr.ecr.eu-north-1.amazonaws.com</td>
</tr>
</tbody>
</table>
### Use built-in algorithms

#### Training image and inference image registry path

<table>
<thead>
<tr>
<th>AWS Region</th>
<th>Training image and inference image registry path</th>
</tr>
</thead>
<tbody>
<tr>
<td>eu-central-1</td>
<td>495149712605.dkr.ecr.eu-central-1.amazonaws.com</td>
</tr>
<tr>
<td>eu-west-1</td>
<td>224300973850.dkr.ecr.eu-west-1.amazonaws.com</td>
</tr>
<tr>
<td>eu-west-2</td>
<td>644912444149.dkr.ecr.eu-west-2.amazonaws.com</td>
</tr>
<tr>
<td>eu-west-3</td>
<td>749696950732.dkr.ecr.eu-west-3.amazonaws.com</td>
</tr>
<tr>
<td>me-south-1</td>
<td>249704162688.dkr.ecr.me-south-1.amazonaws.com</td>
</tr>
<tr>
<td>sa-east-1</td>
<td>855470959533.dkr.ecr.sa-east-1.amazonaws.com</td>
</tr>
<tr>
<td>us-gov-west-1</td>
<td>226302683700.dkr.ecr.us-gov-west-1.amazonaws.com</td>
</tr>
<tr>
<td>us-west-1</td>
<td>632365934929.dkr.ecr.us-west-1.amazonaws.com</td>
</tr>
<tr>
<td>us-west-2</td>
<td>174872318107.dkr.ecr.us-west-2.amazonaws.com</td>
</tr>
<tr>
<td>us-east-1</td>
<td>382416733822.dkr.ecr.us-east-1.amazonaws.com</td>
</tr>
<tr>
<td>us-east-2</td>
<td>404615174143.dkr.ecr.us-east-2.amazonaws.com</td>
</tr>
<tr>
<td>ap-east-1</td>
<td>286214385809.dkr.ecr.ap-east-1.amazonaws.com</td>
</tr>
<tr>
<td>ap-northeast-1</td>
<td>351501993468.dkr.ecr.ap-northeast-1.amazonaws.com</td>
</tr>
<tr>
<td>ap-northeast-2</td>
<td>835164637446.dkr.ecr.ap-northeast-2.amazonaws.com</td>
</tr>
<tr>
<td>ap-south-1</td>
<td>991648021394.dkr.ecr.ap-south-1.amazonaws.com</td>
</tr>
<tr>
<td>ap-southeast-1</td>
<td>475088953585.dkr.ecr.ap-southeast-1.amazonaws.com</td>
</tr>
<tr>
<td>ap-southeast-2</td>
<td>712309505854.dkr.ecr.ap-southeast-2.amazonaws.com</td>
</tr>
<tr>
<td>ca-central-1</td>
<td>469771592824.dkr.ecr.ca-central-1.amazonaws.com</td>
</tr>
<tr>
<td>cn-north-1</td>
<td>390948362332.dkr.ecr.cn-north-1.amazonaws.com</td>
</tr>
<tr>
<td>cn-northwest-1</td>
<td>387376663083.dkr.ecr.cn-northwest-1.amazonaws.com</td>
</tr>
<tr>
<td>eu-central-1</td>
<td>664544806723.dkr.ecr.eu-central-1.amazonaws.com</td>
</tr>
<tr>
<td>eu-north-1</td>
<td>669576153137.dkr.ecr.eu-north-1.amazonaws.com</td>
</tr>
<tr>
<td>eu-west-1</td>
<td>438346466558.dkr.ecr.eu-west-1.amazonaws.com</td>
</tr>
<tr>
<td>eu-west-2</td>
<td>644912444149.dkr.ecr.eu-west-2.amazonaws.com</td>
</tr>
<tr>
<td>eu-west-3</td>
<td>749696950732.dkr.ecr.eu-west-3.amazonaws.com</td>
</tr>
<tr>
<td>me-south-1</td>
<td>249704162688.dkr.ecr.me-south-1.amazonaws.com</td>
</tr>
<tr>
<td>sa-east-1</td>
<td>855470959533.dkr.ecr.sa-east-1.amazonaws.com</td>
</tr>
</tbody>
</table>

### Algorithms: Factorization Machines, IP Insights, k-means, k-nearest-neighbor, Linear Learner, Object2Vec, Neural Topic Model, PCA, and Random Cut Forest
### Algorithms: Latent Dirichlet Allocation (LDA)

<table>
<thead>
<tr>
<th>AWS Region</th>
<th>Training image and inference image registry path</th>
</tr>
</thead>
<tbody>
<tr>
<td>us-west-1</td>
<td>632365934929.dkr.ecr.us-west-1.amazonaws.com</td>
</tr>
<tr>
<td>us-west-2</td>
<td>266724342769.dkr.ecr.us-west-2.amazonaws.com</td>
</tr>
<tr>
<td>us-east-1</td>
<td>766337827248.dkr.ecr.us-east-1.amazonaws.com</td>
</tr>
<tr>
<td>us-east-2</td>
<td>999911452149.dkr.ecr.us-east-2.amazonaws.com</td>
</tr>
<tr>
<td>ap-northeast-1</td>
<td>258307448986.dkr.ecr.ap-northeast-1.amazonaws.com</td>
</tr>
<tr>
<td>ap-northeast-2</td>
<td>293181348795.dkr.ecr.ap-northeast-2.amazonaws.com</td>
</tr>
<tr>
<td>ap-south-1</td>
<td>991648021394.dkr.ecr.ap-south-1.amazonaws.com</td>
</tr>
<tr>
<td>ap-southeast-1</td>
<td>475088953585.dkr.ecr.ap-southeast-1.amazonaws.com</td>
</tr>
<tr>
<td>ap-southeast-2</td>
<td>297031611018.dkr.ecr.ap-southeast-2.amazonaws.com</td>
</tr>
<tr>
<td>ca-central-1</td>
<td>469771592824.dkr.ecr.ca-central-1.amazonaws.com</td>
</tr>
<tr>
<td>eu-central-1</td>
<td>353608530281.dkr.ecr.eu-central-1.amazonaws.com</td>
</tr>
<tr>
<td>eu-west-1</td>
<td>999678624901.dkr.ecr.eu-west-1.amazonaws.com</td>
</tr>
<tr>
<td>eu-west-2</td>
<td>644912444149.dkr.ecr.eu-west-2.amazonaws.com</td>
</tr>
<tr>
<td>us-gov-west-1</td>
<td>226302683700.dkr.ecr.us-gov-west-1.amazonaws.com</td>
</tr>
</tbody>
</table>

### Algorithms: XGBoost (0.90-1, 0.90-2, 1.0-1, 1.2-1)

<table>
<thead>
<tr>
<th>AWS Region</th>
<th>Training image and inference image registry path</th>
</tr>
</thead>
<tbody>
<tr>
<td>us-west-1</td>
<td>746614075791.dkr.ecr.us-west-1.amazonaws.com</td>
</tr>
<tr>
<td>us-west-2</td>
<td>246618743249.dkr.ecr.us-west-2.amazonaws.com</td>
</tr>
<tr>
<td>us-east-1</td>
<td>683313688378.dkr.ecr.us-east-1.amazonaws.com</td>
</tr>
<tr>
<td>us-east-2</td>
<td>257758044811.dkr.ecr.us-east-2.amazonaws.com</td>
</tr>
<tr>
<td>ap-northeast-1</td>
<td>354813040037.dkr.ecr.ap-northeast-1.amazonaws.com</td>
</tr>
<tr>
<td>ap-northeast-2</td>
<td>366743142698.dkr.ecr.ap-northeast-2.amazonaws.com</td>
</tr>
<tr>
<td>ap-southeast-1</td>
<td>121021644041.dkr.ecr.ap-southeast-1.amazonaws.com</td>
</tr>
<tr>
<td>ap-southeast-2</td>
<td>783357654285.dkr.ecr.ap-southeast-2.amazonaws.com</td>
</tr>
<tr>
<td>ap-south-1</td>
<td>720646828776.dkr.ecr.ap-south-1.amazonaws.com</td>
</tr>
<tr>
<td>ap-east-1</td>
<td>651117190479.dkr.ecr.ap-east-1.amazonaws.com</td>
</tr>
</tbody>
</table>
Use built-in algorithms

Use the paths and training input mode as follows:

• To create a training job (with a request to the `CreateTrainingJob` API), specify the Docker registry path and the training input mode for the training image. You create a training job to train a model using a specific dataset.

• To create a model (with a `CreateModel` request), specify the Docker registry path for the inference image. Amazon SageMaker launches machine learning compute instances that are based on the endpoint configuration and deploys the model, which includes the artifacts (the result of model training).

### Common Data Formats for Built-in Algorithms

The following topics explain the data formats for the algorithms provided by Amazon SageMaker.

#### Topics

- Common Data Formats for Training (p. 646)
- Common Data Formats for Inference (p. 650)

#### Common Data Formats for Training

To prepare for training, you can preprocess your data using a variety of AWS services, including AWS Glue, Amazon EMR, Amazon Redshift, Amazon Relational Database Service, and Amazon Athena. After preprocessing, publish the data to an Amazon S3 bucket. For training, the data need to go through a series of conversions and transformations, including:

- Training data serialization (handled by you)
- Training data deserialization (handled by the algorithm)
- Training model serialization (handled by the algorithm)
- Trained model deserialization (optional, handled by you)

<table>
<thead>
<tr>
<th>AWS Region</th>
<th>Training image and inference image registry path</th>
</tr>
</thead>
<tbody>
<tr>
<td>ca-central-1</td>
<td>341280168497.dkr.ecr.ca-central-1.amazonaws.com</td>
</tr>
<tr>
<td>cn-north-1</td>
<td>450853457545.dkr.ecr.cn-north-1.amazonaws.com.cn</td>
</tr>
<tr>
<td>cn-northwest-1</td>
<td>451049120500.dkr.ecr.cn-northwest-1.amazonaws.com.cn</td>
</tr>
<tr>
<td>eu-central-1</td>
<td>492215442770.dkr.ecr.eu-central-1.amazonaws.com</td>
</tr>
<tr>
<td>eu-north-1</td>
<td>662702820516.dkr.ecr.eu-north-1.amazonaws.com</td>
</tr>
<tr>
<td>eu-west-1</td>
<td>141502667606.dkr.ecr.eu-west-1.amazonaws.com</td>
</tr>
<tr>
<td>eu-west-2</td>
<td>764974769150.dkr.ecr.eu-west-2.amazonaws.com</td>
</tr>
<tr>
<td>eu-west-3</td>
<td>659782779980.dkr.ecr.eu-west-3.amazonaws.com</td>
</tr>
<tr>
<td>me-south-1</td>
<td>801668240914.dkr.ecr.me-south-1.amazonaws.com</td>
</tr>
<tr>
<td>sa-east-1</td>
<td>737474898029.dkr.ecr.sa-east-1.amazonaws.com</td>
</tr>
<tr>
<td>us-gov-west-1</td>
<td>414596584902.dkr.ecr.us-gov-west-1.amazonaws.com</td>
</tr>
</tbody>
</table>
When using Amazon SageMaker in the training portion of the algorithm, make sure to upload all data at once. If more data is added to that location, a new training call would need to be made to construct a brand new model.

**Topics**
- Content Types Supported by Built-In Algorithms (p. 647)
- Using CSV Format (p. 647)
- Using RecordIO Format (p. 647)
- Trained Model Deserialization (p. 650)

**Content Types Supported by Built-In Algorithms**

The following table lists some of the commonly supported `ContentType` values and the algorithms that use them:

### ContentTypes for Built-in Algorithms

<table>
<thead>
<tr>
<th>ContentType</th>
<th>Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>application/x-image</td>
<td>Object Detection Algorithm, Semantic Segmentation</td>
</tr>
<tr>
<td>application/x-recordio</td>
<td>Object Detection Algorithm</td>
</tr>
<tr>
<td>application/x-recordio-protobuf</td>
<td>Factorization Machines, K-Means, k-NN, Latent Dirichlet Allocation, Linear Learner, NTM, PCA, RCF, Sequence-to-Sequence</td>
</tr>
<tr>
<td>application/jsonlines</td>
<td>BlazingText, DeepAR</td>
</tr>
<tr>
<td>image/jpeg</td>
<td>Object Detection Algorithm, Semantic Segmentation</td>
</tr>
<tr>
<td>image/png</td>
<td>Object Detection Algorithm, Semantic Segmentation</td>
</tr>
<tr>
<td>text/libsvm</td>
<td>XGBoost</td>
</tr>
</tbody>
</table>

For a summary of the parameters used by each algorithm, see the documentation for the individual algorithms or this table.

**Using CSV Format**

Many Amazon SageMaker algorithms support training with data in CSV format. To use data in CSV format for training, in the input data channel specification, specify `text/csv` as the `ContentType`. Amazon SageMaker requires that a CSV file does not have a header record and that the target variable is in the first column. To run unsupervised learning algorithms that don’t have a target, specify the number of label columns in the content type. For example, in this case `'content_type=text/csv;label_size=0'`.

**Using RecordIO Format**

Most Amazon SageMaker algorithms work best when you use the optimized protobuf `recordIO` data format for training. Using this format allows you to take advantage of Pipe mode. In Pipe mode, your training job streams data directly from Amazon Simple Storage Service (Amazon S3). Streaming can provide faster start times for training jobs and better throughput. This is in contrast to File mode, in which your data from Amazon S3 is stored on the training instance volumes. File mode uses disk space to store both your final model artifacts and your full training dataset. By streaming in your data directly from Amazon S3 in Pipe mode, you reduce the size of Amazon Elastic Block Store volumes of your
training instances. Pipe mode needs only enough disk space to store your final model artifacts. See the AlgorithmSpecification for additional details on the training input mode.

**Note**

For an example that shows how to convert the commonly used numPy array into the protobuf recordIO format, see An Introduction to Factorization Machines with MNIST.

In the protobuf recordIO format, SageMaker converts each observation in the dataset into a binary representation as a set of 4-byte floats, then loads it in the protobuf values field. If you are using Python for your data preparation, we strongly recommend that you use these existing transformations. However, if you are using another language, the protobuf definition file below provides the schema that you use to convert your data into SageMaker protobuf format.

```protobuf
syntax = "proto2";

package aialgs.data;

option java_package = "com.amazonaws.aialgorithms.proto";
option java_outer_classname = "RecordProtos";

message Float32Tensor {
  // Each value in the vector. If keys is empty, this is treated as a dense vector.
  repeated float values = 1 [packed = true];

  // If key is not empty, the vector is treated as sparse, with each key specifying the location of the value in the sparse vector.
  repeated uint64 keys = 2 [packed = true];

  // An optional shape that allows the vector to represent a matrix.
  // For example, if shape = [10, 20], floor(keys[i] / 20) gives the row, and keys[i] % 20 gives the column.
  // This also supports n-dimensional tensors.
  // Note: If the tensor is sparse, you must specify this value.
  repeated uint64 shape = 3 [packed = true];
}

message Float64Tensor {
  // Each value in the vector. If keys is empty, this is treated as a dense vector.
  repeated double values = 1 [packed = true];

  // If this is not empty, the vector is treated as sparse, with each key specifying the location of the value in the sparse vector.
  repeated uint64 keys = 2 [packed = true];

  // An optional shape that allows the vector to represent a matrix.
  // For example, if shape = [10, 20], floor(keys[i] / 20) gives the row, and keys[i] % 20 gives the column.
  // This also supports n-dimensional tensors.
  // Note: If the tensor is sparse, you must specify this value.
  repeated uint64 shape = 3 [packed = true];
}

message Int32Tensor {
  // Each value in the vector. If keys is empty, this is treated as a dense vector.
  repeated int32 values = 1 [packed = true];

  // If this is not empty, the vector is treated as sparse, with each key specifying the location of the value in the sparse vector.
  repeated uint64 keys = 2 [packed = true];

  // An optional shape that allows the vector to represent a matrix.
  // For example, if shape = [10, 20], floor(keys[i] / 20) gives the row, and keys[i] % 20 gives the column.
  // This also supports n-dimensional tensors.
  // Note: If the tensor is sparse, you must specify this value.
  repeated uint64 shape = 3 [packed = true];
}
```

648
repeated uint64 keys = 2 [packed = true];

// An optional shape that allows the vector to represent a matrix.
// For example, if shape = [10, 20], floor(keys[i] / 10) gives the row,
// and keys[i] % 20 gives the column.
// This also supports n-dimensional tensors.
// Note: If the tensor is sparse, you must specify this value.
repeated uint64 shape = 3 [packed = true];

// Support for storing binary data for parsing in other ways (such as JPEG/etc).
// This is an example of another type of value and may not immediately be supported.
message Bytes {
  repeated bytes value = 1;

  // If the content type of the data is known, stores it.
  // This allows for the possibility of using decoders for common formats
  // in the future.
  optional string content_type = 2;
}

message Value {
  oneof value {
    // The numbering assumes the possible use of:
    // - float16, float128
    // - int8, int16, int32
    Float32Tensor float32_tensor = 2;
    Float64Tensor float64_tensor = 3;
    Int32Tensor int32_tensor = 7;
    Bytes bytes = 9;
  }
}

message Record {
  // Map from the name of the feature to the value.
  // For vectors and libsvm-like datasets,
  // a single feature with the name `values`
  // should be specified.
  map<string, Value> features = 1;

  // An optional set of labels for this record.
  // Similar to the features field above, the key used for
  // generic scalar / vector labels should be 'values'.
  map<string, Value> label = 2;

  // A unique identifier for this record in the dataset.
  // Whilst not necessary, this allows better
  // debugging where there are data issues.
  // This is not used by the algorithm directly.
  optional string uid = 3;

  // Textual metadata describing the record.
  // This may include JSON-serialized information
  // about the source of the record.
  // This is not used by the algorithm directly.
  optional string metadata = 4;

  // An optional serialized JSON object that allows per-record
  // hyper-parameters/configuration/other information to be set.
  // The meaning/interpretation of this field is defined by
}
After creating the protocol buffer, store it in an Amazon S3 location that Amazon SageMaker can access and that can be passed as part of InputDataConfig in create_training_job.

Note
For all Amazon SageMaker algorithms, the ChannelName in InputDataConfig must be set to train. Some algorithms also support a validation or test input channels. These are typically used to evaluate the model's performance by using a hold-out dataset. Hold-out datasets are not used in the initial training but can be used to further tune the model.

Trained Model Deserialization

Amazon SageMaker models are stored as model.tar.gz in the S3 bucket specified in OutputDataConfig S3OutputPath parameter of the create_training_job call. You can specify most of these model artifacts when creating a hosting model. You can also open and review them in your notebook instance. When model.tar.gz is untarred, it contains model_algo-1, which is a serialized Apache MXNet object. For example, you use the following to load the k-means model into memory and view it:

```python
import mxnet as mx
print(mx.ndarray.load('model_algo-1'))
```

Common Data Formats for Inference

Amazon SageMaker algorithms accept and produce several different MIME types for the HTTP payloads used in retrieving online and mini-batch predictions. You can use various AWS services to transform or preprocess records prior to running inference. At a minimum, you need to convert the data for the following:

- Inference request serialization (handled by you)
- Inference request deserialization (handled by the algorithm)
- Inference response serialization (handled by the algorithm)
- Inference response deserialization (handled by you)

Topics

- Convert Data for Inference Request Serialization (p. 650)
- Convert Data for Inference Response Deserialization (p. 651)
- Common Request Formats for All Algorithms (p. 653)
- Use Batch Transform with Built-in Algorithms (p. 654)

Convert Data for Inference Request Serialization

Content type options for Amazon SageMaker algorithm inference requests include: text/csv, application/json, and application/x-recordio-protobuf. Algorithms that don't support all of these types can support other types. XGBoost, for example, only supports text/csv from this list, but also supports text/libsvm.

For text/csv, the value for the Body argument to invoke_endpoint should be a string with commas separating the values for each feature. For example, a record for a model with four features might look like 1.5,16.0,14,23.0. Any transformations performed on the training data should also be performed on the data before obtaining inference. The order of the features matters and must remain unchanged.
application/json is significantly more flexible and provides multiple possible formats for developers to use in their applications. At a high level, in JavaScript, the payload might look like the following:

```javascript
let request = {
  // Instances might contain multiple rows that predictions are sought for.
  "instances": [
    
    // Request and algorithm specific inference parameters.
    "configuration": {},
    // Data in the specific format required by the algorithm.
    "data": {
      "<field name>": dataElement
    }
  ]
}
```

You have the following options for specifying the `dataElement`:

**Protocol buffers equivalent**

```javascript
// Has the same format as the protocol buffers implementation described for training.
let dataElement = {
  "keys": [],
  "values": [],
  "shape": []
}
```

**Simple numeric vector**

```javascript
// An array containing numeric values is treated as an instance containing a
// single dense vector.
let dataElement = [1.5, 16.0, 14.0, 23.0]

// It will be converted to the following representation by the SDK.
let converted = {
  "features": dataElement
}
```

**For multiple records**

```javascript
let request = {
  "instances": [
    // First instance.
    {
      "features": [1.5, 16.0, 14.0, 23.0]
    },
    // Second instance.
    {
      "features": [-2.0, 100.2, 15.2, 9.2]
    }
  ]
}
```

Convert Data for Inference Response Deserialization

Amazon SageMaker algorithms return JSON in several layouts. At a high level, the structure is:

```javascript
let response = {
```
The fields that are included in predictions differ across algorithms. The following are examples of output for the k-means algorithm.

### Single-record inference

```javascript
let response = {
  "predictions": [{
    "closest_cluster": 5,
    "distance_to_cluster": 36.5
  }]
}
```

### Multi-record inference

```javascript
let response = {
  "predictions": [
    // First instance prediction.
    {
      "closest_cluster": 5,
      "distance_to_cluster": 36.5
    },
    // Second instance prediction.
    {
      "closest_cluster": 2,
      "distance_to_cluster": 90.3
    }
  ]
}
```

### Multi-record inference with protobuf input

```json
{
  "features": [],
  "label": {
    "closest_cluster": {
      "values": [ 5.0 ] // e.g. the closest centroid/cluster was 1.0
    },
    "distance_to_cluster": {
      "values": [ 36.5 ]
    }
  },
  "uid": "abc123",
  "metadata": "{"created_at": '2017-06-03' }"
}
```

SageMaker algorithms also support the JSONLINES format, where the per-record response content is same as that in JSON format. The multi-record structure is a concatenation of per-record response objects separated by newline characters. The response content for the built-in KMeans algorithm for 2 input data points is:

```
{"distance_to_cluster": 23.40593910217285, "closest_cluster": 0.0}
{"distance_to_cluster": 27.250282287597656, "closest_cluster": 0.0}
```

While running batch transform, we recommended using the jsonlines response type by setting the Accept field in the CreateTransformJobRequest to application/jsonlines.
Common Request Formats for All Algorithms

Most algorithms use several of the following inference request formats.

**JSON Request Format**

**Content type:** application/JSON

**Dense format**

```json
let request = {
    "instances": [
        {
            "features": [1.5, 16.0, 14.0, 23.0]
        }
    ]
}

let request = {
    "instances": [
        {
            "data": {
                "features": {
                    "values": [1.5, 16.0, 14.0, 23.0]
                }
            }
        }
    ]
}
```

**Sparse format**

```json
{
    "instances": [
        {
            "data": {
                "features": {
                    "keys": [26, 182, 232, 243, 431],
                    "shape": [2000],
                    "values": [1, 1, 1, 4, 1]
                }
            },
            {
                "data": {
                    "features": {
                        "keys": [0, 182, 232, 243, 431],
                        "shape": [2000],
                        "values": [13, 1, 1, 4, 1]
                    }
                },
            }
        ]
    }
}
```

**JSONLINES Request Format**

**Content type:** application/JSONLINES

**Dense format**

A single record in dense format can be represented as either:

```json
{ "features": [1.5, 16.0, 14.0, 23.0] }
```
or:

```json
{ "data": { "features": { "values": [ 1.5, 16.0, 14.0, 23.0 ] } } }
```

### Sparse Format

A single record in sparse format is represented as:

```json
{ "data": { "features": { "keys": [26, 182, 232, 243, 431], "shape": [2000], "values": [1, 1, 1, 4, 1] } } }
```

Multiple records are represented as a concatenation of the above single-record representations, separated by newline characters:

```json
{ "data": { "features": { "keys": [0, 1, 3], "shape": [4], "values": [1, 4, 1] } } }
{ "data": { "features": { "values": [ 1.5, 16.0, 14.0, 23.0 ] } } }
{ "features": [1.5, 16.0, 14.0, 23.0] }
```

### CSV Request Format

**Content type:** text/CSV; label_size=0

**Note**

CSV support is not available for factorization machines.

### RECORDIO Request Format

**Content type:** application/x-recordio-protobuf

### Use Batch Transform with Built-in Algorithms

While running batch transform, we recommended using the JSONLINES response type instead of JSON, if supported by the algorithm. This is accomplished by setting the `Accept` field in the `CreateTransformJobRequest` to `application/jsonlines`.

When you create a transform job, the `SplitType` must be set according to the `ContentType` of the input data. Similarly, depending on the `Accept` field in the `CreateTransformJobRequest`, `AssembleWith` must be set accordingly. Please use the following table to help appropriately set these fields:

<table>
<thead>
<tr>
<th><strong>ContentType</strong></th>
<th><strong>Recommended SplitType</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>application/x-recordio-protobuf</td>
<td>RecordIO</td>
</tr>
<tr>
<td>text/csv</td>
<td>Line</td>
</tr>
<tr>
<td>application/jsonlines</td>
<td>Line</td>
</tr>
<tr>
<td>application/json</td>
<td>None</td>
</tr>
<tr>
<td>application/x-image</td>
<td>None</td>
</tr>
<tr>
<td>image/*</td>
<td>None</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Accept</strong></th>
<th><strong>Recommended AssembleWith</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>application/x-recordio-protobuf</td>
<td>None</td>
</tr>
<tr>
<td>application/json</td>
<td>None</td>
</tr>
</tbody>
</table>
Use built-in algorithms

Accept | Recommended AssembleWith
---|---
application/jsonlines | Line

For more information on response formats for specific algorithms, see the following:

- DeepAR Inference Formats (p. 678)
- Factorization Machine Response Formats (p. 689)
- IP Insights Inference Data Formats (p. 709)
- K-Means Response Formats (p. 718)
- k-NN Request and Response Formats (p. 725)
- Linear learner response formats (p. 746)
- NTM Response Formats (p. 753)
- Data Formats for Object2Vec Inference (p. 767)
- Encoder Embeddings for Object2Vec (p. 769)
- PCA Response Formats (p. 784)
- RCF Response Formats (p. 790)

**Instance Types for Built-in Algorithms**

For training and hosting Amazon SageMaker algorithms, we recommend using the following Amazon EC2 instance types:

- ml.m5.xlarge, ml.m5.4xlarge, and ml.m5.10xlarge
- ml.c5.xlarge, ml.c5.2xlarge, and ml.c5.8xlarge
- ml.p3.xlarge, ml.p3.8xlarge, and ml.p3.16xlarge

Most Amazon SageMaker algorithms have been engineered to take advantage of GPU computing for training. Despite higher per-instance costs, GPUs train more quickly, making them more cost effective. Exceptions are noted in this guide.

The size and type of data can have a great effect on which hardware configuration is most effective. When the same model is trained on a recurring basis, initial testing across a spectrum of instance types can discover configurations that are more cost-effective in the long run. Additionally, algorithms that train most efficiently on GPUs might not require GPUs for efficient inference. Experiment to determine the most cost effectiveness solution.

For more information on SageMaker hardware specifications, see Amazon SageMaker ML Instance Types.

**Logs for Built-in Algorithms**

Amazon SageMaker algorithms produce Amazon CloudWatch logs, which provide detailed information on the training process. To see the logs, in the AWS management console, choose CloudWatch, choose Logs, and then choose the /aws/sagemaker/TrainingJobs log group. Each training job has one log stream per node on which it was trained. The log stream's name begins with the value specified in the TrainingJobName parameter when the job was created.

**Note**

If a job fails and logs do not appear in CloudWatch, it's likely that an error occurred before the start of training. Reasons include specifying the wrong training image or S3 location.

The contents of logs vary by algorithms. However, you can typically find the following information:
Use built-in algorithms

- Confirmation of arguments provided at the beginning of the log
- Errors that occurred during training
- Measurement of an algorithm's accuracy or numerical performance
- Timings for the algorithm and any major stages within the algorithm

Common Errors

If a training job fails, some details about the failure are provided by the `FailureReason` return value in the training job description, as follows:

```python
sage = boto3.client('sagemaker')
sage.describe_training_job(TrainingJobName=job_name)['FailureReason']
```

Others are reported only in the CloudWatch logs. Common errors include the following:

1. Misspecifying a hyperparameter or specifying a hyperparameter that is invalid for the algorithm.

   **From the CloudWatch Log**

   ```
   [10/16/2017 23:45:17 ERROR 139623806805824 train.py:48]
   Additional properties are not allowed (u'mini_batch_siz' was unexpected)
   ```

2. Specifying an invalid value for a hyperparameter.

   **FailureReason**

   ```
   AlgorithmError: u'abc' is not valid under any of the given schemas\nFailed validating u'oneOf' in schema[u'properties'][u'feature_dim']:\n   {u'oneOf':
   [{u'pattern': u'^([1-9][0-9]*)$', u'type': u'string'},
    {u'minimum': 1, u'type': u'integer'}]}
   ```

3. Inaccurate protobuf file format.

   **From the CloudWatch log**

   ```
   [10/17/2017 18:01:04 ERROR 140234860816192 train.py:48] cannot copy sequence with size 785 to array axis with dimension 784
   ```

BlazingText algorithm

The Amazon SageMaker BlazingText algorithm provides highly optimized implementations of the Word2vec and text classification algorithms. The Word2vec algorithm is useful for many downstream natural language processing (NLP) tasks, such as sentiment analysis, named entity recognition, machine translation, etc. Text classification is an important task for applications that perform web searches, information retrieval, ranking, and document classification.

The Word2vec algorithm maps words to high-quality distributed vectors. The resulting vector representation of a word is called a word embedding. Words that are semantically similar correspond to
vectors that are close together. That way, word embeddings capture the semantic relationships between words.

Many natural language processing (NLP) applications learn word embeddings by training on large collections of documents. These pretrained vector representations provide information about semantics and word distributions that typically improves the generalizability of other models that are later trained on a more limited amount of data. Most implementations of the Word2vec algorithm are not optimized for multi-core CPU architectures. This makes it difficult to scale to large datasets.

With the BlazingText algorithm, you can scale to large datasets easily. Similar to Word2vec, it provides the Skip-gram and continuous bag-of-words (CBOW) training architectures. BlazingText's implementation of the supervised multi-class, multi-label text classification algorithm extends the fastText text classifier to use GPU acceleration with custom CUDA kernels. You can train a model on more than a billion words in a couple of minutes using a multi-core CPU or a GPU. And, you achieve performance on par with the state-of-the-art deep learning text classification algorithms.

The BlazingText algorithm is not parallelizable. For more information on parameters related to training, see Docker Registry Paths for SageMaker Built-in Algorithms.

The SageMaker BlazingText algorithms provides the following features:

- Accelerated training of the fastText text classifier on multi-core CPUs or a GPU and Word2Vec on GPUs using highly optimized CUDA kernels. For more information, see BlazingText: Scaling and Accelerating Word2Vec using Multiple GPUs.
- **Enriched Word Vectors with Subword Information** by learning vector representations for character n-grams. This approach enables BlazingText to generate meaningful vectors for out-of-vocabulary (OOV) words by representing their vectors as the sum of the character n-gram (subword) vectors.
- A `batch_skipgram` mode for the Word2Vec algorithm that allows faster training and distributed computation across multiple CPU nodes. The `batch_skipgram` mode does mini-batching using the Negative Sample Sharing strategy to convert level-1 BLAS operations into level-3 BLAS operations. This efficiently leverages the multiply-add instructions of modern architectures. For more information, see Parallelizing Word2Vec in Shared and Distributed Memory.

To summarize, the following modes are supported by BlazingText on different types instances:

<table>
<thead>
<tr>
<th>Modes</th>
<th>Word2Vec</th>
<th>Text Classification</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(Unsupervised Learning)</td>
<td>(Supervised Learning)</td>
</tr>
<tr>
<td>Single CPU instance</td>
<td>cbow</td>
<td>supervised</td>
</tr>
<tr>
<td></td>
<td>Skip-gram</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Batch Skip-gram</td>
<td></td>
</tr>
<tr>
<td>Single GPU instance (with 1 or more GPUs)</td>
<td>cbow</td>
<td>supervised with one GPU</td>
</tr>
<tr>
<td></td>
<td>Skip-gram</td>
<td></td>
</tr>
<tr>
<td>Multiple CPU instances</td>
<td>Batch Skip-gram</td>
<td>None</td>
</tr>
</tbody>
</table>

For more information about the mathematics behind BlazingText, see BlazingText: Scaling and Accelerating Word2Vec using Multiple GPUs.

**Topics**

- Input/Output Interface for the BlazingText Algorithm (p. 658)
- EC2 Instance Recommendation for the BlazingText Algorithm (p. 660)
Input/Output Interface for the BlazingText Algorithm

The BlazingText algorithm expects a single preprocessed text file with space-separated tokens. Each line in the file should contain a single sentence. If you need to train on multiple text files, concatenate them into one file and upload the file in the respective channel.

Training and Validation Data Format

Training and Validation Data Format for the Word2Vec Algorithm

For Word2Vec training, upload the file under the `train` channel. No other channels are supported. The file should contain a training sentence per line.

Training and Validation Data Format for the Text Classification Algorithm

For supervised mode, you can train with file mode or with the augmented manifest text format.

Train with File Mode

For supervised mode, the training/validation file should contain a training sentence per line along with the labels. Labels are words that are prefixed by the string `__label__`. Here is an example of a training/validation file:

```plaintext
__label__4  linux ready for prime time , intel says , despite all the linux hype , the open-source movement has yet to make a huge splash in the desktop market . that may be about to change , thanks to chipmaking giant intel corp .
__label__2  bowled by the slower one again , kolkata , november 14 the past caught up with sourav ganguly as the indian skippers return to international cricket was short lived .
```

Note

The order of labels within the sentence doesn't matter.

Upload the training file under the train channel, and optionally upload the validation file under the validation channel.

Train with Augmented Manifest Text Format

The supervised mode also supports the augmented manifest format, which enables you to do training in pipe mode without needing to create RecordIO files. While using the format, an S3 manifest file needs to be generated that contains the list of sentences and their corresponding labels. The manifest file format should be in JSON Lines format in which each line represents one sample. The sentences are specified using the `source` tag and the label can be specified using the `label` tag. Both `source` and `label` tags should be provided under the `AttributeNames` parameter value as specified in the request.

```json
{"source":"linux ready for prime time , intel says , despite all the linux hype", "label":1}
{"source":"bowled by the slower one again , kolkata , november 14 the past caught up with sourav ganguly", "label":2}
```

Multi-label training is also supported by specifying a JSON array of labels.

```json
{"source":"linux ready for prime time , intel says , despite all the linux hype", "label" : [1, 3]}
```
For more information on augmented manifest files, see Provide Dataset Metadata to Training Jobs with an Augmented Manifest File (p. 1124).

Model Artifacts and Inference

Model Artifacts for the Word2Vec Algorithm

For Word2Vec training, the model artifacts consist of vectors.txt, which contains words-to-vectors mapping, and vectors.bin, a binary used by BlazingText for hosting, inference, or both. vectors.txt stores the vectors in a format that is compatible with other tools like Gensim and Spacy. For example, a Gensim user can run the following commands to load the vectors.txt file:

```python
from gensim.models import KeyedVectors
word_vectors = KeyedVectors.load_word2vec_format('vectors.txt', binary=False)
word_vectors.most_similar(positive=['woman', 'king'], negative=['man'])
word_vectors.doesn't_match("breakfast cereal dinner lunch".split())
```

If the evaluation parameter is set to True, an additional file, eval.json, is created. This file contains the similarity evaluation results (using Spearman's rank correlation coefficients) on WS-353 dataset. The number of words from the WS-353 dataset that aren't there in the training corpus are reported.

For inference requests, the model accepts a JSON file containing a list of strings and returns a list of vectors. If the word is not found in vocabulary, inference returns a vector of zeros. If subwords is set to True during training, the model is able to generate vectors for out-of-vocabulary (OOV) words.

Sample JSON Request

Mime-type: application/json

```json
{
  "instances": ["word1", "word2", "word3"]
}
```

Model Artifacts for the Text Classification Algorithm

Training with supervised outputs creates a model.bin file that can be consumed by BlazingText hosting. For inference, the BlazingText model accepts a JSON file containing a list of sentences and returns a list of corresponding predicted labels and probability scores. Each sentence is expected to be a string with space-separated tokens, words, or both.

Sample JSON Request

Mime-type: application/json

```json
{
  "instances": ["the movie was excellent", "i did not like the plot ."]
}
```

By default, the server returns only one prediction, the one with the highest probability. For retrieving the top k predictions, you can set k in the configuration, as follows:

```json
{
  "instances": ["the movie was excellent", "i did not like the plot ."],
  "configuration": {"k": 2}
}
```
For BlazingText, the `content-type` and `accept` parameters must be equal. For batch transform, they both need to be `application/jsonlines`. If they differ, the `Accept` field is ignored. The format for input follows:

```
content-type: application/jsonlines

{"source": "source_0"}
{"source": "source_1"}
```

If you need to pass the value of k for top-k, then you can do it in the following way:

```
{"source": "source_0", "k": 2}
{"source": "source_1", "k": 3}
```

The format for output follows:

```
accept: application/jsonlines

{"prob": [prob_1], "label": ["__label__1"]}
{"prob": [prob_1], "label": ["__label__1"]}
```

If you have passed the value of k to be more than 1, then response will be in this format:

```
{"prob": [prob_1, prob_2], "label": ["__label__1", "__label__2"]}
{"prob": [prob_1, prob_2], "label": ["__label__1", "__label__2"]}
```

For both supervised (text classification) and unsupervised (Word2Vec) modes, the binaries (*.bin) produced by BlazingText can be cross-consumed by fastText and vice versa. You can use binaries produced by BlazingText by fastText. Likewise, you can host the model binaries created with fastText using BlazingText.

Here is an example of how to use a model generated with BlazingText with fastText:

```
#Download the model artifact from S3
aws s3 cp s3://<YOUR_S3_BUCKET>/<PREFIX>/model.tar.gz model.tar.gz

#Unzip the model archive
tar -xzf model.tar.gz

#Use the model archive with fastText
fasttext predict ./model.bin test.txt
```

However, the binaries are only supported when training on CPU and single GPU; training on multi-GPU will not produce binaries.

For more details on dataset formats and model hosting, see the example notebooks Text Classification with the BlazingText Algorithm, FastText Models, and Generating Subword Embeddings with the Word2Vec Algorithm.

**EC2 Instance Recommendation for the BlazingText Algorithm**

For `cbow` and `skipgram` modes, BlazingText supports single CPU and single GPU instances. Both of these modes support learning of subwords embeddings. To achieve the highest speed without compromising accuracy, we recommend that you use an `ml.p3.2xlarge` instance.

For `batch_skipgram` mode, BlazingText supports single or multiple CPU instances. When training on multiple instances, set the value of the `S3DataDistributionType` field of the `S3DataSource` object that you pass to `CreateTrainingJob` to `FullyReplicated`. BlazingText takes care of distributing data across machines.
For the supervised text classification mode, a C5 instance is recommended if the training dataset is less than 2 GB. For larger datasets, use an instance with a single GPU (ml.p2.xlarge or ml.p3.2xlarge).

**BlazingText Sample Notebooks**

For a sample notebook that uses the SageMaker BlazingText algorithm to train and deploy supervised binary and multiclass classification models, see [Blazing Text classification on the DBPedia dataset](#). For instructions for creating and accessing Jupyter notebook instances that you can use to run the example in SageMaker, see [Use Amazon SageMaker Notebook Instances](p. 124). After creating and opening a notebook instance, choose the **SageMaker Examples** tab to see a list of all the SageMaker examples. The topic modeling example notebooks that use the Blazing Text are located in the **Introduction to Amazon algorithms** section. To open a notebook, choose its **Use** tab, then choose **Create copy**.

**BlazingText Hyperparameters**

When you start a training job with a CreateTrainingJob request, you specify a training algorithm. You can also specify algorithm-specific hyperparameters as string-to-string maps. The hyperparameters for the BlazingText algorithm depend on which mode you use: Word2Vec (unsupervised) and Text Classification (supervised).

**Word2Vec Hyperparameters**

The following table lists the hyperparameters for the BlazingText Word2Vec training algorithm provided by Amazon SageMaker.

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
</table>
| mode           | The Word2vec architecture used for training.  
 Required  
 Valid values: batch_skipgram, skipgram, or cbow |
| batch_size     | The size of each batch when mode is set to batch_skipgram. Set to a number between 10 and 20.  
 Optional  
 Valid values: Positive integer  
 Default value: 11 |
| buckets        | The number of hash buckets to use for subwords.  
 Optional  
 Valid values: positive integer  
 Default value: 2000000 |
| epochs         | The number of complete passes through the training data.  
 Optional  
 Valid values: Positive integer  
 Default value: 5 |
<p>| evaluation     | Whether the trained model is evaluated using the WordSimilarity-353 Test. |</p>
<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optional</td>
<td>Valid values: (Boolean) True or False</td>
</tr>
<tr>
<td>Default value: True</td>
<td></td>
</tr>
<tr>
<td>learning_rate</td>
<td>The step size used for parameter updates.</td>
</tr>
<tr>
<td>Optional</td>
<td>Valid values: Positive float</td>
</tr>
<tr>
<td>Default value: 0.05</td>
<td></td>
</tr>
<tr>
<td>min_char</td>
<td>The minimum number of characters to use for subwords/character n-grams.</td>
</tr>
<tr>
<td>Optional</td>
<td>Valid values: positive integer</td>
</tr>
<tr>
<td>Default value: 3</td>
<td></td>
</tr>
<tr>
<td>min_count</td>
<td>Words that appear less than min_count times are discarded.</td>
</tr>
<tr>
<td>Optional</td>
<td>Valid values: Non-negative integer</td>
</tr>
<tr>
<td>Default value: 5</td>
<td></td>
</tr>
<tr>
<td>max_char</td>
<td>The maximum number of characters to use for subwords/character n-grams</td>
</tr>
<tr>
<td>Optional</td>
<td>Valid values: positive integer</td>
</tr>
<tr>
<td>Default value: 6</td>
<td></td>
</tr>
<tr>
<td>negative_samples</td>
<td>The number of negative samples for the negative sample sharing strategy.</td>
</tr>
<tr>
<td>Optional</td>
<td>Valid values: Positive integer</td>
</tr>
<tr>
<td>Default value: 5</td>
<td></td>
</tr>
<tr>
<td>sampling_threshold</td>
<td>The threshold for the occurrence of words. Words that appear with higher</td>
</tr>
<tr>
<td></td>
<td>frequency in the training data are randomly down-sampled.</td>
</tr>
<tr>
<td>Optional</td>
<td>Valid values: Positive fraction. The recommended range is (0, 1e-3]</td>
</tr>
<tr>
<td>Default value: 0.0001</td>
<td></td>
</tr>
</tbody>
</table>
### Use built-in algorithms

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
</table>
| subwords       | Whether to learn subword embeddings on not.  
  **Optional**  
  Valid values: (Boolean) True or False  
  Default value: False |
| vector_dim     | The dimension of the word vectors that the algorithm learns.  
  **Optional**  
  Valid values: Positive integer  
  Default value: 100 |
| window_size    | The size of the context window. The context window is the number of words surrounding the target word used for training.  
  **Optional**  
  Valid values: Positive integer  
  Default value: 5 |

### Text Classification Hyperparameters

The following table lists the hyperparameters for the Text Classification training algorithm provided by Amazon SageMaker.

**Note**  
Although some of the parameters are common between the Text Classification and Word2Vec modes, they might have different meanings depending on the context.

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
</table>
| mode           | The training mode.  
  **Required**  
  Valid values: supervised |
| buckets        | The number of hash buckets to use for word n-grams.  
  **Optional**  
  Valid values: Positive integer  
  Default value: 2000000 |
| early_stopping | Whether to stop training if validation accuracy doesn't improve after a patience number of epochs.  
  **Optional**  
  Valid values: (Boolean) True or False  
  Default value: False |
<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>epochs</td>
<td>The maximum number of complete passes through the training data. Optional Valid values: Positive integer Default value: 5</td>
</tr>
<tr>
<td>learning_rate</td>
<td>The step size used for parameter updates. Optional Valid values: Positive float Default value: 0.05</td>
</tr>
<tr>
<td>min_count</td>
<td>Words that appear less than min_count times are discarded. Optional Valid values: Non-negative integer Default value: 5</td>
</tr>
<tr>
<td>min_epochs</td>
<td>The minimum number of epochs to train before early stopping logic is invoked. Optional Valid values: Positive integer Default value: 5</td>
</tr>
<tr>
<td>patience</td>
<td>The number of epochs to wait before applying early stopping when no progress is made on the validation set. Used only when early_stopping is True. Optional Valid values: Positive integer Default: 4</td>
</tr>
<tr>
<td>vector_dim</td>
<td>The dimension of the embedding layer. Optional Valid values: Positive integer Default value: 100</td>
</tr>
<tr>
<td>word_ngrams</td>
<td>The number of word n-gram features to use. Optional Valid values: Positive integer Default value: 2</td>
</tr>
</tbody>
</table>
Tune a BlazingText Model

Automatic model tuning, also known as hyperparameter tuning, finds the best version of a model by running many jobs that test a range of hyperparameters on your dataset. You choose the tunable hyperparameters, a range of values for each, and an objective metric. You choose the objective metric from the metrics that the algorithm computes. Automatic model tuning searches the hyperparameters chosen to find the combination of values that result in the model that optimizes the objective metric.

For more information about model tuning, see Perform Automatic Model Tuning (p. 1023).

Metrics Computed by the BlazingText Algorithm

The BlazingText Word2Vec algorithm (skipgram, cbow, and batch_skipgram modes) reports on a single metric during training: train:mean_rho. This metric is computed on WS-353 word similarity datasets. When tuning the hyperparameter values for the Word2Vec algorithm, use this metric as the objective.

The BlazingText Text Classification algorithm (supervised mode), also reports on a single metric during training: the validation:accuracy. When tuning the hyperparameter values for the text classification algorithm, use these metrics as the objective.

<table>
<thead>
<tr>
<th>Metric Name</th>
<th>Description</th>
<th>Optimization Direction</th>
</tr>
</thead>
<tbody>
<tr>
<td>train:mean_rho</td>
<td>The mean rho (Spearman's rank correlation coefficient) on WS-353 word similarity datasets</td>
<td>Maximize</td>
</tr>
<tr>
<td>validation:accuracy</td>
<td>The classification accuracy on the user-specified validation dataset</td>
<td>Maximize</td>
</tr>
</tbody>
</table>

Tunable BlazingText Hyperparameters

Tunable Hyperparameters for the Word2Vec Algorithm

Tune an Amazon SageMaker BlazingText Word2Vec model with the following hyperparameters. The hyperparameters that have the greatest impact on Word2Vec objective metrics are: mode, learning_rate, window_size, vector_dim, and negative_samples.

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Parameter Type</th>
<th>Recommended Ranges or Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>batch_size</td>
<td>IntegerParameterRange</td>
<td>[8-32]</td>
</tr>
<tr>
<td>epochs</td>
<td>IntegerParameterRange</td>
<td>[5-15]</td>
</tr>
<tr>
<td>learning_rate</td>
<td>ContinuousParameterRange</td>
<td>MinValue: 0.005, MaxValue: 0.01</td>
</tr>
<tr>
<td>min_count</td>
<td>IntegerParameterRange</td>
<td>[0-100]</td>
</tr>
<tr>
<td>mode</td>
<td>CategoricalParameterRange</td>
<td>['batch_skipgram', 'skipgram', 'cbow']</td>
</tr>
<tr>
<td>negative_samples</td>
<td>IntegerParameterRange</td>
<td>[5-25]</td>
</tr>
<tr>
<td>sampling_threshold</td>
<td>ContinuousParameterRange</td>
<td>MinValue: 0.0001, MaxValue: 0.001</td>
</tr>
<tr>
<td>vector_dim</td>
<td>IntegerParameterRange</td>
<td>[32-300]</td>
</tr>
</tbody>
</table>
DeepAR Forecasting Algorithm

The Amazon SageMaker DeepAR forecasting algorithm is a supervised learning algorithm for forecasting scalar (one-dimensional) time series using recurrent neural networks (RNN). Classical forecasting methods, such as autoregressive integrated moving average (ARIMA) or exponential smoothing (ETS), fit a single model to each individual time series. They then use that model to extrapolate the time series into the future.

In many applications, however, you have many similar time series across a set of cross-sectional units. For example, you might have time series groupings for demand for different products, server loads, and requests for webpages. For this type of application, you can benefit from training a single model jointly over all of the time series. DeepAR takes this approach. When your dataset contains hundreds of related time series, DeepAR outperforms the standard ARIMA and ETS methods. You can also use the trained model to generate forecasts for new time series that are similar to the ones it has been trained on.

The training input for the DeepAR algorithm is one or, preferably, more target time series that have been generated by the same process or similar processes. Based on this input dataset, the algorithm trains a model that learns an approximation of this process/processes and uses it to predict how the target time series evolves. Each target time series can be optionally associated with a vector of static (time-independent) categorical features provided by the cat field and a vector of dynamic (time-dependent) time series provided by the dynamic_feat field. SageMaker trains the DeepAR model by randomly sampling training examples from each target time series in the training dataset. Each training example consists of a pair of adjacent context and prediction windows with fixed predefined lengths. To control how far in the past the network can see, use the context_length hyperparameter. To control how far in the future predictions can be made, use the prediction_length hyperparameter. For more information, see How the DeepAR Algorithm Works (p. 670).

Topics
- Input/Output Interface for the DeepAR Algorithm (p. 667)
Input/Output Interface for the DeepAR Algorithm

DeepAR supports two data channels. The required train channel describes the training dataset. The optional test channel describes a dataset that the algorithm uses to evaluate model accuracy after training. You can provide training and test datasets in JSON Lines format. Files can be in gzip or Parquet file format.

When specifying the paths for the training and test data, you can specify a single file or a directory that contains multiple files, which can be stored in subdirectories. If you specify a directory, DeepAR uses all files in the directory as inputs for the corresponding channel, except those that start with a period (.) and those named _SUCCESS. This ensures that you can directly use output folders produced by Spark jobs as input channels for your DeepAR training jobs.

By default, the DeepAR model determines the input format from the file extension (.json, .json.gz, or .parquet) in the specified input path. If the path does not end in one of these extensions, you must explicitly specify the format in the SDK for Python. Use the content_type parameter of the s3_input class.

The records in your input files should contain the following fields:

- start—A string with the format YYYY-MM-DD HH:MM:SS. The start timestamp can't contain time zone information.
- target—An array of floating-point values or integers that represent the time series. You can encode missing values as null literals, or as "NaN" strings in JSON, or as nan floating-point values in Parquet.
- dynamic_feat (optional)—An array of arrays of floating-point values or integers that represents the vector of custom feature time series (dynamic features). If you set this field, all records must have the same number of inner arrays (the same number of feature time series). In addition, each inner array must have the same length as the associated target value. Missing values are not supported in the features. For example, if target time series represents the demand of different products, an associated dynamic_feat might be a boolean time-series which indicates whether a promotion was applied (1) to the particular product or not (0):

  ```json
  {"start": ..., "target": [1, 5, 10, 2], "dynamic_feat": [[0, 1, 1, 0]]}
  ```

- cat (optional)—An array of categorical features that can be used to encode the groups that the record belongs to. Categorical features must be encoded as a 0-based sequence of positive integers. For example, the categorical domain {R, G, B} can be encoded as {0, 1, 2}. All values from each categorical domain must be represented in the training dataset. That's because the DeepAR algorithm can forecast only for categories that have been observed during training. And, each categorical feature is embedded in a low-dimensional space whose dimensionality is controlled by the embedding_dimension hyperparameter. For more information, see DeepAR Hyperparameters (p. 673).

If you use a JSON file, it must be in JSON Lines format. For example:

```json
{"start": "2009-11-01 00:00:00", "target": [4.3, "NaN", 5.1, ...], "cat": [0, 1], "dynamic_feat": [[1.1, 1.2, 0.5, ...]]}
```
In this example, each time series has two associated categorical features and one time series feature.

For Parquet, you use the same three fields as columns. In addition, "start" can be the datetime type. You can compress Parquet files using gzip (gzip) or the Snappy compression library (snappy).

If the algorithm is trained without cat and dynamic_feat fields, it learns a "global" model, that is a model that is agnostic to the specific identity of the target time series at inference time and is conditioned only on its shape.

If the model is conditioned on the cat and dynamic_feat feature data provided for each time series, the prediction will probably be influenced by the character of time series with the corresponding cat features. For example, if the target time series represents the demand of clothing items, you can associate a two-dimensional cat vector that encodes the type of item (e.g. 0 = shoes, 1 = dress) in the first component and the color of an item (e.g. 0 = red, 1 = blue) in the second component. A sample input would look as follows:

```json
{ "start": ..., "target": ..., "cat": [0, 0], ... } # red shoes
{ "start": ..., "target": ..., "cat": [1, 1], ... } # blue dress
```

At inference time, you can request predictions for targets with cat values that are combinations of the cat values observed in the training data, for example:

```json
{ "start": ..., "target": ..., "cat": [0, 1], ... } # blue shoes
{ "start": ..., "target": ..., "cat": [1, 0], ... } # red dress
```

The following guidelines apply to training data:

- The start time and length of the time series can differ. For example, in marketing, products often enter a retail catalog at different dates, so their start dates naturally differ. But all series must have the same frequency, number of categorical features, and number of dynamic features.
- Shuffle the training file with respect to the position of the time series in the file. In other words, the time series should occur in random order in the file.
- Make sure to set the start field correctly. The algorithm uses the start timestamp to derive the internal features.
- If you use categorical features (cat), all time series must have the same number of categorical features. If the dataset contains the cat field, the algorithm uses it and extracts the cardinality of the groups from the dataset. By default, cardinality is "auto". If the dataset contains the cat field, but you don't want to use it, you can disable it by setting cardinality to "". If a model was trained using a cat feature, you must include it for inference.
- If your dataset contains the dynamic_feat field, the algorithm uses it automatically. All time series have to have the same number of feature time series. The time points in each of the feature time series correspond one-to-one to the time points in the target. In addition, the entry in the dynamic_feat field should have the same length as the target. If the dataset contains the dynamic_feat field, but you don't want to use it, disable it by setting(num_dynamic_feat to ""). If the model was trained with the dynamic_feat field, you must provide this field for inference. In addition, each of the features has to have the length of the provided target plus the prediction_length. In other words, you must provide the feature value in the future.

If you specify optional test channel data, the DeepAR algorithm evaluates the trained model with different accuracy metrics. The algorithm calculates the root mean square error (RMSE) over the test data as follows:
Use built-in algorithms

\[ \text{RMSE} = \sqrt{\frac{1}{nT} \sum_{i,t} (\hat{y}_{i,t} - y_{i,t})^2} \]

\( y_{i,t} \) is the true value of time series \( i \) at the time \( t \). \( \hat{y}_{i,t} \) is the mean prediction. The sum is over all \( n \) time series in the test set and over the last \( T \) time points for each time series, where \( T \) corresponds to the forecast horizon. You specify the length of the forecast horizon by setting the prediction_length hyperparameter. For more information, see DeepAR Hyperparameters (p. 673).

In addition, the algorithm evaluates the accuracy of the forecast distribution using weighted quantile loss. For a quantile in the range \([0, 1]\), the weighted quantile loss is defined as follows:

\[ w\text{QuantileLoss}[\tau] = \frac{\sum_{i,t} Q_{i,t}^{(\tau)}}{\sum_{i,t} |y_{i,t}|} \quad \text{with} \quad Q_{i,t}^{(\tau)} = \begin{cases} (1 - \tau)|q_{i,t}^{(\tau)} - y_{i,t}| & \text{if } q_{i,t}^{(\tau)} > y_{i,t} \\ \tau|q_{i,t}^{(\tau)} - y_{i,t}| & \text{otherwise} \end{cases} \]

\( q_{i,t}^{(\tau)} \) is the \( \tau \)-quantile of the distribution that the model predicts. To specify which quantiles to calculate loss for, set the test_quantiles hyperparameter. In addition to these, the average of the prescribed quantile losses is reported as part of the training logs. For information, see DeepAR Hyperparameters (p. 673).

For inference, DeepAR accepts JSON format and the following fields:

- "instances", which includes one or more time series in JSON Lines format
- A name of "configuration", which includes parameters for generating the forecast

For more information, see DeepAR Inference Formats (p. 678).

**Best Practices for Using the DeepAR Algorithm**

When preparing your time series data, follow these best practices to achieve the best results:

- Except for when splitting your dataset for training and testing, always provide the entire time series for training, testing, and when calling the model for inference. Regardless of how you set context_length, don’t break up the time series or provide only a part of it. The model uses data points further back than the value set in context_length for the lagged values feature.

- When tuning a DeepAR model, you can split the dataset to create a training dataset and a test dataset. In a typical evaluation, you would test the model on the same time series used for training, but on the future prediction_length time points that follow immediately after the last time point visible during training. You can create training and test datasets that satisfy this criteria by using the entire dataset (the full length of all time series that are available) as a test set and removing the last prediction_length points from each time series for training. During training, the model doesn’t see the target values for time points on which it is evaluated during testing. During testing, the algorithm withholds the last prediction_length points of each time series in the test set and generates a prediction. Then it compares the forecast with the withheld values. You can create more complex evaluations by repeating time series multiple times in the test set, but cutting them at different endpoints. With this approach, accuracy metrics are averaged over multiple forecasts from different time points. For more information, see Tune a DeepAR Model (p. 677).

- Avoid using very large values (>400) for the prediction_length because it makes the model slow and less accurate. If you want to forecast further into the future, consider aggregating your data at a higher frequency. For example, use 5min instead of 1min.

- Because lags are used, a model can look further back in the time series than the value specified for context_length. Therefore, you don’t need to set this parameter to a large value. We recommend starting with the value that you used for prediction_length.
• We recommend training a DeepAR model on as many time series as are available. Although a DeepAR model trained on a single time series might work well, standard forecasting algorithms, such as ARIMA or ETS, might provide more accurate results. The DeepAR algorithm starts to outperform the standard methods when your dataset contains hundreds of related time series. Currently, DeepAR requires that the total number of observations available across all training time series is at least 300.

EC2 Instance Recommendations for the DeepAR Algorithm

You can train DeepAR on both GPU and CPU instances and in both single and multi-machine settings. We recommend starting with a single CPU instance (for example, ml.c4.2xlarge or ml.c4.4xlarge), and switching to GPU instances and multiple machines only when necessary. Using GPUs and multiple machines improves throughput only for larger models (with many cells per layer and many layers) and for large mini-batch sizes (for example, greater than 512).

For inference, DeepAR supports only CPU instances.

Specifying large values for context_length, prediction_length, num_cells, num_layers, or mini_batch_size can create models that are too large for small instances. In this case, use a larger instance type or reduce the values for these parameters. This problem also frequently occurs when running hyperparameter tuning jobs. In that case, use an instance type large enough for the model tuning job and consider limiting the upper values of the critical parameters to avoid job failures.

DeepAR Sample Notebooks

For a sample notebook that shows how to prepare a time series dataset for training the SageMaker DeepAR algorithm and how to deploy the trained model for performing inferences, see Time series forecasting with DeepAR - Synthetic data as well as DeepAR demo on electricity dataset, which illustrates the advanced features of DeepAR on a real world dataset. For instructions on creating and accessing Jupyter notebook instances that you can use to run the example in SageMaker, see Use Amazon SageMaker Notebook Instances (p. 124). After creating and opening a notebook instance, choose the SageMaker Examples tab to see a list of all of the SageMaker examples. To open a notebook, choose its Use tab, and choose Create copy.

How the DeepAR Algorithm Works

During training, DeepAR accepts a training dataset and an optional test dataset. It uses the test dataset to evaluate the trained model. In general, the datasets don't have to contain the same set of time series. You can use a model trained on a given training set to generate forecasts for the future of the time series in the training set, and for other time series. Both the training and the test datasets consist of one or, preferably, more target time series. Each target time series can optionally be associated with a vector of feature time series and a vector of categorical features. For more information, see Input/Output Interface for the DeepAR Algorithm (p. 667).

For example, the following is an element of a training set indexed by \(i\) which consists of a target time series, \(Z_{i,t}\) and two associated feature time series, \(X_{i,1,t}\) and \(X_{i,2,t}\):
The target time series might contain missing values, which are represented by line breaks in the time series. DeepAR supports only feature time series that are known in the future. This allows you to run "what if?" scenarios. What happens, for example, if I change the price of a product in some way?

Each target time series can also be associated with a number of categorical features. You can use these features to encode which groupings a time series belongs to. Categorical features allow the model to learn typical behavior for groups, which it can use to increase model accuracy. DeepAR implements this by learning an embedding vector for each group that captures the common properties of all time series in the group.

How Feature Time Series Work in the DeepAR Algorithm

To facilitate learning time-dependent patterns, such as spikes during weekends, DeepAR automatically creates feature time series based on the frequency of the target time series. For example, DeepAR creates two feature time series (day of the month and day of the year) for a weekly time series frequency. It uses these derived feature time series with the custom feature time series that you provide during training and inference. The following figure shows two of these derived time series features: \( u_{i,1,t} \) represents the hour of the day and \( u_{i,2,t} \) the day of the week.

The DeepAR algorithm automatically generates these feature time series. The following table lists the derived features for the supported basic time frequencies.

<table>
<thead>
<tr>
<th>Frequency of the Time Series</th>
<th>Derived Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minute</td>
<td>minute-of-hour, hour-of-day, day-of-week, day-of-month, day-of-year</td>
</tr>
</tbody>
</table>
**Use built-in algorithms**

### Frequency of the Time Series

<table>
<thead>
<tr>
<th>Derived Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hour</td>
</tr>
<tr>
<td>hour-of-day, day-of-week, day-of-month, day-of-year</td>
</tr>
<tr>
<td>Day</td>
</tr>
<tr>
<td>day-of-week, day-of-month, day-of-year</td>
</tr>
<tr>
<td>Week</td>
</tr>
<tr>
<td>day-of-week, day-of-month, day-of-year</td>
</tr>
<tr>
<td>Month</td>
</tr>
<tr>
<td>month-of-year</td>
</tr>
</tbody>
</table>

DeepAR trains a model by randomly sampling several training examples from each of the time series in the training dataset. Each training example consists of a pair of adjacent context and prediction windows with fixed predefined lengths. The `context_length` hyperparameter controls how far in the past the network can see, and the `prediction_length` hyperparameter controls how far in the future predictions can be made. During training, the algorithm ignores training set elements containing time series that are shorter than a specified prediction length. The following figure represents five samples with context lengths of 12 hours and prediction lengths of 6 hours drawn from element $i$. For brevity, we've omitted the feature time series $x_{i,1,t}$ and $u_{i,2,t}$.

To capture seasonality patterns, DeepAR also automatically feeds lagged values from the target time series. In the example with hourly frequency, for each time index, $t = T$, the model exposes the $z_{i,t}$ values, which occurred approximately one, two, and three days in the past.

For inference, the trained model takes as input target time series, which might or might not have been used during training, and forecasts a probability distribution for the next `prediction_length` values.
Because DeepAR is trained on the entire dataset, the forecast takes into account patterns learned from similar time series.

For information on the mathematics behind DeepAR, see DeepAR: Probabilistic Forecasting with Autoregressive Recurrent Networks.

**DeepAR Hyperparameters**

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>context_length</td>
<td>The number of time-points that the model gets to see before making the prediction. The value for this parameter should be about the same as the prediction_length. The model also receives lagged inputs from the target, so context_length can be much smaller than typical seasonalities. For example, a daily time series can have yearly seasonality. The model automatically includes a lag of one year, so the context length can be shorter than a year. The lag values that the model picks depend on the frequency of the time series. For example, lag values for daily frequency are previous week, 2 weeks, 3 weeks, 4 weeks, and year. Required</td>
</tr>
<tr>
<td>epochs</td>
<td>The maximum number of passes over the training data. The optimal value depends on your data size and learning rate. See also early_stopping_patience. Typical values range from 10 to 1000. Required</td>
</tr>
<tr>
<td>prediction_length</td>
<td>The number of time-steps that the model is trained to predict, also called the forecast horizon. The trained model always generates forecasts with this length. It can't generate longer forecasts. The prediction_length is fixed when a model is trained and it cannot be changed later. Required</td>
</tr>
<tr>
<td>time_freq</td>
<td>The granularity of the time series in the dataset. Use time_freq to select appropriate date features and lags. The model supports the following basic frequencies. It also supports multiples of these basic frequencies. For example, 5min specifies a frequency of 5 minutes. Required</td>
</tr>
</tbody>
</table>

- **M**: monthly
- **W**: weekly
- **D**: daily
- **H**: hourly
- **min**: every minute
<table>
<thead>
<tr>
<th><strong>Parameter Name</strong></th>
<th><strong>Description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>cardinality</td>
<td>When using the categorical features (cat), cardinality is an array specifying the number of categories (groups) per categorical feature. Set this to auto to infer the cardinality from the data. The auto mode also works when no categorical features are used in the dataset. This is the recommended setting for the parameter. Set cardinality to ignore to force DeepAR to not use categorical features, even if they are present in the data. To perform additional data validation, it is possible to explicitly set this parameter to the actual value. For example, if two categorical features are provided where the first has 2 and the other has 3 possible values, set this to [2, 3]. For more information on how to use categorical features, see the data-section on the main documentation page of DeepAR. <strong>Optional</strong> Valid values: auto, ignore, array of positive integers, empty string, or <strong>Default value:</strong> auto</td>
</tr>
<tr>
<td>dropout_rate</td>
<td>The dropout rate to use during training. The model uses zoneout regularization. For each iteration, a random subset of hidden neurons are not updated. Typical values are less than 0.2. <strong>Optional</strong> Valid values: float <strong>Default value:</strong> 0.1</td>
</tr>
<tr>
<td>early_stopping_patience</td>
<td>If this parameter is set, training stops when no progress is made within the specified number of epochs. The model that has the lowest loss is returned as the final model. <strong>Optional</strong> Valid values: integer</td>
</tr>
<tr>
<td>Parameter Name</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| embedding_dimension | Size of embedding vector learned per categorical feature (same value is used for all categorical features). The DeepAR model can learn group-level time series patterns when a categorical grouping feature is provided. To do this, the model learns an embedding vector of size embedding_dimension for each group, capturing the common properties of all time series in the group. A larger embedding_dimension allows the model to capture more complex patterns. However, because increasing the embedding_dimension increases the number of parameters in the model, more training data is required to accurately learn these parameters. Typical values for this parameter are between 10-100. **Optional**  
Valid values: positive integer  
Default value: 10 |
| learning_rate       | The learning rate used in training. Typical values range from 1e-4 to 1e-1. **Optional**  
Valid values: float  
Default value: 1e-3 |
| likelihood          | The model generates a probabilistic forecast, and can provide quantiles of the distribution and return samples. Depending on your data, select an appropriate likelihood (noise model) that is used for uncertainty estimates. The following likelihoods can be selected:  
- **gaussian**: Use for real-valued data.  
- **beta**: Use for real-valued targets between 0 and 1 inclusive.  
- **negative-binomial**: Use for count data (non-negative integers).  
- **student-T**: An alternative for real-valued data that works well for bursty data.  
- **deterministic-L1**: A loss function that does not estimate uncertainty and only learns a point forecast. **Optional**  
Valid values: One of gaussian, beta, negative-binomial, student-T, or deterministic-L1.  
Default value: student-T |
<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mini_batch_size</td>
<td>The size of mini-batches used during training. Typical values range from 32 to 512.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: positive integer</td>
</tr>
<tr>
<td></td>
<td>Default value: 128</td>
</tr>
<tr>
<td>num_cells</td>
<td>The number of cells to use in each hidden layer of the RNN. Typical values range from 30 to 100.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: positive integer</td>
</tr>
<tr>
<td></td>
<td>Default value: 40</td>
</tr>
<tr>
<td>num_dynamic_feat</td>
<td>The number of dynamic_feat provided in the data. Set this to <code>auto</code> to infer the number of dynamic features from the data. The <code>auto</code> mode also works when no dynamic features are used in the dataset. This is the recommended setting for the parameter.</td>
</tr>
<tr>
<td></td>
<td>To force DeepAR to not use dynamic features, even if they are present in the data, set <code>num_dynamic_feat</code> to <code>ignore</code>.</td>
</tr>
<tr>
<td></td>
<td>To perform additional data validation, it is possible to explicitly set this parameter to the actual integer value. For example, if two dynamic features are provided, set this to 2.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: <code>auto</code>, <code>ignore</code>, positive integer, or empty string</td>
</tr>
<tr>
<td></td>
<td>Default value: <code>auto</code></td>
</tr>
<tr>
<td>num_eval_samples</td>
<td>The number of samples that are used per time-series when calculating test accuracy metrics. This parameter does not have any influence on the training or the final model. In particular, the model can be queried with a different number of samples. This parameter only affects the reported accuracy scores on the test channel after training. Smaller values result in faster evaluation, but then the evaluation scores are typically worse and more uncertain. When evaluating with higher quantiles, for example 0.95, it may be important to increase the number of evaluation samples.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: integer</td>
</tr>
<tr>
<td></td>
<td>Default value: 100</td>
</tr>
</tbody>
</table>
Tune a DeepAR Model

Automatic model tuning, also known as hyperparameter tuning, finds the best version of a model by running many jobs that test a range of hyperparameters on your dataset. You choose the tunable hyperparameters, a range of values for each, and an objective metric. You choose the objective metric from the metrics that the algorithm computes. Automatic model tuning searches the hyperparameters chosen to find the combination of values that result in the model that optimizes the objective metric.

For more information about model tuning, see Perform Automatic Model Tuning (p. 1023).

Metrics Computed by the DeepAR Algorithm

The DeepAR algorithm reports three metrics, which are computed during training. When tuning a model, choose one of these as the objective. For the objective, use either the forecast accuracy on a provided test channel (recommended) or the training loss. For recommendations for the training/test split for the DeepAR algorithm, see Best Practices for Using the DeepAR Algorithm (p. 669).

<table>
<thead>
<tr>
<th>Metric Name</th>
<th>Description</th>
<th>Optimization Direction</th>
</tr>
</thead>
<tbody>
<tr>
<td>test:RMSE</td>
<td>The root mean square error between the forecast and the actual target computed on the test set.</td>
<td>Minimize</td>
</tr>
<tr>
<td>test:mean_wQuantile</td>
<td>The average overall quantile losses computed on the test set. To control which quantiles are used, set the test_quantiles hyperparameter.</td>
<td>Minimize</td>
</tr>
<tr>
<td>train:final_loss</td>
<td>The training negative log-likelihood loss averaged over the last training epoch for the model.</td>
<td>Minimize</td>
</tr>
</tbody>
</table>

Tunable Hyperparameters for the DeepAR Algorithm

Tune a DeepAR model with the following hyperparameters. The hyperparameters that have the greatest impact, listed in order from the most to least impactful, on DeepAR objective metrics are: epochs, context_length, mini_batch_size, learning_rate, and num_cells.
<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Parameter Type</th>
<th>Recommended Ranges</th>
</tr>
</thead>
<tbody>
<tr>
<td>mini_batch_size</td>
<td>IntegerParameterRanges</td>
<td>MinValue: 32, MaxValue: 1028</td>
</tr>
<tr>
<td>epochs</td>
<td>IntegerParameterRanges</td>
<td>MinValue: 1, MaxValue: 1000</td>
</tr>
<tr>
<td>context_length</td>
<td>IntegerParameterRanges</td>
<td>MinValue: 1, MaxValue: 200</td>
</tr>
<tr>
<td>num_cells</td>
<td>IntegerParameterRanges</td>
<td>MinValue: 30, MaxValue: 200</td>
</tr>
<tr>
<td>num_layers</td>
<td>IntegerParameterRanges</td>
<td>MinValue: 1, MaxValue: 8</td>
</tr>
<tr>
<td>dropout_rate</td>
<td>ContinuousParameterRange</td>
<td>MinValue: 0.00, MaxValue: 0.2</td>
</tr>
<tr>
<td>embedding_dimension</td>
<td>IntegerParameterRanges</td>
<td>MinValue: 1, MaxValue: 50</td>
</tr>
<tr>
<td>learning_rate</td>
<td>ContinuousParameterRange</td>
<td>MinValue: 1e-5, MaxValue: 1e-1</td>
</tr>
</tbody>
</table>

**DeepAR Inference Formats**

**DeepAR JSON Request Formats**

Query a trained model by using the model's endpoint. The endpoint takes the following JSON request format.

In the request, the `instances` field corresponds to the time series that should be forecast by the model.

If the model was trained with categories, you must provide a `cat` for each instance. If the model was trained without the `cat` field, it should be omitted.

If the model was trained with a custom feature time series (`dynamic_feat`), you must provide the same number of `dynamic_feat` values for each instance. Each of them should have a length given by `length(target) + prediction_length`, where the last `prediction_length` values correspond to the time points in the future that will be predicted. If the model was trained without custom feature time series, the field should not be included in the request.

```json
{
    "instances": [
        {
            "start": "2009-11-01 00:00:00",
            "target": [4.0, 10.0, "NaN", 100.0, 113.0],
            "cat": [0, 1],
            "dynamic_feat": [[1.0, 1.1, 2.1, 0.5, 3.1, 4.1, 1.2, 5.0, ...]]
        },
        {
            "start": "2012-01-30",
            "target": [1.0],
            "cat": [2, 1],
            "dynamic_feat": [[2.0, 3.1, 4.5, 1.5, 1.8, 3, 0.1, 3, 0, ...]]
        },
        {
            "start": "1999-01-30",
```
Use built-in algorithms

```
"target": [2.0, 1.0],
"cat": [1, 3],
"dynamic_feat": [[1.0, 0.1, -2.5, 0.3, 2.0, -1.2, -0.1, -3.0, ...]]
}
"configuration": {
  "num_samples": 50,
  "output_types": ["mean", "quantiles", "samples"],
  "quantiles": ["0.5", "0.9"]
}
```

The `configuration` field is optional. `configuration.num_samples` sets the number of sample paths that the model generates to estimate the mean and quantiles. `configuration.output_types` describes the information that will be returned in the request. Valid values are "mean", "quantiles", and "samples". If you specify "quantiles", each of the quantile values in `configuration.quantiles` is returned as a time series. If you specify "samples", the model also returns the raw samples used to calculate the other outputs.

**DeepAR JSON Response Formats**

The following is the format of a response, where `[...]` are arrays of numbers:

```
{
  "predictions": [ 
    { 
      "quantiles": { 
        "0.9": [...],
        "0.5": [...]
      },
      "samples": [...],
      "mean": [...]
    },
    { 
      "quantiles": { 
        "0.9": [...],
        "0.5": [...]
      },
      "samples": [...],
      "mean": [...]
    },
    { 
      "quantiles": { 
        "0.9": [...],
        "0.5": [...]
      },
      "samples": [...],
      "mean": [...]
    }
  ]
}
```

DeepAR has a response timeout of 60 seconds. When passing multiple time series in a single request, the forecasts are generated sequentially. Because the forecast for each time series typically takes about 300 to 1000 milliseconds or longer, depending on the model size, passing too many time series in a single request can cause time outs. It's better to send fewer time series per request and send more requests. Because the DeepAR algorithm uses multiple workers per instance, you can achieve much higher throughput by sending multiple requests in parallel.

By default, DeepAR uses one worker per CPU for inference, if there is sufficient memory per CPU. If the model is large and there isn't enough memory to run a model on each CPU, the number of workers is reduced. The number of workers used for inference can be overwritten using the environment variable
Amazon SageMaker Developer Guide
Use built-in algorithms

MODEL_SERVER_WORKERS For example, by setting MODEL_SERVER_WORKERS=1) when calling the
SageMaker CreateModel API.

Batch Transform with the DeepAR Algorithm
DeepAR forecasting supports getting inferences by using batch transform from data using the JSON
Lines format. In this format, each record is represented on a single line as a JSON object, and lines
are separated by newline characters. The format is identical to the JSON Lines format used for model
training. For information, see Input/Output Interface for the DeepAR Algorithm (p. 667). For example:
{"start": "2009-11-01 00:00:00", "target": [4.3, "NaN", 5.1, ...], "cat": [0, 1],
"dynamic_feat": [[1.1, 1.2, 0.5, ..]]}
{"start": "2012-01-30 00:00:00", "target": [1.0, -5.0, ...], "cat": [2, 3], "dynamic_feat":
[[1.1, 2.05, ...]]}
{"start": "1999-01-30 00:00:00", "target": [2.0, 1.0], "cat": [1, 4], "dynamic_feat":
[[1.3, 0.4]]}

Note

When creating the transformation job with CreateTransformJob, set the BatchStrategy
value to SingleRecord and set the SplitType value in the TransformInput conﬁguration
to Line, as the default values currently cause runtime failures.
Similar to the hosted endpoint inference request format, the cat and the dynamic_feat ﬁelds for each
instance are required if both of the following are true:
• The model is trained on a dataset that contained both the cat and the dynamic_feat ﬁelds.
• The corresponding cardinality and num_dynamic_feat values used in the training job are not set
to "".
Unlike hosted endpoint inference, the conﬁguration ﬁeld is set once for the entire batch
inference job using an environment variable named DEEPAR_INFERENCE_CONFIG. The
value of DEEPAR_INFERENCE_CONFIG can be passed when the model is created by calling
CreateTransformJob API. If DEEPAR_INFERENCE_CONFIG is missing in the container environment,
the inference container uses the following default:
{

}

"num_samples": 100,
"output_types": ["mean", "quantiles"],
"quantiles": ["0.1", "0.2", "0.3", "0.4", "0.5", "0.6", "0.7", "0.8", "0.9"]

The output is also in JSON Lines format, with one line per prediction, in an order identical to the instance
order in the corresponding input ﬁle. Predictions are encoded as objects identical to the ones returned by
responses in online inference mode. For example:
{ "quantiles": { "0.1": [...], "0.2": [...] }, "samples": [...], "mean": [...] }

Note that in the TransformInput conﬁguration of the SageMaker CreateTransformJob request
clients must explicitly set the AssembleWith value to Line, as the default value None concatenates all
JSON objects on the same line.
For example, here is a SageMaker CreateTransformJob request for a DeepAR job with a custom
DEEPAR_INFERENCE_CONFIG:
{

"BatchStrategy": "SingleRecord",
"Environment": {
"DEEPAR_INFERENCE_CONFIG" : "{ \"num_samples\": 200, \"output_types\": [\"mean\"] }",

680


Factorization Machines Algorithm

A factorization machine is a general-purpose supervised learning algorithm that you can use for both classification and regression tasks. It is an extension of a linear model that is designed to capture interactions between features within high dimensional sparse datasets economically. For example, in a click prediction system, the factorization machine model can capture click rate patterns observed when ads from a certain ad-category are placed on pages from a certain page-category. Factorization machines are a good choice for tasks dealing with high dimensional sparse datasets, such as click prediction and item recommendation.

**Note**
The Amazon SageMaker implementation of factorization machines considers only pair-wise (2nd order) interactions between features.

**Topics**
- Input/Output Interface for the Factorization Machines Algorithm (p. 681)
- EC2 Instance Recommendation for the Factorization Machines Algorithm (p. 682)
- Factorization Machines Sample Notebooks (p. 682)
- How Factorization Machines Work (p. 682)
- Factorization Machines Hyperparameters (p. 683)
- Tune a Factorization Machines Model (p. 688)
- Factorization Machine Response Formats (p. 689)

**Input/Output Interface for the Factorization Machines Algorithm**

The factorization machine algorithm can be run in either in binary classification mode or regression mode. In each mode, a dataset can be provided to the test channel along with the train channel dataset. The scoring depends on the mode used. In regression mode, the testing dataset is scored using Root Mean Square Error (RMSE). In binary classification mode, the test dataset is scored using Binary Cross Entropy (Log Loss), Accuracy (at threshold=0.5) and F1 Score (at threshold =0.5).

For training, the factorization machines algorithm currently supports only the recordIO-protobuf format with Float32 tensors. Because their use case is predominantly on sparse data, CSV is not a good candidate. Both File and Pipe mode training are supported for recordIO-wrapped protobuf.

For inference, factorization machines support the application/json and x-recordio-protobuf formats.

- For the **binary classification** problem, the algorithm predicts a score and a label. The label is a number and can be either 0 or 1. The score is a number that indicates how strongly the algorithm believes that the label should be 1. The algorithm computes score first and then derives the label from the score value. If the score is greater than or equal to 0.5, the label is 1.
- For the **regression** problem, just a score is returned and it is the predicted value. For example, if Factorization Machines is used to predict a movie rating, score is the predicted rating value.
Please see Factorization Machines Sample Notebooks (p. 682) for more details on training and inference file formats.

**EC2 Instance Recommendation for the Factorization Machines Algorithm**

The Amazon SageMaker Factorization Machines algorithm is highly scalable and can train across distributed instances. We recommend training and inference with CPU instances for both sparse and dense datasets. In some circumstances, training with one or more GPUs on dense data might provide some benefit. Training with GPUs is available only on dense data. Use CPU instances for sparse data.

**Factorization Machines Sample Notebooks**

For a sample notebook that uses the SageMaker factorization machine learning algorithm to analyze the images of handwritten digits from zero to nine in the MNIST dataset, see An Introduction to Factorization Machines with MNIST. For instructions how to create and access Jupyter notebook instances that you can use to run the example in SageMaker, see Use Amazon SageMaker Notebook Instances (p. 124). Once you have created a notebook instance and opened it, select the SageMaker Examples tab to see a list of all the SageMaker samples. Example notebooks that use Factorization Machines algorithm are located in the Introduction to Amazon algorithms section. To open a notebook, click on its Use tab and select Create copy.

**How Factorization Machines Work**

The prediction task for a factorization machine model is to estimate a function $\hat{y}$ from a feature set $x_i$ to a target domain. This domain is real-valued for regression and binary for classification. The factorization machine model is supervised and so has a training dataset $(x_i, y_j)$ available. The advantages this model presents lie in the way it uses a factorized parametrization to capture the pairwise feature interactions. It can be represented mathematically as follows:

$$\hat{y} = w_0 + \sum_i w_i x_i + \sum_i \sum_{j > i} <v_i, v_j> x_i x_j$$

The three terms in this equation correspond respectively to the three components of the model:

- The $w_0$ term represents the global bias.
- The $w_i$ linear terms model the strength of the $i^{th}$ variable.
- The $<v_i, v_j>$ factorization terms model the pairwise interaction between the $i^{th}$ and $j^{th}$ variable.

The global bias and linear terms are the same as in a linear model. The pairwise feature interactions are modeled in the third term as the inner product of the corresponding factors learned for each feature. Learned factors can also be considered as embedding vectors for each feature. For example, in a classification task, if a pair of features tends to co-occur more often in positive labeled samples, then the inner product of their factors would be large. In other words, their embedding vectors would be close to each other in cosine similarity. For more information about the factorization machine model, see Factorization Machines.

For regression tasks, the model is trained by minimizing the squared error between the model prediction $\hat{y}_n$ and the target value $y_n$. This is known as the square loss:

$$L = \frac{1}{N} \sum_n (y_n - \hat{y}_n)^2$$

For a classification task, the model is trained by minimizing the cross entropy loss, also known as the log loss:

$$L = \frac{1}{N} \sum_n [y_n \log \hat{p}_n + (1 - y_n) \log (1 - \hat{p}_n)]$$
where:

\[ \hat{p}_n = \frac{1}{1 + e^{-\theta_n}} \]

For more information about loss functions for classification, see Loss functions for classification.

**Factorization Machines Hyperparameters**

The following table contains the hyperparameters for the factorization machines algorithm. These are parameters that are set by users to facilitate the estimation of model parameters from data. The required hyperparameters that must be set are listed first, in alphabetical order. The optional hyperparameters that can be set are listed next, also in alphabetical order.

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>feature_dim</td>
<td>The dimension of the input feature space. This could be very high with sparse input.</td>
</tr>
<tr>
<td></td>
<td><strong>Required</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: Positive integer. Suggested value range: [10000,10000000]</td>
</tr>
<tr>
<td>num_factors</td>
<td>The dimensionality of factorization.</td>
</tr>
<tr>
<td></td>
<td><strong>Required</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: Positive integer. Suggested value range: [2,1000], 64 is usually optimal.</td>
</tr>
<tr>
<td>predictor_type</td>
<td>The type of predictor.</td>
</tr>
<tr>
<td></td>
<td>• binary_classifier: For binary classification tasks.</td>
</tr>
<tr>
<td></td>
<td>• regressor: For regression tasks.</td>
</tr>
<tr>
<td></td>
<td><strong>Required</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: String: binary_classifier or regressor</td>
</tr>
<tr>
<td>bias_init_method</td>
<td>The initialization method for the bias term:</td>
</tr>
<tr>
<td></td>
<td>• normal: Initializes weights with random values sampled from a normal distribution with a mean of zero and standard deviation specified by bias_init_sigma.</td>
</tr>
<tr>
<td></td>
<td>• uniform: Initializes weights with random values uniformly sampled from a range specified by [-bias_init_scale, +bias_init_scale].</td>
</tr>
<tr>
<td></td>
<td>• constant: Initializes the weights to a scalar value specified by bias_init_value.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: uniform, normal, or constant</td>
</tr>
<tr>
<td></td>
<td>Default value: normal</td>
</tr>
<tr>
<td>Parameter Name</td>
<td>Description</td>
</tr>
<tr>
<td>------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>bias_init_scale</td>
<td>Range for initialization of the bias term. Takes effect if bias_init_method is set to uniform. Optional Valid values: Non-negative float. Suggested value range: [1e-8, 512]. Default value: None</td>
</tr>
<tr>
<td>bias_init_sigma</td>
<td>The standard deviation for initialization of the bias term. Takes effect if bias_init_method is set to normal. Optional Valid values: Non-negative float. Suggested value range: [1e-8, 512]. Default value: 0.01</td>
</tr>
<tr>
<td>bias_init_value</td>
<td>The initial value of the bias term. Takes effect if bias_init_method is set to constant. Optional Valid values: Float. Suggested value range: [1e-8, 512]. Default value: None</td>
</tr>
<tr>
<td>bias_lr</td>
<td>The learning rate for the bias term. Optional Valid values: Non-negative float. Suggested value range: [1e-8, 512]. Default value: 0.1</td>
</tr>
<tr>
<td>bias_wd</td>
<td>The weight decay for the bias term. Optional Valid values: Non-negative float. Suggested value range: [1e-8, 512]. Default value: 0.01</td>
</tr>
<tr>
<td>clip_gradient</td>
<td>Gradient clipping optimizer parameter. Clips the gradient by projecting onto the interval [-clip_gradient, +clip_gradient]. Optional Valid values: Float Default value: None</td>
</tr>
<tr>
<td>Parameter Name</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------</td>
<td>------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>epochs</td>
<td>The number of training epochs to run.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: Positive integer</td>
</tr>
<tr>
<td></td>
<td>Default value: 1</td>
</tr>
<tr>
<td>eps</td>
<td>Epsilon parameter to avoid division by 0.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: Float. Suggested value: small.</td>
</tr>
<tr>
<td></td>
<td>Default value: None</td>
</tr>
<tr>
<td>factors_init_method</td>
<td>The initialization method for factorization terms:</td>
</tr>
<tr>
<td></td>
<td>• normal: Initializes weights with random values sampled from a normal distribution with a mean of zero and standard deviation specified by factors_init_sigma.</td>
</tr>
<tr>
<td></td>
<td>• uniform: Initializes weights with random values uniformly sampled from a range specified by [-factors_init_scale, +factors_init_scale].</td>
</tr>
<tr>
<td></td>
<td>• constant: Initializes the weights to a scalar value specified by factors_init_value.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: uniform, normal, or constant.</td>
</tr>
<tr>
<td></td>
<td>Default value: normal</td>
</tr>
<tr>
<td>factors_init_scale</td>
<td>The range for initialization of factorization terms. Takes effect if factors_init_method is set to uniform.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: Non-negative float. Suggested value range: [1e-8, 512].</td>
</tr>
<tr>
<td></td>
<td>Default value: None</td>
</tr>
<tr>
<td>factors_init_sigma</td>
<td>The standard deviation for initialization of factorization terms. Takes effect if factors_init_method is set to normal.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: Non-negative float. Suggested value range: [1e-8, 512].</td>
</tr>
<tr>
<td></td>
<td>Default value: 0.001</td>
</tr>
<tr>
<td>Parameter Name</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------</td>
<td>-------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>factors_init_value</td>
<td>The initial value of factorization terms. Takes effect if factors_init_method is set to constant.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: Float. Suggested value range: [1e-8, 512].</td>
</tr>
<tr>
<td></td>
<td>Default value: None</td>
</tr>
<tr>
<td>factors_lr</td>
<td>The learning rate for factorization terms.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: Non-negative float. Suggested value range: [1e-8, 512].</td>
</tr>
<tr>
<td></td>
<td>Default value: 0.0001</td>
</tr>
<tr>
<td>factors_wd</td>
<td>The weight decay for factorization terms.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: Non-negative float. Suggested value range: [1e-8, 512].</td>
</tr>
<tr>
<td></td>
<td>Default value: 0.00001</td>
</tr>
<tr>
<td>linear_lr</td>
<td>The learning rate for linear terms.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: Non-negative float. Suggested value range: [1e-8, 512].</td>
</tr>
<tr>
<td></td>
<td>Default value: 0.001</td>
</tr>
<tr>
<td>linear_init_method</td>
<td>The initialization method for linear terms:</td>
</tr>
<tr>
<td></td>
<td>• <em>normal</em> Initializes weights with random values sampled from a normal distribution with a mean of zero and standard deviation specified by linear_init_sigma.</td>
</tr>
<tr>
<td></td>
<td>• <em>uniform</em> Initializes weights with random values uniformly sampled from a range specified by [-linear_init_scale, +linear_init_scale].</td>
</tr>
<tr>
<td></td>
<td>• <em>constant</em> Initializes the weights to a scalar value specified by linear_init_value.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: uniform, normal, or constant.</td>
</tr>
<tr>
<td></td>
<td>Default value: normal</td>
</tr>
<tr>
<td>Parameter Name</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>linear_init_scale</td>
<td>Range for initialization of linear terms. Takes effect if <code>linear_init_method</code> is set to <code>uniform</code>.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: Non-negative float. Suggested value range: [1e-8, 512].</td>
</tr>
<tr>
<td></td>
<td>Default value: None</td>
</tr>
<tr>
<td>linear_init_sigma</td>
<td>The standard deviation for initialization of linear terms. Takes effect if <code>linear_init_method</code> is set to <code>normal</code>.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: Non-negative float. Suggested value range: [1e-8, 512].</td>
</tr>
<tr>
<td></td>
<td>Default value: 0.01</td>
</tr>
<tr>
<td>linear_init_value</td>
<td>The initial value of linear terms. Takes effect if <code>linear_init_method</code> is set to <code>constant</code>.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: Float. Suggested value range: [1e-8, 512].</td>
</tr>
<tr>
<td></td>
<td>Default value: None</td>
</tr>
<tr>
<td>linear_wd</td>
<td>The weight decay for linear terms.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: Non-negative float. Suggested value range: [1e-8, 512].</td>
</tr>
<tr>
<td></td>
<td>Default value: 0.001</td>
</tr>
<tr>
<td>mini_batch_size</td>
<td>The size of mini-batch used for training.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: Positive integer</td>
</tr>
<tr>
<td></td>
<td>Default value: 1000</td>
</tr>
<tr>
<td>rescale_grad</td>
<td>Gradient rescaling optimizer parameter. If set, multiplies the gradient with <code>rescale_grad</code> before updating. Often choose to be 1.0/batch_size.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: Float</td>
</tr>
<tr>
<td></td>
<td>Default value: None</td>
</tr>
</tbody>
</table>
Tune a Factorization Machines Model

Automatic model tuning, also known as hyperparameter tuning, finds the best version of a model by running many jobs that test a range of hyperparameters on your dataset. You choose the tunable hyperparameters, a range of values for each, and an objective metric. You choose the objective metric from the metrics that the algorithm computes. Automatic model tuning searches the hyperparameters chosen to find the combination of values that result in the model that optimizes the objective metric.

For more information about model tuning, see Perform Automatic Model Tuning (p. 1023).

Metrics Computed by the Factorization Machines Algorithm

The factorization machines algorithm has both binary classification and regression predictor types. The predictor type determines which metric you can use for automatic model tuning. The algorithm reports a \text{test:rmse} regressor metric, which is computed during training. When tuning the model for regression tasks, choose this metric as the objective.

<table>
<thead>
<tr>
<th>Metric Name</th>
<th>Description</th>
<th>Optimization Direction</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{test:rmse}</td>
<td>Root Mean Square Error</td>
<td>Minimize</td>
</tr>
</tbody>
</table>

The factorization machines algorithm reports three binary classification metrics, which are computed during training. When tuning the model for binary classification tasks, choose one of these as the objective.

<table>
<thead>
<tr>
<th>Metric Name</th>
<th>Description</th>
<th>Optimization Direction</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{test:binary_classification_accuracy}</td>
<td>Accuracy</td>
<td>Maximize</td>
</tr>
<tr>
<td>\text{test:binary_classification_cross_entropy}</td>
<td>Cross Entropy</td>
<td>Minimize</td>
</tr>
<tr>
<td>\text{test:binary_f_beta}</td>
<td>Beta</td>
<td>Maximize</td>
</tr>
</tbody>
</table>

Tunable Factorization Machines Hyperparameters

You can tune the following hyperparameters for the factorization machines algorithm. The initialization parameters that contain the terms bias, linear, and factorization depend on their initialization method. There are three initialization methods: uniform, normal, and constant. These initialization methods are not themselves tunable. The parameters that are tunable are dependent on this choice of the initialization method. For example, if the initialization method is uniform, then only the scale parameters are tunable. Specifically, if \text{bias_init_method=uniform}, then \text{bias_init_scale, linear_init_scale, and factors_init_scale} are tunable. Similarly, if the initialization method is normal, then only \text{sigma} parameters are tunable. If the initialization method is constant, then only \text{value} parameters are tunable. These dependencies are listed in the following table.

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Parameter Type</th>
<th>Recommended Ranges</th>
<th>Dependency</th>
</tr>
</thead>
<tbody>
<tr>
<td>bias_init_scale</td>
<td>ContinuousParameterRange</td>
<td>MinValue: 1e-8, MaxValue: 512</td>
<td>\text{bias_init_method=uniform}</td>
</tr>
<tr>
<td>bias_init_sigma</td>
<td>ContinuousParameterRange</td>
<td>MinValue: 1e-8, MaxValue: 512</td>
<td>\text{bias_init_method=normal}</td>
</tr>
<tr>
<td>bias_init_value</td>
<td>ContinuousParameterRange</td>
<td>MinValue: 1e-8, MaxValue: 512</td>
<td>\text{bias_init_method=constant}</td>
</tr>
<tr>
<td>Parameter Name</td>
<td>Parameter Type</td>
<td>Recommended Ranges</td>
<td>Dependency</td>
</tr>
<tr>
<td>------------------</td>
<td>---------------------------</td>
<td>-----------------------------------</td>
<td>-------------------------------------</td>
</tr>
<tr>
<td>bias_lr</td>
<td>ContinuousParameterRange</td>
<td>MinValue: 1e-8, MaxValue: 512</td>
<td>None</td>
</tr>
<tr>
<td>bias_wd</td>
<td>ContinuousParameterRange</td>
<td>MinValue: 1e-8, MaxValue: 512</td>
<td>None</td>
</tr>
<tr>
<td>epoch</td>
<td>IntegerParameterRange</td>
<td>MinValue: 1, MaxValue: 1000</td>
<td>None</td>
</tr>
<tr>
<td>factors_init_scale</td>
<td>ContinuousParameterRange</td>
<td>MinValue: 1e-8, MaxValue: 512</td>
<td>bias_init_method==uniform</td>
</tr>
<tr>
<td>factors_init_sigma</td>
<td>ContinuousParameterRange</td>
<td>MinValue: 1e-8, MaxValue: 512</td>
<td>bias_init_method==normal</td>
</tr>
<tr>
<td>factors_init_value</td>
<td>ContinuousParameterRange</td>
<td>MinValue: 1e-8, MaxValue: 512</td>
<td>bias_init_method==constant</td>
</tr>
<tr>
<td>factors_lr</td>
<td>ContinuousParameterRange</td>
<td>MinValue: 1e-8, MaxValue: 512</td>
<td>None</td>
</tr>
<tr>
<td>factors_wd</td>
<td>ContinuousParameterRange</td>
<td>MinValue: 1e-8, MaxValue: 512</td>
<td>None</td>
</tr>
<tr>
<td>linear_init_scale</td>
<td>ContinuousParameterRange</td>
<td>MinValue: 1e-8, MaxValue: 512</td>
<td>bias_init_method==uniform</td>
</tr>
<tr>
<td>linear_init_sigma</td>
<td>ContinuousParameterRange</td>
<td>MinValue: 1e-8, MaxValue: 512</td>
<td>bias_init_method==normal</td>
</tr>
<tr>
<td>linear_init_value</td>
<td>ContinuousParameterRange</td>
<td>MinValue: 1e-8, MaxValue: 512</td>
<td>bias_init_method==constant</td>
</tr>
<tr>
<td>linear_lr</td>
<td>ContinuousParameterRange</td>
<td>MinValue: 1e-8, MaxValue: 512</td>
<td>None</td>
</tr>
<tr>
<td>linear_wd</td>
<td>ContinuousParameterRange</td>
<td>MinValue: 1e-8, MaxValue: 512</td>
<td>None</td>
</tr>
<tr>
<td>mini_batch_size</td>
<td>IntegerParameterRange</td>
<td>MinValue: 100, MaxValue: 10000</td>
<td>None</td>
</tr>
</tbody>
</table>

**Factorization Machine Response Formats**

**JSON Response Format**

Binary classification

```javascript
let response = {
  "predictions": [
    {
      "score": 0.4,
      "predicted_label": 0
    }
  ]
}
```
Regression

```javascript
let response = {
  "predictions": [
    {
      "score": 0.4
    }
  ]
}
```

JSONLINES Response Format

Binary classification

```json
{"score": 0.4, "predicted_label": 0}
```

Regression

```json
{"score": 0.4}
```

RECORDIO Response Format

Binary classification

```json
[
  Record = {
    features = {},
    label = {
      'score': {
        keys: [],
        values: [0.4]  # float32
      },
      'predicted_label': {
        keys: [],
        values: [0.0]  # float32
      }
    }
  }
]
```

Regression

```json
[
  Record = {
    features = {},
    label = {
      'score': {
        keys: [],
        values: [0.4]  # float32
      }
    }
  }
]
```

Image Classification Algorithm

The Amazon SageMaker image classification algorithm is a supervised learning algorithm that supports multi-label classification. It takes an image as input and outputs one or more labels assigned to that image. It uses a convolutional neural network (ResNet) that can be trained from scratch or trained using transfer learning when a large number of training images are not available.
The recommended input format for the Amazon SageMaker image classification algorithms is Apache MXNet RecordIO. However, you can also use raw images in .jpg or .png format. Refer to this discussion for a broad overview of efficient data preparation and loading for machine learning systems.

**Note**
To maintain better interoperability with existing deep learning frameworks, this differs from the protobuf data formats commonly used by other Amazon SageMaker algorithms.

For more information on convolutional networks, see:

- Deep residual learning for image recognition Kaiming He, et al., 2016 IEEE Conference on Computer Vision and Pattern Recognition
- ImageNet image database
- Image classification with Gluon-CV and MXNet

**Topics**
- Input/Output Interface for the Image Classification Algorithm (p. 691)
- EC2 Instance Recommendation for the Image Classification Algorithm (p. 693)
- Image Classification Sample Notebooks (p. 693)
- How Image Classification Works (p. 694)
- Image Classification Hyperparameters (p. 694)
- Tune an Image Classification Model (p. 700)

**Input/Output Interface for the Image Classification Algorithm**

The SageMaker Image Classification algorithm supports both RecordIO (application/x-recordio) and image (image/png, image/jpeg, and application/x-image) content types for training in file mode, and supports the RecordIO (application/x-recordio) content type for training in pipe mode. However, you can also train in pipe mode using the image files (image/png, image/jpeg, and application/x-image), without creating RecordIO files, by using the augmented manifest format.

Distributed training is supported for file mode and pipe mode. When using the RecordIO content type in pipe mode, you must set the S3DataDistributionType of the S3DataSource to FullyReplicated. The algorithm supports a fully replicated model where your data is copied onto each machine.

The algorithm supports image/png, image/jpeg, and application/x-image for inference.

**Train with RecordIO Format**

If you use the RecordIO format for training, specify both train and validation channels as values for the InputDataConfig parameter of the CreateTrainingJob request. Specify one RecordIO (.rec) file in the train channel and one RecordIO file in the validation channel. Set the content type for both channels to application/x-recordio.

**Train with Image Format**

If you use the Image format for training, specify train, validation, train_lst, and validation_lst channels as values for the InputDataConfig parameter of the CreateTrainingJob request. Specify the individual image data (.jpg or .png files) for the train and validation channels. Specify one .lst file in each of the train_lst and validation_lst channels. Set the content type for all four channels to application/x-image.

**Note**
SageMaker reads the training and validation data separately from different channels, so you must store the training and validation data in different folders.

A .lst file is a tab-separated file with three columns that contains a list of image files. The first column specifies the image index, the second column specifies the class label index for the image, and the third
column specifies the relative path of the image file. The image index in the first column must be unique across all of the images. The set of class label indices are numbered successively and the numbering should start with 0. For example, 0 for the cat class, 1 for the dog class, and so on for additional classes.

The following is an example of a .lst file:

```
5      1   your_image_directory/train_img_dog1.jpg
1000   0   your_image_directory/train_img_cat1.jpg
22     1   your_image_directory/train_img_dog2.jpg
```

For example, if your training images are stored in s3://<your_bucket>/train/class_dog, s3://<your_bucket>/train/class_cat, and so on, specify the path for your train channel as s3://<your_bucket>/train, which is the top-level directory for your data. In the .lst file, specify the relative path for an individual file named train_image_dog1.jpg in the class_dog class directory as class_dog/train_image_dog1.jpg. You can also store all your image files under one subdirectory inside the train directory. In that case, use that subdirectory for the relative path. For example, s3://<your_bucket>/train/your_image_directory.

**Train with Augmented Manifest Image Format**

The augmented manifest format enables you to do training in Pipe mode using image files without needing to create RecordIO files. You need to specify both train and validation channels as values for the InputDataConfig parameter of the CreateTrainingJob request. While using the format, an S3 manifest file needs to be generated that contains the list of images and their corresponding annotations. The manifest file format should be in JSON Lines format in which each line represents one sample. The images are specified using the 'source-ref' tag that points to the S3 location of the image. The annotations are provided under the "AttributeNames" parameter value as specified in the CreateTrainingJob request. It can also contain additional metadata under the metadata tag, but these are ignored by the algorithm. In the following example, the "AttributeNames" are contained in the list of image and annotation references ['"source-ref", "class"']. The corresponding label value is "0" for the first image and "1" for the second image:

```
{"source-ref":"s3://image/filename1.jpg", "class":"0"}
{"source-ref":"s3://image/filename2.jpg", "class":"1", "class-metadata": {"class-name": "cat", "type" : "groundtruth/image-classification"}}
```

The order of "AttributeNames" in the input files matters when training the ImageClassification algorithm. It accepts piped data in a specific order, with image first, followed by label. So the "AttributeNames" in this example are provided with "source-ref" first, followed by "class".

When using the ImageClassification algorithm with Augmented Manifest, the value of the RecordWrapperType parameter must be "RecordIO".

Multi-label training is also supported by specifying a JSON array of values. The num_classes hyperparameter must be set to match the total number of classes. There are two valid label formats: multi-hot and class-id.

In the multi-hot format, each label is a multi-hot encoded vector of all classes, where each class takes the value of 0 or 1. In the following example, there are three classes. The first image is labeled with classes 0 and 2, while the second image is labeled with class 2 only:

```
{"image-ref": "s3://mybucket/sample01/image1.jpg", "class": [1, 0, 1]}
{"image-ref": "s3://mybucket/sample02/image2.jpg", "class": [0, 0, 1]}
```

In the class-id format, each label is a list of the class ids, from [0, num_classes), which apply to the data point. The previous example would instead look like this:

```
{"image-ref": "s3://mybucket/sample01/image1.jpg", "class": [0, 2]}
{"image-ref": "s3://mybucket/sample02/image2.jpg", "class": [2]}
```
The multi-hot format is the default, but can be explicitly set in the content type with the `label-format` parameter: "application/x-recordio; label-format=multi-hot". The class-id format, which is the format outputted by GroundTruth, must be set explicitly: "application/x-recordio; label-format=class-id".

For more information on augmented manifest files, see [Provide Dataset Metadata to Training Jobs with an Augmented Manifest File](p. 1124).

**Incremental Training**

You can also seed the training of a new model with the artifacts from a model that you trained previously with SageMaker. Incremental training saves training time when you want to train a new model with the same or similar data. SageMaker image classification models can be seeded only with another built-in image classification model trained in SageMaker.

To use a pretrained model, in the `CreateTrainingJob` request, specify the `ChannelName` as "model" in the `InputDataConfig` parameter. Set the `ContentType` for the model channel to `application/x-sagemaker-model`. The input hyperparameters of both the new model and the pretrained model that you upload to the model channel must have the same settings for the `num_layers`, `image_shape` and `num_classes` input parameters. These parameters define the network architecture. For the pretrained model file, use the compressed model artifacts (in .tar.gz format) output by SageMaker. You can use either RecordIO or image formats for input data.

For a sample notebook that shows how to use incremental training with the SageMaker image classification algorithm, see the [End-to-End Incremental Training Image Classification Example](p. 1118). For more information on incremental training and for instructions on how to use it, see [Incremental Training in Amazon SageMaker](p. 1118).

**Inference with the Image Classification Algorithm**

The generated models can be hosted for inference and support encoded .jpg and .png image formats as `image/png`, `image/jpeg`, and `application/x-image` content-type. The input image is resized automatically. The output is the probability values for all classes encoded in JSON format, or in JSON Lines text format for batch transform. The image classification model processes a single image per request and so outputs only one line in the JSON or JSON Lines format. The following is an example of a response in JSON Lines format:

```json
accept: application/jsonlines

{"prediction": [prob_0, prob_1, prob_2, prob_3, ...]}
```

For more details on training and inference, see the image classification sample notebook instances referenced in the introduction.

**EC2 Instance Recommendation for the Image Classification Algorithm**

For image classification, we support the following GPU instances for training: ml.p2.xlarge, ml.p2.8xlarge, ml.p2.16xlarge, ml.p3.2xlarge, ml.p3.8xlarge, and ml.p3.16xlarge. We recommend using GPU instances with more memory for training with large batch sizes. However, both CPU (such as C4) and GPU (such as P2 and P3) instances can be used for the inference. You can also run the algorithm on multi-GPU and multi-machine settings for distributed training.

Both P2 and P3 instances are supported in the image classification algorithm.

**Image Classification Sample Notebooks**

For a sample notebook that uses the SageMaker image classification algorithm to train a model on the caltech-256 dataset and then to deploy it to perform inferences, see the [End-to-End Multiclass Image Classification Example](p. 1118). For instructions how to create and access Jupyter notebook instances that you can use to run the example in SageMaker, see [Use Amazon SageMaker Notebook Instances](p. 124).
you have created a notebook instance and opened it, select the **SageMaker Examples** tab to see a list of all the SageMaker samples. The example image classification notebooks are located in the **Introduction to Amazon algorithms** section. To open a notebook, click on its **Use** tab and select **Create copy**.

**How Image Classification Works**

The image classification algorithm takes an image as input and classifies it into one of the output categories. Deep learning has revolutionized the image classification domain and has achieved great performance. Various deep learning networks such as ResNet [1], DenseNet, inception, and so on, have been developed to be highly accurate for image classification. At the same time, there have been efforts to collect labeled image data that are essential for training these networks. ImageNet[2] is one such large dataset that has more than 11 million images with about 11,000 categories. Once a network is trained with ImageNet data, it can then be used to generalize with other datasets as well, by simple re-adjustment or fine-tuning. In this transfer learning approach, a network is initialized with weights (in this example, trained on ImageNet), which can be later fine-tuned for an image classification task in a different dataset.

Image classification in Amazon SageMaker can be run in two modes: full training and transfer learning. In full training mode, the network is initialized with random weights and trained on user data from scratch. In transfer learning mode, the network is initialized with pre-trained weights and just the top fully connected layer is initialized with random weights. Then, the whole network is fine-tuned with new data. In this mode, training can be achieved even with a smaller dataset. This is because the network is already trained and therefore can be used in cases without sufficient training data.

**Image Classification Hyperparameters**

Hyperparameters are parameters that are set before a machine learning model begins learning. The following hyperparameters are supported by the Amazon SageMaker built-in Image Classification algorithm. See **Tune an Image Classification Model (p. 700)** for information on image classification hyperparameter tuning.

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>num_classes</td>
<td>Number of output classes. This parameter defines the dimensions of the network output and is typically set to the number of classes in the dataset.</td>
</tr>
<tr>
<td></td>
<td>Besides multi-class classification, multi-label classification is supported too. Please refer to <strong>Input/Output Interface for the Image Classification Algorithm (p. 691)</strong> for details on how to work with multi-label classification with augmented manifest files.</td>
</tr>
<tr>
<td></td>
<td><strong>Required</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: positive integer</td>
</tr>
<tr>
<td>num_training_samples</td>
<td>Number of training examples in the input dataset.</td>
</tr>
<tr>
<td></td>
<td>If there is a mismatch between this value and the number of samples in the training set, then the behavior of the <strong>lr_scheduler_step</strong> parameter is undefined and distributed training accuracy might be affected.</td>
</tr>
<tr>
<td></td>
<td><strong>Required</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: positive integer</td>
</tr>
<tr>
<td>augmentation_type</td>
<td>Data augmentation type. The input images can be augmented in multiple ways as specified below.</td>
</tr>
<tr>
<td>Parameter Name</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
|                         | • **crop**: Randomly crop the image and flip the image horizontally  
|                         | • **crop_color**: In addition to ‘crop’, three random values in the range [-36, 36], [-50, 50], and [-50, 50] are added to the corresponding Hue-Saturation-Lightness channels respectively  
|                         | • **crop_color_transform**: In addition to **crop_color**, random transformations, including rotation, shear, and aspect ratio variations are applied to the image. The maximum angle of rotation is 10 degrees, the maximum shear ratio is 0.1, and the maximum aspect changing ratio is 0.25.                                                                                                                      |
| **Optional**            | **Valid values**: **crop**, **crop_color**, or **crop_color_transform**  
|                         | **Default value**: no default value                                                                                                                                                                                                                                                                                                        |
| **beta_1**              | The beta1 for **adam**, that is the exponential decay rate for the first moment estimates.                                                                                                                                                                                                                                                   |
| **Optional**            | **Valid values**: float. Range in [0, 1].  
|                         | **Default value**: 0.9                                                                                                                                                                                                                                                                                                                        |
| **beta_2**              | The beta2 for **adam**, that is the exponential decay rate for the second moment estimates.                                                                                                                                                                                                                                                  |
| **Optional**            | **Valid values**: float. Range in [0, 1].  
|                         | **Default value**: 0.999                                                                                                                                                                                                                                                                                                                       |
| **checkpoint_frequency**| Period to store model parameters (in number of epochs).  
|                         | Note that all checkpoint files are saved as part of the final model file "model.tar.gz" and uploaded to S3 to the specified model location. This increases the size of the model file proportionally to the number of checkpoints saved during training.                                                                                                                                                                         |
| **Optional**            | **Valid values**: positive integer no greater than **epochs**.  
|                         | **Default value**: no default value (Save checkpoint at the epoch that has the best validation accuracy)                                                                                                                                                                                                                     |
| **early_stopping**      | True to use early stopping logic during training. **False** not to use it.                                                                                                                                                                                                                                                                 |
| **Optional**            | **Valid values**: **True** or **False**  
|                         | **Default value**: **False**
<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>early_stopping_min_epochs</td>
<td>The minimum number of epochs that must be run before the early stopping logic can be invoked. It is used only when early_stopping = True.</td>
</tr>
<tr>
<td></td>
<td>Optional</td>
</tr>
<tr>
<td></td>
<td>Valid values: positive integer</td>
</tr>
<tr>
<td></td>
<td>Default value: 10</td>
</tr>
<tr>
<td>early_stopping_patience</td>
<td>The number of epochs to wait before ending training if no improvement is made in the relevant metric. It is used only when early_stopping = True.</td>
</tr>
<tr>
<td></td>
<td>Optional</td>
</tr>
<tr>
<td></td>
<td>Valid values: positive integer</td>
</tr>
<tr>
<td></td>
<td>Default value: 5</td>
</tr>
<tr>
<td>early_stopping_tolerance</td>
<td>Relative tolerance to measure an improvement in accuracy validation metric. If the ratio of the improvement in accuracy divided by the previous best accuracy is smaller than the early_stopping_tolerance value set, early stopping considers there is no improvement. It is used only when early_stopping = True.</td>
</tr>
<tr>
<td></td>
<td>Optional</td>
</tr>
<tr>
<td></td>
<td>Valid values: 0 ≤ float ≤ 1</td>
</tr>
<tr>
<td></td>
<td>Default value: 0.0</td>
</tr>
<tr>
<td>epochs</td>
<td>Number of training epochs.</td>
</tr>
<tr>
<td></td>
<td>Optional</td>
</tr>
<tr>
<td></td>
<td>Valid values: positive integer</td>
</tr>
<tr>
<td></td>
<td>Default value: 30</td>
</tr>
<tr>
<td>eps</td>
<td>The epsilon for adam and rmsprop. It is usually set to a small value to avoid division by 0.</td>
</tr>
<tr>
<td></td>
<td>Optional</td>
</tr>
<tr>
<td></td>
<td>Valid values: float. Range in [0, 1].</td>
</tr>
<tr>
<td></td>
<td>Default value: 1e-8</td>
</tr>
<tr>
<td>gamma</td>
<td>The gamma for rmsprop, the decay factor for the moving average of the squared gradient.</td>
</tr>
<tr>
<td></td>
<td>Optional</td>
</tr>
<tr>
<td></td>
<td>Valid values: float. Range in [0, 1].</td>
</tr>
<tr>
<td></td>
<td>Default value: 0.9</td>
</tr>
</tbody>
</table>
### Parameter Name | Description
--- | ---
**image_shape** | The input image dimensions, which is the same size as the input layer of the network. The format is defined as 'num_channels, height, width'. The image dimension can take on any value as the network can handle varied dimensions of the input. However, there may be memory constraints if a larger image dimension is used. Pretrained models can use only a fixed 224 x 224 image size. Typical image dimensions for image classification are '3, 224, 224'. This is similar to the ImageNet dataset.

For training, if any input image is smaller than this parameter in any dimension, training fails. If an image is larger, a portion of the image is cropped, with the cropped area specified by this parameter. If hyperparameter augmentation_type is set, random crop is taken; otherwise, central crop is taken.

At inference, input images are resized to the image_shape that was used during training. Aspect ratio is not preserved, and images are not cropped.

**Optional**

Valid values: string

Default value: '3, 224, 224'

**kv_store** | Weight update synchronization mode during distributed training. The weights updates can be updated either synchronously or asynchronously across machines. Synchronous updates typically provide better accuracy than asynchronous updates but can be slower. See distributed training in MXNet for more details.

This parameter is not applicable to single machine training.

- **dist_sync**: The gradients are synchronized after every batch with all the workers. With dist_sync, batch-size now means the batch size used on each machine. So if there are n machines and we use batch size b, then dist_sync behaves like local with batch size n*b

- **dist_async**: Performs asynchronous updates. The weights are updated whenever gradients are received from any machine and the weight updates are atomic. However, the order is not guaranteed.

**Optional**

Valid values: dist_sync or dist_async

Default value: no default value

**learning_rate** | Initial learning rate.

**Optional**

Valid values: float. Range in [0, 1].

Default value: 0.1
<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>lr_scheduler_factor</td>
<td>The ratio to reduce learning rate used in conjunction with the lr_scheduler_step parameter, defined as $lr_{new} = lr_{old} \times lr_{scheduler_factor}$.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: float. Range in $[0, 1]$.</td>
</tr>
<tr>
<td></td>
<td>Default value: 0.1</td>
</tr>
<tr>
<td>lr_scheduler_step</td>
<td>The epochs at which to reduce the learning rate. As explained in the lr_scheduler_factor parameter, the learning rate is reduced by lr_scheduler_factor at these epochs. For example, if the value is set to &quot;10, 20&quot;, then the learning rate is reduced by lr_scheduler_factor after 10th epoch and again by lr_scheduler_factor after 20th epoch. The epochs are delimited by &quot;,&quot;.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: string</td>
</tr>
<tr>
<td></td>
<td>Default value: no default value</td>
</tr>
<tr>
<td>mini_batch_size</td>
<td>The batch size for training. In a single-machine multi-GPU setting, each GPU handles mini_batch_size/num_gpus training samples. For the multi-machine training in dist_sync mode, the actual batch size is mini_batch_size*number of machines. See MXNet docs for more details.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: positive integer</td>
</tr>
<tr>
<td></td>
<td>Default value: 32</td>
</tr>
<tr>
<td>momentum</td>
<td>The momentum for sgd and nag, ignored for other optimizers.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: float. Range in $[0, 1]$.</td>
</tr>
<tr>
<td></td>
<td>Default value: 0.9</td>
</tr>
<tr>
<td>multi_label</td>
<td>Flag to use for multi-label classification where each sample can be assigned multiple labels. Average accuracy across all classes is logged.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: 0 or 1</td>
</tr>
<tr>
<td></td>
<td>Default value: 0</td>
</tr>
<tr>
<td>Parameter Name</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>num_layers</td>
<td>Number of layers for the network. For data with large image size (for example, 224x224 - like ImageNet), we suggest selecting the number of layers from the set [18, 34, 50, 101, 152, 200]. For data with small image size (for example, 28x28 - like CIFAR), we suggest selecting the number of layers from the set [20, 32, 44, 56, 110]. The number of layers in each set is based on the ResNet paper. For transfer learning, the number of layers defines the architecture of base network and hence can only be selected from the set [18, 34, 50, 101, 152, 200].</td>
</tr>
<tr>
<td>optimizer</td>
<td>The optimizer type. For more details of the parameters for the optimizers, please refer to MXNet's API.</td>
</tr>
<tr>
<td>precision_dtype</td>
<td>The precision of the weights used for training. The algorithm can use either single precision (float32) or half precision (float16) for the weights. Using half-precision for weights results in reduced memory consumption.</td>
</tr>
<tr>
<td>resize</td>
<td>The number of pixels in the shortest side of an image after resizing it for training. If the parameter is not set, then the training data is used without resizing. The parameter should be larger than both the width and height components of image_shape to prevent training failure.</td>
</tr>
<tr>
<td>Parameter Name</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>top_k</td>
<td>Reports the top-k accuracy during training. This parameter has to be greater than 1, since the top-1 training accuracy is the same as the regular training accuracy that has already been reported.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: positive integer larger than 1.</td>
</tr>
<tr>
<td></td>
<td>Default value: no default value</td>
</tr>
<tr>
<td>use_pretrained_model</td>
<td>Flag to use pre-trained model for training. If set to 1, then the pretrained model with the corresponding number of layers is loaded and used for training. Only the top FC layer are reinitialized with random weights. Otherwise, the network is trained from scratch.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: 0 or 1</td>
</tr>
<tr>
<td></td>
<td>Default value: 0</td>
</tr>
<tr>
<td>use_weighted_loss</td>
<td>Flag to use weighted cross-entropy loss for multi-label classification (used only when multi_label = 1), where the weights are calculated based on the distribution of classes.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: 0 or 1</td>
</tr>
<tr>
<td></td>
<td>Default value: 0</td>
</tr>
<tr>
<td>weight_decay</td>
<td>The coefficient weight decay for sgd and nag, ignored for other optimizers.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: float. Range in [0, 1].</td>
</tr>
<tr>
<td></td>
<td>Default value: 0.0001</td>
</tr>
</tbody>
</table>

Tune an Image Classification Model

*Automatic model tuning*, also known as hyperparameter tuning, finds the best version of a model by running many jobs that test a range of hyperparameters on your dataset. You choose the tunable hyperparameters, a range of values for each, and an objective metric. You choose the objective metric from the metrics that the algorithm computes. Automatic model tuning searches the hyperparameters chosen to find the combination of values that result in the model that optimizes the objective metric.

For more information about model tuning, see [Perform Automatic Model Tuning](#).

Metrics Computed by the Image Classification Algorithm

The image classification algorithm is a supervised algorithm. It reports an accuracy metric that is computed during training. When tuning the model, choose this metric as the objective metric.
### Tunable Image Classification Hyperparameters

Tune an image classification model with the following hyperparameters. The hyperparameters that have the greatest impact on image classification objective metrics are: `mini_batch_size`, `learning_rate`, and `optimizer`. Tune the optimizer-related hyperparameters, such as `momentum`, `weight_decay`, `beta_1`, `beta_2`, `eps`, and `gamma`, based on the selected optimizer. For example, use `beta_1` and `beta_2` only when `adam` is the optimizer.

For more information about which hyperparameters are used in each optimizer, see Image Classification Hyperparameters (p. 694).

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Parameter Type</th>
<th>Recommended Ranges</th>
</tr>
</thead>
<tbody>
<tr>
<td>beta_1</td>
<td>ContinuousParameterRanges</td>
<td>MinValue: 1e-6, MaxValue: 0.999</td>
</tr>
<tr>
<td>beta_2</td>
<td>ContinuousParameterRanges</td>
<td>MinValue: 1e-6, MaxValue: 0.999</td>
</tr>
<tr>
<td>eps</td>
<td>ContinuousParameterRanges</td>
<td>MinValue: 1e-8, MaxValue: 1.0</td>
</tr>
<tr>
<td>gamma</td>
<td>ContinuousParameterRanges</td>
<td>MinValue: 1e-8, MaxValue: 0.999</td>
</tr>
<tr>
<td>learning_rate</td>
<td>ContinuousParameterRanges</td>
<td>MinValue: 1e-6, MaxValue: 0.5</td>
</tr>
<tr>
<td>mini_batch_size</td>
<td>IntegerParameterRanges</td>
<td>MinValue: 8, MaxValue: 512</td>
</tr>
<tr>
<td>momentum</td>
<td>ContinuousParameterRanges</td>
<td>MinValue: 0.0, MaxValue: 0.999</td>
</tr>
<tr>
<td>optimizer</td>
<td>CategoricalParameterRanges</td>
<td>['sgd', 'adam', 'rmsprop', 'nag']</td>
</tr>
<tr>
<td>weight_decay</td>
<td>ContinuousParameterRanges</td>
<td>MinValue: 0.0, MaxValue: 0.999</td>
</tr>
</tbody>
</table>

### IP Insights

Amazon SageMaker IP Insights is an unsupervised learning algorithm that learns the usage patterns for IPv4 addresses. It is designed to capture associations between IPv4 addresses and various entities, such as user IDs or account numbers. You can use it to identify a user attempting to log into a web service from an anomalous IP address, for example. Or you can use it to identify an account that is attempting to create computing resources from an unusual IP address. Trained IP Insight models can be hosted at an endpoint for making real-time predictions or used for processing batch transforms.

SageMaker IP insights ingests historical data as (entity, IPv4 Address) pairs and learns the IP usage patterns of each entity. When queried with an (entity, IPv4 Address) event, a SageMaker IP Insights
model returns a score that infers how anomalous the pattern of the event is. For example, when a user attempts to log in from an IP address, if the IP Insights score is high enough, a web login server might decide to trigger a multi-factor authentication system. In more advanced solutions, you can feed the IP Insights score into another machine learning model. For example, you can combine the IP Insight score with other features to rank the findings of another security system, such as those from Amazon GuardDuty.

The SageMaker IP Insights algorithm can also learn vector representations of IP addresses, known as embeddings. You can use vector-encoded embeddings as features in downstream machine learning tasks that use the information observed in the IP addresses. For example, you can use them in tasks such as measuring similarities between IP addresses in clustering and visualization tasks.

Topics
- Input/Output Interface for the IP Insights Algorithm (p. 702)
- EC2 Instance Recommendation for the IP Insights Algorithm (p. 702)
- IP Insights Sample Notebooks (p. 703)
- How IP Insights Works (p. 703)
- IP Insights Hyperparameters (p. 704)
- Tune an IP Insights Model (p. 707)
- IP Insights Data Formats (p. 709)

Input/Output Interface for the IP Insights Algorithm

Training and Validation
The SageMaker IP Insights algorithm supports training and validation data channels. It uses the optional validation channel to compute an area-under-curve (AUC) score on a predefined negative sampling strategy. The AUC metric validates how well the model discriminates between positive and negative samples. Training and validation data content types need to be in text/csv format. The first column of the CSV data is an opaque string that provides a unique identifier for the entity. The second column is an IPv4 address in decimal-dot notation. IP Insights currently supports only File mode. For more information and some examples, see IP Insights Training Data Formats (p. 709).

Inference
For inference, IP Insights supports text/csv, application/json, and application/jsonlines data content types. For more information about the common data formats for inference provided by SageMaker, see Common Data Formats for Inference (p. 650). IP Insights inference returns output formatted as either application/json or application/jsonlines. Each record in the output data contains the corresponding dot_product (or compatibility score) for each input data point. For more information and some examples, see IP Insights Inference Data Formats (p. 709).

EC2 Instance Recommendation for the IP Insights Algorithm
The SageMaker IP Insights algorithm can run on both GPU and CPU instances. For training jobs, we recommend using GPU instances. However, for certain workloads with large training datasets, distributed CPU instances might reduce training costs. For inference, we recommend using CPU instances.

GPU Instances for the IP Insights Algorithm
IP Insights supports all available GPUs. If you need to speed up training, we recommend starting with a single GPU instance, such as ml.p3.2xlarge, and then moving to a multi-GPU environment, such as ml.p3.8xlarge and ml.p3.16xlarge. Multi-GPUs automatically divide the mini batches of training data across themselves. If you switch from a single GPU to multiple GPUs, the mini_batch_size is divided equally into the number of GPUs used. You may want to increase the value of the mini_batch_size to compensate for this.
CPU Instances for the IP Insights Algorithm

The type of CPU instance that we recommend depends largely on the instance's available memory and the model size. The model size is determined by two hyperparameters: `vector_dim` and `num_entity_vectors`. The maximum supported model size is 8 GB. The following table lists typical EC2 instance types that you would deploy based on these input parameters for various model sizes. In Table 1, the value for `vector_dim` in the first column range from 32 to 2048 and the values for `num_entity_vectors` in the first row range from 10,000 to 50,000,000.

<table>
<thead>
<tr>
<th>vector_dim</th>
<th>num_entity_vectors</th>
<th>ml.m5.large</th>
<th>ml.m5.large</th>
<th>ml.m5.large</th>
<th>ml.m5.large</th>
<th>ml.m5.large</th>
<th>ml.m5.large</th>
<th>ml.m5.xlarge</th>
<th>ml.m5.2xlarge</th>
<th>ml.m5.4xlarge</th>
<th>ml.m5.4xlarge</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>10,000</td>
<td>ml.m5.large</td>
<td>ml.m5.large</td>
<td>ml.m5.large</td>
<td>ml.m5.large</td>
<td>ml.m5.large</td>
<td>ml.m5.xlarge</td>
<td>ml.m5.2xlarge</td>
<td>ml.m5.4xlarge</td>
<td>ml.m5.4xlarge</td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>50,000</td>
<td>ml.m5.large</td>
<td>ml.m5.large</td>
<td>ml.m5.large</td>
<td>ml.m5.large</td>
<td>ml.m5.large</td>
<td>ml.m5.xlarge</td>
<td>ml.m5.2xlarge</td>
<td>ml.m5.4xlarge</td>
<td></td>
<td></td>
</tr>
<tr>
<td>128</td>
<td>100,000</td>
<td>ml.m5.large</td>
<td>ml.m5.large</td>
<td>ml.m5.large</td>
<td>ml.m5.large</td>
<td>ml.m5.large</td>
<td>ml.m5.xlarge</td>
<td>ml.m5.4xlarge</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>256</td>
<td>500,000</td>
<td>ml.m5.large</td>
<td>ml.m5.large</td>
<td>ml.m5.large</td>
<td>ml.m5.large</td>
<td>ml.m5.large</td>
<td>ml.m5.xlarge</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>512</td>
<td>1,000,000</td>
<td>ml.m5.large</td>
<td>ml.m5.large</td>
<td>ml.m5.large</td>
<td>ml.m5.xlarge</td>
<td>ml.m5.xlarge</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1024</td>
<td>5,000,000</td>
<td>ml.m5.large</td>
<td>ml.m5.large</td>
<td>ml.m5.large</td>
<td>ml.m5.xlarge</td>
<td>ml.m5.xlarge</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2048</td>
<td>10,000,000</td>
<td>ml.m5.large</td>
<td>ml.m5.large</td>
<td>ml.m5.large</td>
<td>ml.m5.xlarge</td>
<td>ml.m5.xlarge</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The values for the `mini_batch_size`, `num_ip_encoder_layers`, `random_negative_sampling_rate`, and `shuffled_negative_sampling_rate` hyperparameters also affect the amount of memory required. If these values are large, you might need to use a larger instance type than normal.

IP Insights Sample Notebooks

For a sample notebook that shows how to train the SageMaker IP Insights algorithm and perform inferences with it, see An Introduction to the SageMaker IP Insights Algorithm. For instructions how to create and access Jupyter notebook instances that you can use to run the example in SageMaker, see Use Amazon SageMaker Notebook Instances (p. 124). After creating a notebook instance, choose the SageMaker Examples tab to see a list of all the SageMaker examples. To open a notebook, choose its Use tab and choose Create copy.

How IP Insights Works

Amazon SageMaker IP Insights is an unsupervised algorithm that consumes observed data in the form of (entity, IPv4 address) pairs that associates entities with IP addresses. IP Insights determines how likely it is that an entity would use a particular IP address by learning latent vector representations for both entities and IP addresses. The distance between these two representations can then serve as the proxy for how likely this association is.

The IP Insights algorithm uses a neural network to learn the latent vector representations for entities and IP addresses. Entities are first hashed to a large but fixed hash space and then encoded by a simple embedding layer. Character strings such as user names or account IDs can be fed directly into IP Insights as they appear in log files. You don't need to preprocess the data for entity identifiers. You can provide entities as an arbitrary string value during both training and inference. The hash size should be configured with a value that is high enough to insure that the number of collisions, which occur when distinct entities are mapped to the same latent vector, remain insignificant. For more information about how to select appropriate hash sizes, see Feature Hashing for Large Scale Multitask Learning. For
representing IP addresses, on the other hand, IP Insights uses a specially designed encoder network to uniquely represent each possible IPv4 address by exploiting the prefix structure of IP addresses.

During training, IP Insights automatically generates negative samples by randomly pairing entities and IP addresses. These negative samples represent data that is less likely to occur in reality. The model is trained to discriminate between positive samples that are observed in the training data and these generated negative samples. More specifically, the model is trained to minimize the cross entropy, also known as the log loss, defined as follows:

\[
L = \frac{1}{N} \sum_{n} [y_n \log p_n + (1 - y_n) \log (1 - p_n)]
\]

\(y_n\) is the label that indicates whether the sample is from the real distribution governing observed data \((y_n=1)\) or from the distribution generating negative samples \((y_n=0)\). \(p_n\) is the probability that the sample is from the real distribution, as predicted by the model.

Generating negative samples is an important process that is used to achieve an accurate model of the observed data. If negative samples are extremely unlikely, for example, if all of the IP addresses in negative samples are 10.0.0.0, then the model trivially learns to distinguish negative samples and fails to accurately characterize the actual observed dataset. To keep negative samples more realistic, IP Insights generates negative samples both by randomly generating IP addresses and randomly picking IP addresses from training data. You can configure the type of negative sampling and the rates at which negative samples are generated with the random_negative_sampling_rate and shuffled_negative_sampling_rate hyperparameters.

Given an nth (entity, IP address pair), the IP Insights model outputs a score, \(S_n\), that indicates how compatible the entity is with the IP address. This score corresponds to the log odds ratio for a given (entity, IP address) of the pair coming from a real distribution as compared to coming from a negative distribution. It is defined as follows:

\[
S_n = \log \left( \frac{P_{\text{real}}(n)}{P_{\text{neg}}(n)} \right)
\]

The score is essentially a measure of the similarity between the vector representations of the nth entity and IP address. It can be interpreted as how much more likely it would be to observe this event in reality than in a randomly generated dataset. During training, the algorithm uses this score to calculate an estimate of the probability of a sample coming from the real distribution, \(p_n\), to use in the cross entropy minimization, where:

\[
p_n = \frac{1}{1 + e^{-S_n}}
\]

**IP Insights Hyperparameters**

In the `CreateTransformJob` request, you specify the training algorithm. You can also specify algorithm-specific hyperparameters as string-to-string maps. The following table lists the hyperparameters for the Amazon SageMaker IP Insights algorithm.

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>num_entity_vectors</td>
<td>The number of entity vector representations (entity embedding vectors) to train. Each entity in the training set is randomly assigned to one of these vectors using a hash function. Because of hash collisions, it might be possible to have multiple entities assigned to the same vector. This would cause the same vector to represent multiple entities. This generally has a negligible effect on model performance, as long as the collision rate is not too severe. To keep the collision rate low, set this value as</td>
</tr>
<tr>
<td>Parameter Name</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td></td>
<td>high as possible. However, the model size, and, therefore, the memory requirement, for both training and inference, scales linearly with this hyperparameter. We recommend that you set this value to twice the number of unique entity identifiers.</td>
</tr>
<tr>
<td><strong>vector_dim</strong></td>
<td>The size of embedding vectors to represent entities and IP addresses. The larger the value, the more information that can be encoded using these representations. In practice, model size scales linearly with this parameter and limits how large the dimension can be. In addition, using vector representations that are too large can cause the model to overfit, especially for small training datasets. Overfitting occurs when a model doesn't learn any pattern in the data but effectively memorizes the training data and, therefore, cannot generalize well and performs poorly during inference. The recommended value is 128.</td>
</tr>
<tr>
<td></td>
<td><strong>Required</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: 4 ≤ positive integer ≤ 4096</td>
</tr>
<tr>
<td><strong>batch_metrics_publish_interval</strong></td>
<td>The interval (every X batches) at which the Apache MXNet Speedometer function prints the training speed of the network (samples/second).</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: positive integer ≥ 1</td>
</tr>
<tr>
<td></td>
<td>Default value: 1,000</td>
</tr>
<tr>
<td><strong>epochs</strong></td>
<td>The number of passes over the training data. The optimal value depends on your data size and learning rate. Typical values range from 5 to 100.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: positive integer ≥ 1</td>
</tr>
<tr>
<td></td>
<td>Default value: 10</td>
</tr>
<tr>
<td>Parameter Name</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>learning_rate</td>
<td>The learning rate for the optimizer. IP Insights use a gradient-descent-based Adam optimizer. The learning rate effectively controls the step size to update model parameters at each iteration. Too large a learning rate can cause the model to diverge because the training is likely to overshoot a minima. On the other hand, too small a learning rate slows down convergence. Typical values range from 1e-4 to 1e-1.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: 1e-6 ≤ float ≤ 10.0</td>
</tr>
<tr>
<td></td>
<td>Default value: 0.001</td>
</tr>
<tr>
<td>mini_batch_size</td>
<td>The number of examples in each mini batch. The training procedure processes data in mini batches. The optimal value depends on the number of unique account identifiers in the dataset. In general, the larger the mini_batch_size, the faster the training and the greater the number of possible shuffled-negative-sample combinations. However, with a large mini_batch_size, the training is more likely to converge to a poor local minimum and perform relatively worse for inference.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: 1 ≤ positive integer ≤ 500000</td>
</tr>
<tr>
<td></td>
<td>Default value: 10,000</td>
</tr>
<tr>
<td>num_ip_encoder_layers</td>
<td>The number of fully connected layers used to encode the IP address embedding. The larger the number of layers, the greater the model's capacity to capture patterns among IP addresses. However, using a large number of layers increases the chance of overfitting.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: 0 ≤ positive integer ≤ 100</td>
</tr>
<tr>
<td></td>
<td>Default value: 1</td>
</tr>
<tr>
<td>Parameter Name</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>random_negative_sampling_rate</td>
<td>The number of random negative samples, R, to generate per input example. The training procedure relies on negative samples to prevent the vector representations of the model collapsing to a single point. Random negative sampling generates R random IP addresses for each input account in the mini batch. The sum of the random_negative_sampling_rate (R) and shuffled_negative_sampling_rate (S) must be in the interval: $1 \leq R + S \leq 500$.</td>
</tr>
<tr>
<td>Optional</td>
<td>Valid values: $0 \leq$ positive integer $\leq 500$</td>
</tr>
<tr>
<td>Default value: 1</td>
<td></td>
</tr>
<tr>
<td>shuffled_negative_sampling_rate</td>
<td>The number of shuffled negative samples, S, to generate per input example. In some cases, it helps to use more realistic negative samples that are randomly picked from the training data itself. This kind of negative sampling is achieved by shuffling the data within a mini batch. Shuffled negative sampling generates S negative IP addresses by shuffling the IP address and account pairings within a mini batch. The sum of the random_negative_sampling_rate (R) and shuffled_negative_sampling_rate (S) must be in the interval: $1 \leq R + S \leq 500$.</td>
</tr>
<tr>
<td>Optional</td>
<td>Valid values: $0 \leq$ positive integer $\leq 500$</td>
</tr>
<tr>
<td>Default value: 1</td>
<td></td>
</tr>
<tr>
<td>weight_decay</td>
<td>The weight decay coefficient. This parameter adds an L2 regularization factor that is required to prevent the model from overfitting the training data.</td>
</tr>
<tr>
<td>Optional</td>
<td>Valid values: $0.0 \leq$ float $\leq 10.0$</td>
</tr>
<tr>
<td>Default value: 0.00001</td>
<td></td>
</tr>
</tbody>
</table>

**Tune an IP Insights Model**

*Automatic model tuning*, also called hyperparameter tuning, finds the best version of a model by running many jobs that test a range of hyperparameters on your dataset. You choose the tunable hyperparameters, a range of values for each, and an objective metric. You choose the objective metric from the metrics that the algorithm computes. Automatic model tuning searches the hyperparameters chosen to find the combination of values that result in the model that optimizes the objective metric.

For more information about model tuning, see [Perform Automatic Model Tuning](p. 1023).
Metrics Computed by the IP Insights Algorithm

The Amazon SageMaker IP Insights algorithm is an unsupervised learning algorithm that learns associations between IP addresses and entities. The algorithm trains a discriminator model, which learns to separate observed data points (positive samples) from randomly generated data points (negative samples). Automatic model tuning on IP Insights helps you find the model that can most accurately distinguish between unlabeled validation data and automatically generated negative samples. The model accuracy on the validation dataset is measured by the area under the receiver operating characteristic curve. This validation:discriminator_auc metric can take values between 0.0 and 1.0, where 1.0 indicates perfect accuracy.

The IP Insights algorithm computes a validation:discriminator_auc metric during validation, the value of which is used as the objective function to optimize for hyperparameter tuning.

<table>
<thead>
<tr>
<th>Metric Name</th>
<th>Description</th>
<th>Optimization Direction</th>
</tr>
</thead>
<tbody>
<tr>
<td>validation:discriminator_auc</td>
<td>Area under the receiver operating characteristic curve on the validation dataset. The validation dataset is not labeled. Area Under the Curve (AUC) is a metric that describes the model's ability to discriminate validation data points from randomly generated data points.</td>
<td>Maximize</td>
</tr>
</tbody>
</table>

Tunable IP Insights Hyperparameters

You can tune the following hyperparameters for the SageMaker IP Insights algorithm.

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Parameter Type</th>
<th>Recommended Ranges</th>
</tr>
</thead>
<tbody>
<tr>
<td>epochs</td>
<td>IntegerParameterRange</td>
<td>MinValue: 1, MaxValue: 100</td>
</tr>
<tr>
<td>learning_rate</td>
<td>ContinuousParameterRange</td>
<td>MinValue: 1e-4, MaxValue: 0.1</td>
</tr>
<tr>
<td>mini_batch_size</td>
<td>IntegerParameterRanges</td>
<td>MinValue: 100, MaxValue: 50000</td>
</tr>
<tr>
<td>num_entity_vectors</td>
<td>IntegerParameterRanges</td>
<td>MinValue: 10000, MaxValue: 1000000</td>
</tr>
<tr>
<td>num_ip_encoder_layers</td>
<td>IntegerParameterRanges</td>
<td>MinValue: 1, MaxValue: 10</td>
</tr>
<tr>
<td>random_negative_sampling</td>
<td>IntegerParameterRanges</td>
<td>MinValue: 0, MaxValue: 10</td>
</tr>
<tr>
<td>shuffled_negative_sampling</td>
<td>IntegerParameterRanges</td>
<td>MinValue: 0, MaxValue: 10</td>
</tr>
<tr>
<td>vector_dim</td>
<td>IntegerParameterRanges</td>
<td>MinValue: 8, MaxValue: 256</td>
</tr>
<tr>
<td>weight_decay</td>
<td>ContinuousParameterRange</td>
<td>MinValue: 0.0, MaxValue: 1.0</td>
</tr>
</tbody>
</table>
IP Insights Data Formats

This section provides examples of the available input and output data formats used by the IP Insights algorithm during training and inference.

Topics
- IP Insights Training Data Formats (p. 709)
- IP Insights Inference Data Formats (p. 709)

IP Insights Training Data Formats

The following are the available data input formats for the IP Insights algorithm. Amazon SageMaker built-in algorithms adhere to the common input training format described in Common Data Formats for Training (p. 646). However, the SageMaker IP Insights algorithm currently supports only the CSV data input format.

IP Insights Training Data Input Formats

INPUT: CSV

The CSV file must have two columns. The first column is an opaque string that corresponds to an entity's unique identifier. The second column is the IPv4 address of the entity's access event in decimal-dot notation.

```text/csv
entity_id_1, 192.168.1.2
entity_id_2, 10.10.1.2
```

IP Insights Inference Data Formats

The following are the available input and output formats for the IP Insights algorithm. Amazon SageMaker built-in algorithms adhere to the common input inference format described in Common Data Formats for Inference (p. 650). However, the SageMaker IP Insights algorithm does not currently support RecordIO format.

IP Insights Input Request Formats

INPUT: CSV Format

The CSV file must have two columns. The first column is an opaque string that corresponds to an entity's unique identifier. The second column is the IPv4 address of the entity's access event in decimal-dot notation.

```text/csv
entity_id_1, 192.168.1.2
entity_id_2, 10.10.1.2
```

INPUT: JSON Format

JSON data can be provided in different formats. IP Insights follows the common SageMaker formats. For more information about inference formats, see Common Data Formats for Inference (p. 650).

```application/json
{
    "instances": [
```
INPUT: JSONLINES Format

The JSON Lines content type is useful for running batch transform jobs. For more information on SageMaker inference formats, see Common Data Formats for Inference (p. 650). For more information on running batch transform jobs, see Get Inferences for an Entire Dataset with Batch Transform (p. 14).

```
{"data": "{features": {"values": ["entity_id_1", "192.168.1.2"]},
 {"features": ["entity_id_2", "10.10.1.2"]}
 ]}
```

IP Insights Output Response Formats

OUTPUT: JSON Response Format

The default output of the SageMaker IP Insights algorithm is the dot_product between the input entity and IP address. The dot_product signifies how compatible the model considers the entity and IP address. The dot_product is unbounded. To make predictions about whether an event is anomalous, you need to set a threshold based on your defined distribution. For information about how to use the dot_product for anomaly detection, see the An Introduction to the SageMakerIP Insights Algorithm.

```
accept: application/json

{
 "predictions": [
 {"dot_product": 0.0},
 {"dot_product": 2.0}
 ]
}
```

Advanced users can access the model's learned entity and IP embeddings by providing the additional content-type parameter verbose=True to the Accept heading. You can use the entity_embedding and ip_embedding for debugging, visualizing, and understanding the model. Additionally, you can use these embeddings in other machine learning techniques, such as classification or clustering.

```
accept: application/json;verbose=True

{
 "predictions": [
 {"dot_product": 0.0,
 "entity_embedding": [1.0, 0.0, 0.0],
 "ip_embedding": [0.0, 1.0, 0.0]
 },
 {"dot_product": 2.0,
 "entity_embedding": [1.0, 0.0, 1.0],
 "ip_embedding": [1.0, 0.0, 1.0]
 }
 ]
}
```

OUTPUT: JSONLINES Response Format

```
accept: application/jsonlines

710
```
K-Means Algorithm

K-means is an unsupervised learning algorithm. It attempts to find discrete groupings within data, where members of a group are as similar as possible to one another and as different as possible from members of other groups. You define the attributes that you want the algorithm to use to determine similarity.

Amazon SageMaker uses a modified version of the web-scale k-means clustering algorithm. Compared with the original version of the algorithm, the version used by Amazon SageMaker is more accurate. Like the original algorithm, it scales to massive datasets and delivers improvements in training time. To do this, the version used by Amazon SageMaker streams mini-batches (small, random subsets) of the training data. For more information about mini-batch k-means, see Web-scale k-means Clustering.

The k-means algorithm expects tabular data, where rows represent the observations that you want to cluster, and the columns represent attributes of the observations. The \( n \) attributes in each row represent a point in \( n \)-dimensional space. The Euclidean distance between these points represents the similarity of the corresponding observations. The algorithm groups observations with similar attribute values (the points corresponding to these observations are closer together). For more information about how k-means works in Amazon SageMaker, see How K-Means Clustering Works (p. 712).

Topics
- Input/Output Interface for the K-Means Algorithm (p. 711)
- EC2 Instance Recommendation for the K-Means Algorithm (p. 712)
- K-Means Sample Notebooks (p. 712)
- How K-Means Clustering Works (p. 712)
- K-Means Hyperparameters (p. 715)
- Tune a K-Means Model (p. 717)
- K-Means Response Formats (p. 718)

Input/Output Interface for the K-Means Algorithm

For training, the k-means algorithm expects data to be provided in the train channel (recommended S3DataDistributionType=ShardedByS3Key), with an optional test channel (recommended S3DataDistributionType=FullyReplicated) to score the data on. Both recordIO-wrapped-protobuf and CSV formats are supported for training. You can use either File mode or Pipe mode to train models on data that is formatted as recordIO-wrapped-protobuf or as CSV.

For inference, text/csv, application/json, and application/x-recordio-protobuf are supported. k-means returns a closest_cluster label and the distance_to_cluster for each observation.

For more information on input and output file formats, see K-Means Response Formats (p. 718) for inference and the K-Means Sample Notebooks (p. 712). The k-means algorithm does not support multiple instance learning, in which the training set consists of labeled “bags”, each of which is a collection of unlabeled instances.
EC2 Instance Recommendation for the K-Means Algorithm

We recommend training k-means on CPU instances. You can train on GPU instances, but should limit GPU training to p*.xlarge instances because only one GPU per instance is used.

K-Means Sample Notebooks

For a sample notebook that uses the SageMaker K-means algorithm to segment the population of counties in the United States by attributes identified using principle component analysis, see Analyze US census data for population segmentation using Amazon SageMaker. For instructions how to create and access Jupyter notebook instances that you can use to run the example in SageMaker, see Use Amazon SageMaker Notebook Instances (p. 124). Once you have created a notebook instance and opened it, select the SageMaker Examples tab to see a list of all the SageMaker samples. To open a notebook, click on its Use tab and select Create copy.

How K-Means Clustering Works

K-means is an algorithm that trains a model that groups similar objects together. The k-means algorithm accomplishes this by mapping each observation in the input dataset to a point in the n-dimensional space (where n is the number of attributes of the observation). For example, your dataset might contain observations of temperature and humidity in a particular location, which are mapped to points (t, h) in 2-dimensional space.

Note
Clustering algorithms are unsupervised. In unsupervised learning, labels that might be associated with the objects in the training dataset aren’t used.

In k-means clustering, each cluster has a center. During model training, the k-means algorithm uses the distance of the point that corresponds to each observation in the dataset to the cluster centers as the basis for clustering. You choose the number of clusters (k) to create.

For example, suppose that you want to create a model to recognize handwritten digits and you choose the MNIST dataset for training. The dataset provides thousands of images of handwritten digits (0 through 9). In this example, you might choose to create 10 clusters, one for each digit (0, 1, ..., 9). As part of model training, the k-means algorithm groups the input images into 10 clusters.

Each image in the MNIST dataset is a 28x28-pixel image, with a total of 784 pixels. Each image corresponds to a point in a 784-dimensional space, similar to a point in a 2-dimensional space (x,y). To find a cluster to which a point belongs, the k-means algorithm finds the distance of that point from all of the cluster centers. It then chooses the cluster with the closest center as the cluster to which the image belongs.

Note
Amazon SageMaker uses a customized version of the algorithm where, instead of specifying that the algorithm create k clusters, you might choose to improve model accuracy by specifying extra cluster centers (K = k*x). However, the algorithm ultimately reduces these to k clusters.

In SageMaker, you specify the number of clusters when creating a training job. For more information, see CreateTrainingJob. In the request body, you add the HyperParameters string map to specify the k and extra_center_factor strings.

The following is a summary of how k-means works for model training in SageMaker:

1. It determines the initial K cluster centers.

   Note
   In the following topics, K clusters refer to k * x, where you specify k and x when creating a model training job.

2. It iterates over input training data and recalculates cluster centers.
3. It reduces resulting clusters to \(k\) (if the data scientist specified the creation of \(k\times x\) clusters in the request).

The following sections also explain some of the parameters that a data scientist might specify to configure a model training job as part of the `HyperParameters` string map.

**Topics**
- Step 1: Determine the Initial Cluster Centers (p. 713)
- Step 2: Iterate over the Training Dataset and Calculate Cluster Centers (p. 714)
- Step 3: Reduce the Clusters from \(K\) to \(k\) (p. 714)

**Step 1: Determine the Initial Cluster Centers**

When using k-means in SageMaker, the initial cluster centers are chosen from the observations in a small, randomly sampled batch. Choose one of the following strategies to determine how these initial cluster centers are selected:

- The random approach—Randomly choose \(K\) observations in your input dataset as cluster centers. For example, you might choose a cluster center that points to the 784-dimensional space that corresponds to any 10 images in the MNIST training dataset.

- The k-means++ approach, which works as follows:
  1. Start with one cluster and determine its center. You randomly select an observation from your training dataset and use the point corresponding to the observation as the cluster center. For example, in the MNIST dataset, randomly choose a handwritten digit image. Then choose the point in the 784-dimensional space that corresponds to the image as your cluster center. This is cluster center 1.
  2. Determine the center for cluster 2. From the remaining observations in the training dataset, pick an observation at random. Choose one that is different than the one you previously selected. This observation corresponds to a point that is far away from cluster center 1. Using the MNIST dataset as an example, you do the following:
    - For each of the remaining images, find the distance of the corresponding point from cluster center 1. Square the distance and assign a probability that is proportional to the square of the distance. That way, an image that is different from the one that you previously selected has a higher probability of getting selected as cluster center 2.
    - Choose one of the images randomly, based on probabilities assigned in the previous step. The point that corresponds to the image is cluster center 2.
  3. Repeat Step 2 to find cluster center 3. This time, find the distances of the remaining images from cluster center 2.
  4. Repeat the process until you have the \(K\) cluster centers.

To train a model in SageMaker, you create a training job. In the request, you provide configuration information by specifying the following `HyperParameters` string maps:

- To specify the number of clusters to create, add the `k` string.
- For greater accuracy, add the optional `extra_center_factor` string.
- To specify the strategy that you want to use to determine the initial cluster centers, add the `init_method` string and set its value to `random` or `k-means++`.

For more information, see [CreateTrainingJob](https://docs.aws.amazon.com/sagemaker/latest/dg/API_CreateTrainingJob.html). For an example, see [Create and Run a Training Job (AWS SDK for Python (Boto3))]((https://boto3.amazonaws.com/v1/documentation/api/latest/guide/creating-training-job.html)).
You now have an initial set of cluster centers.

Step 2: Iterate over the Training Dataset and Calculate Cluster Centers

The cluster centers that you created in the preceding step are mostly random, with some consideration for the training dataset. In this step, you use the training dataset to move these centers toward the true cluster centers. The algorithm iterates over the training dataset, and recalculates the \( K \) cluster centers.

1. Read a mini-batch of observations (a small, randomly chosen subset of all records) from the training dataset and do the following.

   **Note**
   When creating a model training job, you specify the batch size in the mini_batch_size string in the HyperParameters string map.

   a. Assign all of the observations in the mini-batch to one of the clusters with the closest cluster center.

   b. Calculate the number of observations assigned to each cluster. Then, calculate the proportion of new points assigned per cluster.

   For example, consider the following clusters:

   Cluster \( c_1 \) = 100 previously assigned points. You added 25 points from the mini-batch in this step.

   Cluster \( c_2 \) = 150 previously assigned points. You added 40 points from the mini-batch in this step.

   Cluster \( c_3 \) = 450 previously assigned points. You added 5 points from the mini-batch in this step.

   Calculate the proportion of new points assigned to each of clusters as follows:

   \[
   \begin{align*}
   p_1 &= \text{proportion of points assigned to } \ c_1 = \frac{25}{(100+25)} \\
   p_2 &= \text{proportion of points assigned to } \ c_2 = \frac{40}{(150+40)} \\
   p_3 &= \text{proportion of points assigned to } \ c_3 = \frac{5}{(450+5)}
   \end{align*}
   \]

   c. Compute the center of the new points added to each cluster:

   \[
   \begin{align*}
   d_1 &= \text{center of the new points added to } \ c_1 \\
   d_2 &= \text{center of the new points added to } \ c_2 \\
   d_3 &= \text{center of the new points added to } \ c_3
   \end{align*}
   \]

   d. Compute the weighted average to find the updated cluster centers as follows:

   \[
   \begin{align*}
   \text{Center of cluster } 1 &= ((1 - p_1) \times \text{center of cluster } 1) + (p_1 \times d_1) \\
   \text{Center of cluster } 2 &= ((1 - p_2) \times \text{center of cluster } 2) + (p_2 \times d_2) \\
   \text{Center of cluster } 3 &= ((1 - p_3) \times \text{center of cluster } 3) + (p_3 \times d_3)
   \end{align*}
   \]

2. Read the next mini-batch, and repeat Step 1 to recalculate the cluster centers.

3. For more information about mini-batch \( k \)-means, see Web-Scale \( k \)-means Clustering).

Step 3: Reduce the Clusters from \( K \) to \( k \)

If the algorithm created \( K \) clusters—\( (K = k \times x) \) where \( x \) is greater than 1—then it reduces the \( K \) clusters to \( k \) clusters. (For more information, see extra_center_factor in the preceding discussion.) It does this by applying Lloyd's method with \( k \)-means++ initialization to the \( K \) cluster centers. For more information about Lloyd's method, see \( k \)-means clustering.
## K-Means Hyperparameters

In the `CreateTrainingJob` request, you specify the training algorithm that you want to use. You can also specify algorithm-specific hyperparameters as string-to-string maps. The following table lists the hyperparameters for the k-means training algorithm provided by Amazon SageMaker. For more information about how k-means clustering works, see How K-Means Clustering Works (p. 712).

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>feature_dim</code></td>
<td>The number of features in the input data.</td>
</tr>
<tr>
<td></td>
<td><strong>Required</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: Positive integer</td>
</tr>
<tr>
<td><code>k</code></td>
<td>The number of required clusters.</td>
</tr>
<tr>
<td></td>
<td><strong>Required</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: Positive integer</td>
</tr>
<tr>
<td><code>epochs</code></td>
<td>The number of passes done over the training data.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: Positive integer</td>
</tr>
<tr>
<td></td>
<td>Default value: 1</td>
</tr>
<tr>
<td><code>eval_metrics</code></td>
<td>A JSON list of metric types used to report a score for the model.</td>
</tr>
<tr>
<td></td>
<td>Allowed values are <code>msd</code> for Means Square Error and <code>ssd</code> for Sum of Square Distance. If test data is provided, the score is reported for each of the metrics requested.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: Either <code>[&quot;msd&quot;]</code> or <code>[&quot;ssd&quot;]</code> or <code>[&quot;msd&quot;, &quot;ssd&quot;]</code></td>
</tr>
<tr>
<td></td>
<td>Default value: <code>[&quot;msd&quot;]</code></td>
</tr>
<tr>
<td><code>extra_center_factor</code></td>
<td>The algorithm creates K centers = num_clusters * extra_center_factor as it runs and reduces the number of centers from K to k when finalizing the model.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: Either a positive integer or <code>auto</code>.</td>
</tr>
<tr>
<td></td>
<td>Default value: <code>auto</code></td>
</tr>
<tr>
<td><code>half_life_time_size</code></td>
<td>Used to determine the weight given to an observation when computing a cluster mean. This weight decays exponentially as more points are observed. When a point is first observed, it is assigned a weight of 1 when computing the cluster mean. The decay constant for the exponential decay function is chosen so that after observing <code>half_life_time_size</code> points, its weight is 1/2. If set to 0, there is no decay.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td>Parameter Name</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>init_method</td>
<td>Method by which the algorithm chooses the initial cluster centers. The standard k-means approach chooses them at random. An alternative k-means++ method chooses the first cluster center at random. Then it spreads out the position of the remaining initial clusters by weighting the selection of centers with a probability distribution that is proportional to the square of the distance of the remaining data points from existing centers.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: Either <code>random</code> or <code>kmeans++</code>.</td>
</tr>
<tr>
<td></td>
<td>Default value: <code>random</code></td>
</tr>
<tr>
<td>local_lloyd_init_method</td>
<td>The initialization method for Lloyd's expectation-maximization (EM) procedure used to build the final model containing ( k ) centers.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: Either <code>random</code> or <code>kmeans++</code>.</td>
</tr>
<tr>
<td></td>
<td>Default value: <code>kmeans++</code></td>
</tr>
<tr>
<td>local_lloyd_max_iter</td>
<td>The maximum number of iterations for Lloyd's expectation-maximization (EM) procedure used to build the final model containing ( k ) centers.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: Positive integer</td>
</tr>
<tr>
<td></td>
<td>Default value: 300</td>
</tr>
<tr>
<td>local_lloyd_num_trials</td>
<td>The number of times the Lloyd's expectation-maximization (EM) procedure with the least loss is run when building the final model containing ( k ) centers.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: Either a positive integer or <code>auto</code>.</td>
</tr>
<tr>
<td></td>
<td>Default value: <code>auto</code></td>
</tr>
<tr>
<td>local_lloyd_tol</td>
<td>The tolerance for change in loss for early stopping of Lloyd's expectation-maximization (EM) procedure used to build the final model containing ( k ) centers.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: Float. Range in ([0, 1]).</td>
</tr>
<tr>
<td></td>
<td>Default value: 0.0001</td>
</tr>
</tbody>
</table>
Use built-in algorithms

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mini_batch_size</td>
<td>The number of observations per mini-batch for the data iterator.</td>
</tr>
<tr>
<td></td>
<td>Optional</td>
</tr>
<tr>
<td></td>
<td>Valid values: Positive integer</td>
</tr>
<tr>
<td></td>
<td>Default value: 5000</td>
</tr>
</tbody>
</table>

**Tune a K-Means Model**

*Automatic model tuning*, also known as hyperparameter tuning, finds the best version of a model by running many jobs that test a range of hyperparameters on your dataset. You choose the tunable hyperparameters, a range of values for each, and an objective metric. You choose the objective metric from the metrics that the algorithm computes. Automatic model tuning searches the hyperparameters chosen to find the combination of values that result in the model that optimizes the objective metric.

The Amazon SageMaker k-means algorithm is an unsupervised algorithm that groups data into clusters whose members are as similar as possible. Because it is unsupervised, it doesn’t use a validation dataset that hyperparameters can optimize against. But it does take a test dataset and emits metrics that depend on the squared distance between the data points and the final cluster centroids at the end of each training run. To find the model that reports the tightest clusters on the test dataset, you can use a hyperparameter tuning job. The clusters optimize the similarity of their members.

For more information about model tuning, see *Perform Automatic Model Tuning (p. 1023)*.

**Metrics Computed by the K-Means Algorithm**

The k-means algorithm computes the following metrics during training. When tuning a model, choose one of these metrics as the objective metric.

<table>
<thead>
<tr>
<th>Metric Name</th>
<th>Description</th>
<th>Optimization Direction</th>
</tr>
</thead>
<tbody>
<tr>
<td>test:msd</td>
<td>Mean squared distances between each record in the test set and the closest center of the model.</td>
<td>Minimize</td>
</tr>
<tr>
<td>test:ssd</td>
<td>Sum of the squared distances between each record in the test set and the closest center of the model.</td>
<td>Minimize</td>
</tr>
</tbody>
</table>

**Tunable K-Means Hyperparameters**

Tune the Amazon SageMaker k-means model with the following hyperparameters. The hyperparameters that have the greatest impact on k-means objective metrics are: mini_batch_size, extra_center_factor, and init_method. Tuning the hyperparameter epochs generally results in minor improvements.

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Parameter Type</th>
<th>Recommended Ranges</th>
</tr>
</thead>
<tbody>
<tr>
<td>epochs</td>
<td>IntegerParameterRanges</td>
<td>MinValue: 1, MaxValue: 10</td>
</tr>
<tr>
<td>extra_center_factor</td>
<td>IntegerParameterRanges</td>
<td>MinValue: 4, MaxValue: 10</td>
</tr>
</tbody>
</table>
K-Means Response Formats

All SageMaker built-in algorithms adhere to the common input inference format described in Common Data Formats - Inference. This topic contains a list of the available output formats for the SageMaker k-means algorithm.

JSON Response Format

```json
{
    "predictions": [
        {
            "closest_cluster": 1.0,
            "distance_to_cluster": 3.0,
        },
        {
            "closest_cluster": 2.0,
            "distance_to_cluster": 5.0,
        },
        ...
    ]
}
```

JSONLINES Response Format

```
{"closest_cluster": 1.0, "distance_to_cluster": 3.0}
{"closest_cluster": 2.0, "distance_to_cluster": 5.0}
```

RECORDIO Response Format

```
[
  Record = {
    features = {},
    label = {
      'closest_cluster': {
        keys: [],
        values: [1.0, 2.0]  # float32
      },
      'distance_to_cluster': {
        keys: [],
        values: [3.0, 5.0]  # float32
      },
    }
  }
]
```

CSV Response Format

The first value in each line corresponds to closest_cluster.

The second value in each line corresponds to distance_to_cluster.
K-Nearest Neighbors (k-NN) Algorithm

Amazon SageMaker k-nearest neighbors (k-NN) algorithm is an index-based algorithm. It uses a non-parametric method for classification or regression. For classification problems, the algorithm queries the $k$ points that are closest to the sample point and returns the most frequently used label of their class as the predicted label. For regression problems, the algorithm queries the $k$ closest points to the sample point and returns the average of their feature values as the predicted value.

Training with the k-NN algorithm has three steps: sampling, dimension reduction, and index building. Sampling reduces the size of the initial dataset so that it fits into memory. For dimension reduction, the algorithm decreases the feature dimension of the data to reduce the footprint of the k-NN model in memory and inference latency. We provide two methods of dimension reduction methods: random projection and the fast Johnson-Lindenstrauss transform. Typically, you use dimension reduction for high-dimensional ($d > 1000$) datasets to avoid the “curse of dimensionality” that troubles the statistical analysis of data that becomes sparse as dimensionality increases. The main objective of k-NN's training is to construct the index. The index enables efficient lookups of distances between points whose values or class labels have not yet been determined and the $k$ nearest points to use for inference.

Topics

- Input/Output Interface for the k-NN Algorithm (p. 719)
- k-NN Sample Notebooks (p. 720)
- How the k-NN Algorithm Works (p. 720)
- EC2 Instance Recommendation for the k-NN Algorithm (p. 721)
- k-NN Hyperparameters (p. 721)
- Tune a k-NN Model (p. 723)
- Data Formats for k-NN Training Input (p. 724)
- k-NN Request and Response Formats (p. 725)

Input/Output Interface for the k-NN Algorithm

SageMaker k-NN supports train and test data channels.

- Use a train channel for data that you want to sample and construct into the k-NN index.
- Use a test channel to emit scores in log files. Scores are listed as one line per mini-batch: accuracy for classifier, mean-squared error (mse) for regressor for score.

For training inputs, k-NN supports text/csv and application/x-recordio-protobuf data formats. For input type text/csv, the first label_size columns are interpreted as the label vector for that row. You can use either File mode or Pipe mode to train models on data that is formatted as recordIO-wrapped-protobuf or as CSV.

For inference inputs, k-NN supports the application/json, application/x-recordio-protobuf, and text/csv data formats. The text/csv format accepts a label_size and encoding parameter. It assumes a label_size of 0 and a UTF-8 encoding.

For inference outputs, k-NN supports the application/json and application/x-recordio-protobuf data formats. These two data formats also support a verbose output mode. In verbose output mode, the API provides the search results with the distances vector sorted from smallest to largest, and corresponding elements in the labels vector.
For batch transform, k-NN supports the application/jsonlines data format for both input and output. An example input is as follows:

```
content-type: application/jsonlines

{"features": [1.5, 16.0, 14.0, 23.0]}
{"data": {"features": {"values": [1.5, 16.0, 14.0, 23.0]}}}
```

An example output is as follows:

```
accept: application/jsonlines

{"predicted_label": 0.0}
{"predicted_label": 2.0}
```

For more information on input and output file formats, see Data Formats for k-NN Training Input (p. 724) for training, k-NN Request and Response Formats (p. 725) for inference, and the k-NN Sample Notebooks (p. 720).

**k-NN Sample Notebooks**

For a sample notebook that uses the SageMaker k-nearest neighbor algorithm to predict wilderness cover types from geological and forest service data, see the [K-Nearest Neighbor Covertype](#).

Use a Jupyter notebook instance to run the example in SageMaker. To learn how to create and open a Jupyter notebook instance in SageMaker, see Use Amazon SageMaker Notebook Instances (p. 124). Once you have created a notebook instance and opened it, select the SageMaker Examples tab to see a list of all the SageMaker example notebooks. Find K-Nearest Neighbor notebooks in the Introduction to Amazon algorithms section. To open a notebook, click on its Use tab and select Create copy.

**How the k-NN Algorithm Works**

**Step 1: Sample**

To specify the total number of data points to be sampled from the training dataset, use the sample_size parameter. For example, if the initial dataset has 1,000 data points and the sample_size is set to 100, where the total number of instances is 2, each worker would sample 50 points. A total set of 100 data points would be collected. Sampling runs in linear time with respect to the number of data points.

**Step 2: Perform Dimension Reduction**

The current implementation of the k-NN algorithm has two methods of dimension reduction. You specify the method in the dimension_reduction_type hyperparameter. The sign method specifies a random projection, which uses a linear projection using a matrix of random signs, and the fjlt method specifies a fast Johnson-Lindenstrauss transform, a method based on the Fourier transform. Both methods preserve the L2 and inner product distances. The fjlt method should be used when the target dimension is large and has better performance with CPU inference. The methods differ in their computational complexity. The sign method requires O(ndk) time to reduce the dimension of a batch of n points of dimension d into a target dimension k. The fjlt method requires O(nd log(d)) time, but the constants involved are larger. Using dimension reduction introduces noise into the data and this noise can reduce prediction accuracy.

**Step 3: Build an Index**

During inference, the algorithm queries the index for the k-nearest-neighbors of a sample point. Based on the references to the points, the algorithm makes the classification or regression prediction. It makes the prediction based on the class labels or values provided. k-NN provides three different types of indexes: a flat index, an inverted index, and an inverted index with product quantization. You specify the type with the index_type parameter.
Serialize the Model

When the k-NN algorithm finishes training, it serializes three files to prepare for inference.

- model_algo-1: Contains the serialized index for computing the nearest neighbors.
- model_algo-1.labels: Contains serialized labels (np.float32 binary format) for computing the predicted label based on the query result from the index.
- model_algo-1.json: Contains the JSON-formatted model metadata which stores the \( k \) and predictor_type hyper-parameters from training for inference along with other relevant state.

With the current implementation of k-NN, you can modify the metadata file to change the way predictions are computed. For example, you can change \( k \) to 10 or change predictor_type to regressor.

```
{
   "k": 5,
   "predictor_type": "classifier",
   "dimension_reduction": {
      "type": "sign", "seed": 3, "target_dim": 10, "input_dim": 20,
      "normalize": False,
      "version": "1.0"
   }
}
```

EC2 Instance Recommendation for the k-NN Algorithm

Instance Recommendation for Training with the k-NN Algorithm

To start, try running training on a CPU, using, for example, an ml.m5.2xlarge instance, or on a GPU using, for example, an ml.p2.xlarge instance.

Instance Recommendation for Inference with the k-NN Algorithm

Inference requests from CPUs generally have a lower average latency than requests from GPUs because there is a tax on CPU-to-GPU communication when you use GPU hardware. However, GPUs generally have higher throughput for larger batches.

**k-NN Hyperparameters**

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>feature_dim</td>
<td>The number of features in the input data. Required</td>
</tr>
<tr>
<td></td>
<td>Valid values: positive integer.</td>
</tr>
<tr>
<td>k</td>
<td>The number of nearest neighbors. Required</td>
</tr>
<tr>
<td></td>
<td>Valid values: positive integer</td>
</tr>
<tr>
<td>predictor_type</td>
<td>The type of inference to use on the data labels. Required</td>
</tr>
<tr>
<td></td>
<td>Valid values: classifier for classification or regressor for regression.</td>
</tr>
<tr>
<td>sample_size</td>
<td>The number of data points to be sampled from the training data set.</td>
</tr>
<tr>
<td>Parameter Name</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>dimension_reduction_target</td>
<td>The target dimension to reduce to.</td>
</tr>
<tr>
<td></td>
<td><strong>Required</strong> when you specify the <code>dimension_reduction_type</code> parameter.</td>
</tr>
<tr>
<td></td>
<td>Valid values: positive integer greater than 0 and less than <code>feature_dim</code>.</td>
</tr>
<tr>
<td>dimension_reduction_type</td>
<td>The type of dimension reduction method.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: <code>sign</code> for random projection or <code>fjlt</code> for the fast Johnson-Lindenstrauss transform.</td>
</tr>
<tr>
<td></td>
<td>Default value: No dimension reduction</td>
</tr>
<tr>
<td>faiss_index_ivf_nlists</td>
<td>The number of centroids to construct in the index when <code>index_type</code> is <code>faiss.IVFFlat</code> or <code>faiss.IVFPQ</code>.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: positive integer</td>
</tr>
<tr>
<td></td>
<td>Default value: <code>auto</code>, which resolves to <code>sqrt(sample_size)</code>.</td>
</tr>
<tr>
<td>faiss_index_pq_m</td>
<td>The number of vector sub-components to construct in the index when <code>index_type</code> is set to <code>faiss.IVFPQ</code>.</td>
</tr>
<tr>
<td></td>
<td>The FaceBook AI Similarity Search (FAISS) library requires that the value of <code>faiss_index_pq_m</code> is a divisor of the data dimension. If <code>faiss_index_pq_m</code> is not a divisor of the data dimension, we increase the data dimension to smallest integer divisible by <code>faiss_index_pq_m</code>.</td>
</tr>
<tr>
<td></td>
<td>If no dimension reduction is applied, the algorithm adds a padding of zeros. If dimension reduction is applied, the algorithm increase the value of the <code>dimension_reduction_target</code> hyper-parameter.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: One of the following positive integers: 1, 2, 3, 4, 8, 12, 16, 20, 24, 28, 32, 40, 48, 56, 64, 96</td>
</tr>
<tr>
<td>index_metric</td>
<td>The metric to measure the distance between points when finding nearest neighbors. When training with <code>index_type</code> set to <code>faiss.IVFPQ</code>, the <code>INNER_PRODUCT</code> distance and <code>COSINE</code> similarity are not supported.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: <code>L2</code> for Euclidean-distance, <code>INNER_PRODUCT</code> for inner-product distance, <code>COSINE</code> for cosine similarity.</td>
</tr>
<tr>
<td></td>
<td>Default value: <code>L2</code></td>
</tr>
</tbody>
</table>
### Use built-in algorithms

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>index_type</td>
<td>The type of index.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: <code>faiss.Flat</code>, <code>faiss.IVFFlat</code>, <code>faiss.IVFPQ</code>.</td>
</tr>
<tr>
<td></td>
<td>Default values: <code>faiss.Flat</code></td>
</tr>
<tr>
<td>mini_batch_size</td>
<td>The number of observations per mini-batch for the data iterator.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: positive integer</td>
</tr>
<tr>
<td></td>
<td>Default value: 5000</td>
</tr>
</tbody>
</table>

### Tune a k-NN Model

The Amazon SageMaker k-nearest neighbors algorithm is a supervised algorithm. The algorithm consumes a test data set and emits a metric about the accuracy for a classification task or about the mean squared error for a regression task. These accuracy metrics compare the model predictions for their respective task to the ground truth provided by the empirical test data. To find the best model that reports the highest accuracy or lowest error on the test dataset, run a hyperparameter tuning job for k-NN.

**Automatic model tuning**, also known as hyperparameter tuning, finds the best version of a model by running many jobs that test a range of hyperparameters on your dataset. You choose the tunable hyperparameters, a range of values for each, and an objective metric. You choose the objective metric appropriate for the prediction task of the algorithm. Automatic model tuning searches the hyperparameters chosen to find the combination of values that result in the model that optimizes the objective metric. The hyperparameters are used only to help estimate model parameters and are not used by the trained model to make predictions.

For more information about model tuning, see [Perform Automatic Model Tuning](p. 1023).

### Metrics Computed by the k-NN Algorithm

The k-nearest neighbors algorithm computes one of two metrics in the following table during training depending on the type of task specified by the `predictor_type` hyper-parameter.

- `classifier` specifies a classification task and computes `test:accuracy`
- `regressor` specifies a regression task and computes `test:mse`.

Choose the `predictor_type` value appropriate for the type of task undertaken to calculate the relevant objective metric when tuning a model.

<table>
<thead>
<tr>
<th>Metric Name</th>
<th>Description</th>
<th>Optimization Direction</th>
</tr>
</thead>
<tbody>
<tr>
<td>test:accuracy</td>
<td>When <code>predictor_type</code> is set to <code>classifier</code>, k-NN compares the predicted label, based on the average of the k-nearest neighbors' labels, to the ground truth label provided in the test channel data. The accuracy reported ranges from 0.0 (0%) to 1.0 (100%).</td>
<td>Maximize</td>
</tr>
</tbody>
</table>
### Test MSE Metric

**Metric Name**: test:mse  
**Description**: When `predictor_type` is set to `regressor`, k-NN compares the predicted label, based on the average of the k-nearest neighbors’ labels, to the ground truth label provided in the test channel data. The mean squared error is computed by comparing the two labels.  
**Optimization Direction**: Minimize

---

### Tunable k-NN Hyperparameters

Tune the Amazon SageMaker k-nearest neighbor model with the following hyperparameters.

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Parameter Type</th>
<th>Recommended Ranges</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>k</code></td>
<td>IntegerParameterRanges</td>
<td>MinValue: 1, MaxValue: 1024</td>
</tr>
<tr>
<td><code>sample_size</code></td>
<td>IntegerParameterRanges</td>
<td>MinValue: 256, MaxValue: 2000000</td>
</tr>
</tbody>
</table>

---

### Data Formats for k-NN Training Input

All Amazon SageMaker built-in algorithms adhere to the common input training formats described in [Common Data Formats - Training](#). This topic contains a list of the available input formats for the SageMaker k-nearest-neighbor algorithm.

**CSV Data Format**

```plaintext
content-type: text/csv; label_size=1
```

```
4,1.2,1.3,9.6,20.3
```

The first `label_size` columns are interpreted as the label vector for that row.

**RECORDIO Data Format**

```plaintext
content-type: application/x-recordio-protobuf
```

```
[  
  Record = {    
    features = {      
      'values': {        
        values: [1.2, 1.3, 9.6, 20.3] # float32
      }
    },    
    label = {      
      'values': {        
        values: [4] # float32
      }
    }
  }
]
```

---
k-NN Request and Response Formats

All Amazon SageMaker built-in algorithms adhere to the common input inference format described in Common Data Formats - Inference. This topic contains a list of the available output formats for the SageMaker k-nearest-neighbor algorithm.

**INPUT: CSV Request Format**

```
content-type: text/csv

1.2,1.3,9.6,20.3
```

This accepts a `label_size` or encoding parameter. It assumes a `label_size` of 0 and a utf-8 encoding.

**INPUT: JSON Request Format**

```
content-type: application/json

{
   "instances": [
      {
         "data": {
            "features": {
               "values": [-3, -1, -4, 2]
            }
         }
      },
      {
         "features": [3.0, 0.1, 0.04, 0.002]
      }
   ]
}
```

**INPUT: JSONLINES Request Format**

```
content-type: application/jsonlines

{"features": [1.5, 16.0, 14.0, 23.0]}
{"data": {"features": {"values": [1.5, 16.0, 14.0, 23.0]}}
```

**INPUT: RECORDIO Request Format**

```
content-type: application/x-recordio-protobuf

[
   Record = {
      features = {
         'values': {
            values: [-3, -1, -4, 2]  # float32
         }
      },
      label = {}
   },
   Record = {
      features = {
         'values': {
            values: [3.0, 0.1, 0.04, 0.002]  # float32
         }
      },
      label = {}
   }
]
```

**OUTPUT: JSON Response Format**

```
accept: application/json

{
   "predictions": [
```

725
Amazon SageMaker Developer Guide
Use built-in algorithms

}

]

{"predicted_label": 0.0},
{"predicted_label": 2.0}

OUTPUT: JSONLINES Response Format
accept: application/jsonlines
{"predicted_label": 0.0}
{"predicted_label": 2.0}

OUTPUT: VERBOSE JSON Response Format
In verbose mode, the API provides the search results with the distances vector sorted from smallest to
largest, with corresponding elements in the labels vector. In this example, k is set to 3.
accept: application/json; verbose=true
{

}

"predictions": [
{
"predicted_label": 0.0,
"distances": [3.11792408, 3.89746071, 6.32548437],
"labels": [0.0, 1.0, 0.0]
},
{
"predicted_label": 2.0,
"distances": [1.08470316, 3.04917915, 5.25393973],
"labels": [2.0, 2.0, 0.0]
}
]

OUTPUT: RECORDIO-PROTOBUF Response Format
content-type: application/x-recordio-protobuf
[

]

Record = {
features = {},
label = {
'predicted_label':
values: [0.0]
}
}
},
Record = {
features = {},
label = {
'predicted_label':
values: [2.0]
}
}
}

{
# float32

{
# float32

OUTPUT: VERBOSE RECORDIO-PROTOBUF Response Format
In verbose mode, the API provides the search results with the distances vector sorted from smallest to
largest, with corresponding elements in the labels vector. In this example, k is set to 3.

726


SAMPLE OUTPUT for the k-NN Algorithm

For regressor tasks:

```
[06/08/2018 20:15:33 INFO 140026520049408] #test_score (algo-1) : ('mse',
0.013333333333333334)
```

For classifier tasks:

```
[06/08/2018 20:15:46 INFO 140285487171328] #test_score (algo-1) : ('accuracy',
0.9866666666666669)
```

Latent Dirichlet Allocation (LDA) Algorithm

The Amazon SageMaker Latent Dirichlet Allocation (LDA) algorithm is an unsupervised learning algorithm that attempts to describe a set of observations as a mixture of distinct categories. LDA is most commonly used to discover a user-specified number of topics shared by documents within a text corpus. Here each observation is a document, the features are the presence (or occurrence count) of each word, and the categories are the topics. Since the method is unsupervised, the topics are not specified up front, and are not guaranteed to align with how a human may naturally categorize documents. The topics are learned as a probability distribution over the words that occur in each document. Each document, in turn, is described as a mixture of topics.

The exact content of two documents with similar topic mixtures will not be the same. But overall, you would expect these documents to more frequently use a shared subset of words, than when compared...
with a document from a different topic mixture. This allows LDA to discover these word groups and use them to form topics. As an extremely simple example, given a set of documents where the only words that occur within them are: eat, sleep, play, meow, and bark, LDA might produce topics like the following:

<table>
<thead>
<tr>
<th>Topic</th>
<th>eat</th>
<th>sleep</th>
<th>play</th>
<th>meow</th>
<th>bark</th>
</tr>
</thead>
<tbody>
<tr>
<td>Topic 1</td>
<td>0.1</td>
<td>0.3</td>
<td>0.2</td>
<td>0.4</td>
<td>0.0</td>
</tr>
<tr>
<td>Topic 2</td>
<td>0.2</td>
<td>0.1</td>
<td>0.4</td>
<td>0.0</td>
<td>0.3</td>
</tr>
</tbody>
</table>

You can infer that documents that are more likely to fall into Topic 1 are about cats (who are more likely to meow and sleep), and documents that fall into Topic 2 are about dogs (who prefer to play and bark). These topics can be found even though the words dog and cat never appear in any of the texts.

**Topics**

- Input/Output Interface for the LDA Algorithm (p. 728)
- EC2 Instance Recommendation for the LDA Algorithm (p. 728)
- LDA Sample LDA Notebooks (p. 728)
- How LDA Works (p. 729)
- LDA Hyperparameters (p. 730)
- Tune an LDA Model (p. 731)

**Input/Output Interface for the LDA Algorithm**

LDA expects data to be provided on the train channel, and optionally supports a test channel, which is scored by the final model. LDA supports both recordIO-wrapped-protobuf (dense and sparse) and CSV file formats. For CSV, the data must be dense and have dimension equal to number of records * vocabulary size. LDA can be trained in File or Pipe mode when using recordIO-wrapped protobuf, but only in File mode for the CSV format.

For inference, text/csv, application/json, and application/x-recordio-protobuf content types are supported. Sparse data can also be passed for application/json and application/x-recordio-protobuf. LDA inference returns application/json or application/x-recordio-protobuf predictions, which include the topic_mixture vector for each observation.

Please see the LDA Sample LDA Notebooks (p. 728) for more detail on training and inference formats.

**EC2 Instance Recommendation for the LDA Algorithm**

LDA currently only supports single-instance CPU training. CPU instances are recommended for hosting/inference.

**LDA Sample LDA Notebooks**

For a sample notebook that shows how to train the SageMaker Latent Dirichlet Allocation algorithm on a dataset and then how to deploy the trained model to perform inferences about the topic mixtures in input documents, see the An Introduction to SageMaker LDA. For instructions how to create and access Jupyter notebook instances that you can use to run the example in SageMaker, see Use Amazon SageMaker Notebook Instances (p. 124). Once you have created a notebook instance and opened it, select the SageMaker Examples tab to see a list of all the SageMaker samples. The topic modeling example notebooks using the NTM algorithms are located in the Introduction to Amazon algorithms section. To open a notebook, click on its Use tab and select Create copy.
How LDA Works

Amazon SageMaker LDA is an unsupervised learning algorithm that attempts to describe a set of observations as a mixture of different categories. These categories are themselves a probability distribution over the features. LDA is a generative probability model, which means it attempts to provide a model for the distribution of outputs and inputs based on latent variables. This is opposed to discriminative models, which attempt to learn how inputs map to outputs.

You can use LDA for a variety of tasks, from clustering customers based on product purchases to automatic harmonic analysis in music. However, it is most commonly associated with topic modeling in text corpuses. Observations are referred to as documents. The feature set is referred to as vocabulary. A feature is referred to as a word. And the resulting categories are referred to as topics.

Note
Lemmatization significantly increases algorithm performance and accuracy. Consider pre-processing any input text data.

An LDA model is defined by two parameters:
- $\alpha$—A prior estimate on topic probability (in other words, the average frequency that each topic within a given document occurs).
- $\beta$—a collection of $k$ topics where each topic is given a probability distribution over the vocabulary used in a document corpus, also called a “topic-word distribution.”

LDA is a "bag-of-words" model, which means that the order of words does not matter. LDA is a generative model where each document is generated word-by-word by choosing a topic mixture $\theta \sim \text{Dirichlet}(\alpha)$.

For each word in the document:
- Choose a topic $z \sim \text{Multinomial}(\theta)$
- Choose the corresponding topic-word distribution $\beta_z$.
- Draw a word $w \sim \text{Multinomial}(\beta_z)$.

When training the model, the goal is to find parameters $\alpha$ and $\beta$, which maximize the probability that the text corpus is generated by the model.

The most popular methods for estimating the LDA model use Gibbs sampling or Expectation Maximization (EM) techniques. The Amazon SageMaker LDA uses tensor spectral decomposition. This provides several advantages:

- **Theoretical guarantees on results.** The standard EM-method is guaranteed to converge only to local optima, which are often of poor quality.
- **Embarrassingly parallelizable.** The work can be trivially divided over input documents in both training and inference. The EM-method and Gibbs Sampling approaches can be parallelized, but not as easily.
- **Fast.** Although the EM-method has low iteration cost it is prone to slow convergence rates. Gibbs Sampling is also subject to slow convergence rates and also requires a large number of samples.

At a high-level, the tensor decomposition algorithm follows this process:

1. The goal is to calculate the spectral decomposition of a $V \times V \times V$ tensor, which summarizes the moments of the documents in our corpus. $V$ is vocabulary size (in other words, the number of distinct words in all of the documents). The spectral components of this tensor are the LDA parameters $\alpha$ and $\beta$, which maximize the overall likelihood of the document corpus. However, because vocabulary size tends to be large, this $V \times V \times V$ tensor is prohibitively large to store in memory.
2. Instead, it uses a $V \times V$ moment matrix, which is the two-dimensional analog of the tensor from step 1, to find a whitening matrix of dimension $V \times k$. This matrix can be used to convert the $V \times V$ moment matrix into a $k \times k$ identity matrix. $k$ is the number of topics in the model.

3. This same whitening matrix can then be used to find a smaller $k \times k \times k$ tensor. When spectrally decomposed, this tensor has components that have a simple relationship with the components of the $V \times V \times V$ tensor.

4. **Alternating Least Squares** is used to decompose the smaller $k \times k \times k$ tensor. This provides a substantial improvement in memory consumption and speed. The parameters $\alpha$ and $\beta$ can be found by "unwhitening" these outputs in the spectral decomposition.

After the LDA model's parameters have been found, you can find the topic mixtures for each document. You use stochastic gradient descent to maximize the likelihood function of observing a given topic mixture corresponding to these data.

Topic quality can be improved by increasing the number of topics to look for in training and then filtering out poor quality ones. This is in fact done automatically in SageMaker LDA: 25% more topics are computed and only the ones with largest associated Dirichlet priors are returned. To perform further topic filtering and analysis, you can increase the topic count and modify the resulting LDA model as follows:

```python
> import mxnet as mx
> alpha, beta = mx.ndarray.load('model.tar.gz')
> # modify alpha and beta
> mx.nd.save('new_model.tar.gz', [new_alpha, new_beta])
> # upload to S3 and create new SageMaker model using the console
```

For more information about algorithms for LDA and the SageMaker implementation, see the following:


**LDA Hyperparameters**

In the CreateTrainingJob request, you specify the training algorithm. You can also specify algorithm-specific hyperparameters as string-to-string maps. The following table lists the hyperparameters for the LDA training algorithm provided by Amazon SageMaker. For more information, see How LDA Works (p. 729).

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>num_topics</td>
<td>The number of topics for LDA to find within the data.</td>
</tr>
<tr>
<td></td>
<td>Required</td>
</tr>
<tr>
<td></td>
<td>Valid values: positive integer</td>
</tr>
<tr>
<td>feature_dim</td>
<td>The size of the vocabulary of the input document corpus.</td>
</tr>
<tr>
<td></td>
<td>Required</td>
</tr>
<tr>
<td>Parameter Name</td>
<td>Description</td>
</tr>
<tr>
<td>----------------</td>
<td>-------------</td>
</tr>
<tr>
<td>mini_batch_size</td>
<td>The total number of documents in the input document corpus. <strong>Required</strong> Valid values: positive integer</td>
</tr>
<tr>
<td>alpha0</td>
<td>Initial guess for the concentration parameter: the sum of the elements of the Dirichlet prior. Small values are more likely to generate sparse topic mixtures and large values (greater than 1.0) produce more uniform mixtures. <strong>Optional</strong> Valid values: Positive float Default value: 1.0</td>
</tr>
<tr>
<td>max_restarts</td>
<td>The number of restarts to perform during the Alternating Least Squares (ALS) spectral decomposition phase of the algorithm. Can be used to find better quality local minima at the expense of additional computation, but typically should not be adjusted. <strong>Optional</strong> Valid values: Positive integer Default value: 10</td>
</tr>
<tr>
<td>max_iterations</td>
<td>The maximum number of iterations to perform during the ALS phase of the algorithm. Can be used to find better quality minima at the expense of additional computation, but typically should not be adjusted. <strong>Optional</strong> Valid values: Positive integer Default value: 1000</td>
</tr>
<tr>
<td>tol</td>
<td>Target error tolerance for the ALS phase of the algorithm. Can be used to find better quality minima at the expense of additional computation, but typically should not be adjusted. <strong>Optional</strong> Valid values: Positive float Default value: 1e-8</td>
</tr>
</tbody>
</table>

**Tune an LDA Model**

*Automatic model tuning*, also known as hyperparameter tuning, finds the best version of a model by running many jobs that test a range of hyperparameters on your dataset. You choose the tunable hyperparameters, a range of values for each, and an objective metric. You choose the objective metric from the metrics that the algorithm computes. Automatic model tuning searches the hyperparameters chosen to find the combination of values that result in the model that optimizes the objective metric.
LDA is an unsupervised topic modeling algorithm that attempts to describe a set of observations (documents) as a mixture of different categories (topics). The “per-word log-likelihood” (PWLL) metric measures the likelihood that a learned set of topics (an LDA model) accurately describes a test document dataset. Larger values of PWLL indicate that the test data is more likely to be described by the LDA model.

For more information about model tuning, see Perform Automatic Model Tuning (p. 1023).

Metrics Computed by the LDA Algorithm

The LDA algorithm reports on a single metric during training: \texttt{test:pwll}. When tuning a model, choose this metric as the objective metric.

<table>
<thead>
<tr>
<th>Metric Name</th>
<th>Description</th>
<th>Optimization Direction</th>
</tr>
</thead>
<tbody>
<tr>
<td>test:pwll</td>
<td>Per-word log-likelihood on the test dataset. The likelihood that the test dataset is accurately described by the learned LDA model.</td>
<td>Maximize</td>
</tr>
</tbody>
</table>

Tunable LDA Hyperparameters

You can tune the following hyperparameters for the LDA algorithm. Both hyperparameters, \texttt{alpha0} and \texttt{num_topics}, can affect the LDA objective metric (test:pwll). If you don't already know the optimal values for these hyperparameters, which maximize per-word log-likelihood and produce an accurate LDA model, automatic model tuning can help find them.

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Parameter Type</th>
<th>Recommended Ranges</th>
</tr>
</thead>
<tbody>
<tr>
<td>alpha0</td>
<td>ContinuousParameterRanges</td>
<td>MinValue: 0.1, MaxValue: 10</td>
</tr>
<tr>
<td>num_topics</td>
<td>IntegerParameterRanges</td>
<td>MinValue: 1, MaxValue: 150</td>
</tr>
</tbody>
</table>

Linear Learner Algorithm

Linear models are supervised learning algorithms used for solving either classification or regression problems. For input, you give the model labeled examples \((x, y)\), where \(x\) is a high-dimensional vector and \(y\) is a numeric label. For binary classification problems, the label must be either 0 or 1. For multiclass classification problems, the labels must be from 0 to \(\text{num\_classes} - 1\). For regression problems, \(y\) is a real number. The algorithm learns a linear function, or, for classification problems, a linear threshold function, and maps a vector \(x\) to an approximation of the label \(y\).

The Amazon SageMaker linear learner algorithm provides a solution for both classification and regression problems. With the SageMaker algorithm, you can simultaneously explore different training objectives and choose the best solution from a validation set. You can also explore a large number of models and choose the best. The best model optimizes either of the following:

- Continuous objectives, such as mean square error, cross entropy loss, absolute error.
- Discrete objectives suited for classification, such as F1 measure, precision, recall, or accuracy.

Compared with methods that provide a solution for only continuous objectives, the SageMaker linear learner algorithm provides a significant increase in speed over naive hyperparameter optimization techniques. It is also more convenient.
The linear learner algorithm requires a data matrix, with rows representing the observations, and columns representing the dimensions of the features. It also requires an additional column that contains the labels that match the data points. At a minimum, Amazon SageMaker linear learner requires you to specify input and output data locations, and objective type (classification or regression) as arguments. The feature dimension is also required. For more information, see CreateTrainingJob. You can specify additional parameters in the HyperParameters string map of the request body. These parameters control the optimization procedure, or specifics of the objective function that you train on. For example, the number of epochs, regularization, and loss type.

If you're using Managed Spot Training, the linear learner algorithm supports using checkpoints to take a snapshot of the state of the model.

**Topics**

- Input/Output interface for the linear learner algorithm (p. 733)
- EC2 instance recommendation for the linear learner algorithm (p. 734)
- Linear learner sample notebooks (p. 734)
- How linear learner works (p. 734)
- Linear learner hyperparameters (p. 735)
- Tune a linear learner model (p. 744)
- Linear learner response formats (p. 746)

**Input/Output interface for the linear learner algorithm**

The Amazon SageMaker linear learner algorithm supports three data channels: train, validation (optional), and test (optional). If you provide validation data, the S3DataDistributionType should be FullyReplicated. The algorithm logs validation loss at every epoch, and uses a sample of the validation data to calibrate and select the best model. If you don't provide validation data, the algorithm uses a sample of the training data to calibrate and select the model. If you provide test data, the algorithm logs include the test score for the final model.

For training, the linear learner algorithm supports both recordIO-wrapped protobuf and CSV formats. For the application/x-recordio-protobuf input type, only Float32 tensors are supported. For the text/csv input type, the first column is assumed to be the label, which is the target variable for prediction. You can use either File mode or Pipe mode to train linear learner models on data that is formatted as recordIO-wrapped-protobuf or as CSV.

For inference, the linear learner algorithm supports the application/json, application/x-recordio-protobuf, and text/csv formats. When you make predictions on new data, the format of the response depends on the type of model. For regression (predictor_type='regressor'), the score is the prediction produced by the model. For classification (predictor_type='binary_classifier' or predictor_type='multiclass_classifier'), the model returns a score and also a predicted_label. The predicted_label is the class predicted by the model and the score measures the strength of that prediction.

- For binary classification, predicted_label is 0 or 1, and score is a single floating point number that indicates how strongly the algorithm believes that the label should be 1.
- For multiclass classification, the predicted_class will be an integer from 0 to num_classes-1, and score will be a list of one floating point number per class.

To interpret the score in classification problems, you have to consider the loss function used. If the loss hyperparameter value is logistic for binary classification or softmax_loss for multiclass classification, then the score can be interpreted as the probability of the corresponding class. These
are the loss values used by the linear learner when the loss value is auto default value. But if the loss
is set to hinge_loss, then the score cannot be interpreted as a probability. This is because hinge loss
corresponds to a Support Vector Classifier, which does not produce probability estimates.

For more information on input and output file formats, see Linear learner response formats (p. 746).
For more information on inference formats, and the Linear learner sample notebooks (p. 734).

**EC2 instance recommendation for the linear learner algorithm**

You can train the linear learner algorithm on single- or multi-machine CPU and GPU instances. During
testing, we have not found substantial evidence that multi-GPU computers are faster than single-GPU
computers. Results can vary, depending on your specific use case.

**Linear learner sample notebooks**

The following table outlines a variety of sample notebooks that address different use cases of Amazon
SageMaker linear learner algorithm.

<table>
<thead>
<tr>
<th>Notebook Title</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>An Introduction with the MNIST dataset</td>
<td>Using the MNIST dataset, we train a binary classifier to predict a single digit.</td>
</tr>
<tr>
<td>Predict Breast Cancer</td>
<td>Using UCI's Breast Cancer dataset, we train a model to predict Breast Cancer.</td>
</tr>
<tr>
<td>How to Build a Multiclass Classifier?</td>
<td>Using UCI's Covertype dataset, we demonstrate how to train a multiclass classifier.</td>
</tr>
<tr>
<td>How to Build a Machine Learning (ML) Pipeline for Inference?</td>
<td>Using a Scikit-learn container, we demonstrate how to build an end-to-end ML pipeline.</td>
</tr>
</tbody>
</table>

For instructions on how to create and access Jupyter notebook instances that you can use to run the
template in SageMaker, see Use Amazon SageMaker Notebook Instances (p. 124). After you have created
a notebook instance and opened it, choose the SageMaker Examples tab to see a list of all of the
SageMaker samples. The topic modeling example notebooks using the linear learning algorithm are
located in the Introduction to Amazon algorithms section. To open a notebook, choose its Use tab and
choose Create copy.

**How linear learner works**

There are three steps involved in the implementation of the linear learner algorithm: preprocess, train,
and validate. Step 4 provides example code that shows how to deploy a trained model.

**Step 1: Preprocess**

Normalization, or feature scaling, is an important preprocessing step for certain loss functions that
ensures the model being trained on a dataset does not become dominated by the weight of a single
feature. The Amazon SageMaker Linear Learner algorithm has a normalization option to assist with this
preprocessing step. If normalization is turned on, the algorithm first goes over a small sample of the data
to learn the mean value and standard deviation for each feature and for the label. Each of the features in
the full dataset is then shifted to have mean of zero and scaled to have a unit standard deviation.

**Note**

For best results, ensure your data is shuffled before training. Training with unshuffled data may
cause training to fail.
You can configure whether the linear learner algorithm normalizes the feature data and the labels using the `normalize_data` and `normalize_label` hyperparameters, respectively. Normalization is enabled by default for both features and labels for regression. Only the features can be normalized for binary classification and this is the default behavior.

**Step 2: Train**

With the linear learner algorithm, you train with a distributed implementation of stochastic gradient descent (SGD). You can control the optimization process by choosing the optimization algorithm. For example, you can choose to use Adam, AdaGrad, stochastic gradient descent, or other optimization algorithms. You also specify their hyperparameters, such as momentum, learning rate, and the learning rate schedule. If you aren't sure which algorithm or hyperparameter value to use, choose a default that works for the majority of datasets.

During training, you simultaneously optimize multiple models, each with slightly different objectives. For example, you vary L1 or L2 regularization and try out different optimizer settings.

**Step 3: Validate and set the threshold**

When training multiple models in parallel, the models are evaluated against a validation set to select the most optimal model once training is complete. For regression, the most optimal model is the one that achieves the best loss on the validation set. For classification, a sample of the validation set is used to calibrate the classification threshold. The most optimal model selected is the one that achieves the best binary classification selection criteria on the validation set. Examples of such criteria include the F1 measure, accuracy, and cross-entropy loss.

**Note**

If the algorithm is not provided a validation set, then evaluating and selecting the most optimal model is not possible. To take advantage of parallel training and model selection ensure you provide a validation set to the algorithm.

**Step 4: Deploy a trained linear model**

Here is an example of Python code that you can use to deploy a model in MXNet that has been trained with SageMaker linear learner

```python
import mxnet as mx
import tarfile
t = tarfile.open('model.tar.gz', 'r:gz')
t.extractall()

# Load the mxnet module from the model files
mod = mx.module.Module.load('mx-mod', 0)
# model's weights
mod._arg_params['fc0_weight'].asnumpy().flatten()

# model bias
mod._arg_params['fc0_bias'].asnumpy().flatten()

#perform inference
result = mod.predict(data_iter)
```

**Linear learner hyperparameters**

The following table contains the hyperparameters for the linear learner algorithm. These are parameters that are set by users to facilitate the estimation of model parameters from data. The required hyperparameters that must be set are listed first, in alphabetical order. The optional hyperparameters that can be set are listed next, also in alphabetical order. When a hyperparameter is set to `auto`, Amazon SageMaker will automatically calculate and set the value of that hyperparameter.
<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>num_classes</td>
<td>The number of classes for the response variable. The algorithm assumes that classes are labeled 0, ..., num_classes - 1. Required when predictor_type is multiclass_classifier. Otherwise, the algorithm ignores it. Valid values: Integers from 3 to 1,000,000.</td>
</tr>
<tr>
<td>predictor_type</td>
<td>Specifies the type of target variable as a binary classification, multiclass classification, or regression. Required Valid values: binary_classifier, multiclass_classifier, or regressor.</td>
</tr>
<tr>
<td>accuracy_top_k</td>
<td>When computing the top-k accuracy metric for multiclass classification, the value of k. If the model assigns one of the top-k scores to the true label, an example is scored as correct. Optional Valid values: Positive integers Default value: 3</td>
</tr>
<tr>
<td>balance_multiclass_weights</td>
<td>Specifies whether to use class weights, which give each class equal importance in the loss function. Used only when the predictor_type is multiclass_classifier. Optional Valid values: true, false Default value: false</td>
</tr>
<tr>
<td>beta_1</td>
<td>The exponential decay rate for first-moment estimates. Applies only when the optimizer value is adam. Optional Valid values: auto or floating-point value between 0 and 1.0 Default value: auto</td>
</tr>
<tr>
<td>beta_2</td>
<td>The exponential decay rate for second-moment estimates. Applies only when the optimizer value is adam. Optional Valid values: auto or floating-point integer between 0 and 1.0 Default value: auto</td>
</tr>
<tr>
<td>bias_lr_mult</td>
<td>Allows a different learning rate for the bias term. The actual learning rate for the bias is learning_rate * bias_lr_mult. Optional</td>
</tr>
<tr>
<td>Parameter Name</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td></td>
<td>Valid values: auto or positive floating-point integer</td>
</tr>
<tr>
<td></td>
<td>Default value: auto</td>
</tr>
<tr>
<td>bias_wd_mult</td>
<td>Allows different regularization for the bias term. The actual L2 regularization weight for the bias is $wd \times bias_wd_mult$. By default, there is no regularization on the bias term.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: auto or non-negative floating-point integer</td>
</tr>
<tr>
<td></td>
<td>Default value: auto</td>
</tr>
<tr>
<td>binary_classifier_model_selection_criteria</td>
<td>When <code>predictor_type</code> is set to binary_classifier, the model evaluation criteria for the validation dataset (or for the training dataset if you don't provide a validation dataset). Criteria include:</td>
</tr>
<tr>
<td></td>
<td>• accuracy — The model with the highest accuracy.</td>
</tr>
<tr>
<td></td>
<td>• f_beta — The model with the highest F1 score. The default is F1.</td>
</tr>
<tr>
<td></td>
<td>• precision_at_target_recall — The model with the highest precision at a given recall target.</td>
</tr>
<tr>
<td></td>
<td>• recall_at_target_precision — The model with the highest recall at a given precision target.</td>
</tr>
<tr>
<td></td>
<td>• loss_function — The model with the lowest value of the loss function used in training.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: accuracy, f_beta, precision_at_target_recall, recall_at_target_precision, or loss_function</td>
</tr>
<tr>
<td></td>
<td>Default value: accuracy</td>
</tr>
<tr>
<td>early_stopping_patience</td>
<td>If no improvement is made in the relevant metric, the number of epochs to wait before ending training. If you have provided a value for binary_classifier_model_selection_criteria, the metric is that value. Otherwise, the metric is the same as the value specified for the loss hyperparameter.</td>
</tr>
<tr>
<td></td>
<td>The metric is evaluated on the validation data. If you haven't provided validation data, the metric is always the same as the value specified for the loss hyperparameter and is evaluated on the training data. To disable early stopping, set early_stopping_patience to a value greater than the value specified for epochs.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: Positive integer</td>
</tr>
<tr>
<td></td>
<td>Default value: 3</td>
</tr>
<tr>
<td>Parameter Name</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>early_stopping_tolerance</td>
<td>The relative tolerance to measure an improvement in loss. If the ratio of the improvement in loss divided by the previous best loss is smaller than this value, early stopping considers the improvement to be zero.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: Positive floating-point integer</td>
</tr>
<tr>
<td></td>
<td>Default value: 0.001</td>
</tr>
<tr>
<td>epochs</td>
<td>The maximum number of passes over the training data.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: Positive integer</td>
</tr>
<tr>
<td></td>
<td>Default value: 15</td>
</tr>
<tr>
<td>f_beta</td>
<td>The value of beta to use when calculating F score metrics for binary or multiclass classification. Also used if the value specified for binary_classifier_model_selection_criteria is f_beta.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: Positive floating-point integers</td>
</tr>
<tr>
<td></td>
<td>Default value: 1.0</td>
</tr>
<tr>
<td>feature_dim</td>
<td>The number of features in the input data.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: auto or positive integer</td>
</tr>
<tr>
<td></td>
<td>Default values: auto</td>
</tr>
<tr>
<td>huber_delta</td>
<td>The parameter for Huber loss. During training and metric evaluation, compute L2 loss for errors smaller than delta and L1 loss for errors larger than delta.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: Positive floating-point integer</td>
</tr>
<tr>
<td></td>
<td>Default value: 1.0</td>
</tr>
<tr>
<td>init_bias</td>
<td>Initial weight for the bias term.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: Floating-point integer</td>
</tr>
<tr>
<td></td>
<td>Default value: 0</td>
</tr>
<tr>
<td>Parameter Name</td>
<td>Description</td>
</tr>
<tr>
<td>----------------</td>
<td>-------------</td>
</tr>
</tbody>
</table>
| init_method    | Sets the initial distribution function used for model weights. Functions include:  
• uniform—Uniformly distributed between (-scale, +scale)  
• normal—Normal distribution, with mean 0 and sigma  
Optional  
Valid values: uniform or normal  
Default value: uniform |
| init_scale     | Scales an initial uniform distribution for model weights. Applies only when the `init_method` hyperparameter is set to uniform.  
Optional  
Valid values: Positive floating-point integer  
Default value: 0.07 |
| init_sigma     | The initial standard deviation for the normal distribution. Applies only when the `init_method` hyperparameter is set to normal.  
Optional  
Valid values: Positive floating-point integer  
Default value: 0.01 |
| l1             | The L1 regularization parameter. If you don’t want to use L1 regularization, set the value to 0.  
Optional  
Valid values: auto or non-negative float  
Default value: auto |
| learning_rate  | The step size used by the optimizer for parameter updates.  
Optional  
Valid values: auto or positive floating-point integer  
Default value: auto, whose value depends on the optimizer chosen. |
<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>loss</td>
<td>Specifies the loss function. The available loss functions and their default values depend on the value of predictor_type:</td>
</tr>
<tr>
<td></td>
<td>• If the predictor_type is set to regressor, the available options are auto, squared_loss, absolute_loss, eps_insensitive_squared_loss, eps_insensitive_absolute_loss, quantile_loss, and huber_loss. The default value for auto is squared_loss.</td>
</tr>
<tr>
<td></td>
<td>• If the predictor_type is set to binary_classifier, the available options are auto, logistic, and hinge_loss. The default value for auto is logistic.</td>
</tr>
<tr>
<td></td>
<td>• If the predictor_type is set to multiclass_classifier, the available options are auto and softmax_loss. The default value for auto is softmax_loss.</td>
</tr>
<tr>
<td></td>
<td>Valid values: auto, logistic, squared_loss, absolute_loss, hinge_loss, eps_insensitive_squared_loss, eps_insensitive_absolute_loss, quantile_loss, or huber_loss</td>
</tr>
<tr>
<td></td>
<td>Optional</td>
</tr>
<tr>
<td></td>
<td>Default value: auto</td>
</tr>
<tr>
<td>loss_insensitivity</td>
<td>The parameter for the epsilon-insensitive loss type. During training and metric evaluation, any error smaller than this value is considered to be zero.</td>
</tr>
<tr>
<td></td>
<td>Optional</td>
</tr>
<tr>
<td></td>
<td>Valid values: Positive floating-point integer</td>
</tr>
<tr>
<td></td>
<td>Default value: 0.01</td>
</tr>
<tr>
<td>lr_scheduler_factor</td>
<td>For every lr_scheduler_step hyperparameter, the learning rate decreases by this quantity. Applies only when the use_lr_scheduler hyperparameter is set to true.</td>
</tr>
<tr>
<td></td>
<td>Optional</td>
</tr>
<tr>
<td></td>
<td>Valid values: auto or positive floating-point integer between 0 and 1</td>
</tr>
<tr>
<td></td>
<td>Default value: auto</td>
</tr>
<tr>
<td>lr_scheduler_minimum</td>
<td>The learning rate never decreases to a value lower than the value set for lr_scheduler_minimum_lr. Applies only when the use_lr_scheduler hyperparameter is set to true.</td>
</tr>
<tr>
<td></td>
<td>Optional</td>
</tr>
<tr>
<td></td>
<td>Valid values: auto or positive floating-point integer</td>
</tr>
<tr>
<td></td>
<td>Default values: auto</td>
</tr>
<tr>
<td>Parameter Name</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>lr_scheduler_step</td>
<td>The number of steps between decreases of the learning rate. Applies only when the use_lr_scheduler hyperparameter is set to true.</td>
</tr>
<tr>
<td></td>
<td>Optional</td>
</tr>
<tr>
<td></td>
<td>Valid values: auto or positive integer</td>
</tr>
<tr>
<td></td>
<td>Default value: auto</td>
</tr>
<tr>
<td>margin</td>
<td>The margin for the hinge_loss function.</td>
</tr>
<tr>
<td></td>
<td>Optional</td>
</tr>
<tr>
<td></td>
<td>Valid values: Positive floating-point integer</td>
</tr>
<tr>
<td></td>
<td>Default value: 1.0</td>
</tr>
<tr>
<td>mini_batch_size</td>
<td>The number of observations per mini-batch for the data iterator.</td>
</tr>
<tr>
<td></td>
<td>Optional</td>
</tr>
<tr>
<td></td>
<td>Valid values: Positive integer</td>
</tr>
<tr>
<td></td>
<td>Default value: 1000</td>
</tr>
<tr>
<td>momentum</td>
<td>The momentum of the sgd optimizer.</td>
</tr>
<tr>
<td></td>
<td>Optional</td>
</tr>
<tr>
<td></td>
<td>Valid values: auto or a floating-point integer between 0 and 1.0</td>
</tr>
<tr>
<td></td>
<td>Default value: auto</td>
</tr>
<tr>
<td>normalize_data</td>
<td>Normalizes the feature data before training. Data normalization shifts the data for each feature to have a mean of zero and scales it to have unit standard deviation.</td>
</tr>
<tr>
<td></td>
<td>Optional</td>
</tr>
<tr>
<td></td>
<td>Valid values: auto, true, or false</td>
</tr>
<tr>
<td></td>
<td>Default value: true</td>
</tr>
<tr>
<td>normalize_label</td>
<td>Normalizes the label. Label normalization shifts the label to have a mean of zero and scales it to have unit standard deviation.</td>
</tr>
<tr>
<td></td>
<td>The auto default value normalizes the label for regression problems but does not for classification problems. If you set the normalize_label hyperparameter to true for classification problems, the algorithm ignores it.</td>
</tr>
<tr>
<td></td>
<td>Optional</td>
</tr>
<tr>
<td></td>
<td>Valid values: auto, true, or false</td>
</tr>
<tr>
<td></td>
<td>Default value: auto</td>
</tr>
<tr>
<td>Parameter Name</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>num_calibration_samples</td>
<td>The number of observations from the validation dataset to use for model calibration (when finding the best threshold). Optional Valid values: auto or positive integer Default value: auto</td>
</tr>
<tr>
<td>num_models</td>
<td>The number of models to train in parallel. For the default, auto, the algorithm decides the number of parallel models to train. One model is trained according to the given training parameter (regularization, optimizer, loss), and the rest by close parameters. Optional Valid values: auto or positive integer Default values: auto</td>
</tr>
<tr>
<td>num_point_for_scaler</td>
<td>The number of data points to use for calculating normalization or unbiasing of terms. Optional Valid values: Positive integer Default value: 10,000</td>
</tr>
<tr>
<td>optimizer</td>
<td>The optimization algorithm to use. Optional Valid values:  • auto—The default value.  • sgd—Stochastic gradient descent.  • adam—Adaptive momentum estimation.  • rmsprop—A gradient-based optimization technique that uses a moving average of squared gradients to normalize the gradient. Default value: auto. The default setting for auto is adam.</td>
</tr>
<tr>
<td>positive_example_weight_mult</td>
<td>The weight assigned to positive examples when training a binary classifier. The weight of negative examples is fixed at 1. If you want the algorithm to choose a weight so that errors in classifying negative vs. positive examples have equal impact on training loss, specify balanced. If you want the algorithm to choose the weight that optimizes performance, specify auto. Optional Valid values: balanced, auto, or a positive floating-point integer Default value: 1.0</td>
</tr>
<tr>
<td>Parameter Name</td>
<td>Description</td>
</tr>
<tr>
<td>------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>quantile</td>
<td>The quantile for quantile loss. For quantile q, the model attempts to produce predictions so that the value of true_label is greater than the prediction with probability q.</td>
</tr>
<tr>
<td>target_precision</td>
<td>The target precision. If binary_classifier_model_selection_criteria is recall_at_target_precision, then precision is held at this value while recall is maximized.</td>
</tr>
<tr>
<td>target_recall</td>
<td>The target recall. If binary_classifier_model_selection_criteria is precision_at_target_recall, then recall is held at this value while precision is maximized.</td>
</tr>
<tr>
<td>unbias_data</td>
<td>Unbiases the features before training so that the mean is 0. By default data is unbiased as the use_bias hyperparameter is set to true.</td>
</tr>
<tr>
<td>unbias_label</td>
<td>Unbiases labels before training so that the mean is 0. Applies to regression only if the use_bias hyperparameter is set to true.</td>
</tr>
<tr>
<td>use_bias</td>
<td>Specifies whether the model should include a bias term, which is the intercept term in the linear equation.</td>
</tr>
</tbody>
</table>
Tune a linear learner model

Automatic model tuning, also known as hyperparameter tuning, finds the best version of a model by running many jobs that test a range of hyperparameters on your dataset. You choose the tunable hyperparameters, a range of values for each, and an objective metric. You choose the objective metric from the metrics that the algorithm computes. Automatic model tuning searches the hyperparameters chosen to find the combination of values that result in the model that optimizes the objective metric.

The linear learner algorithm also has an internal mechanism for tuning hyperparameters separate from the automatic model tuning feature described here. By default, the linear learner algorithm tunes hyperparameters by training multiple models in parallel. When you use automatic model tuning, the linear learner internal tuning mechanism is turned off automatically. This sets the number of parallel models, num_models, to 1. The algorithm ignores any value that you set for num_models.

For more information about model tuning, see Perform Automatic Model Tuning (p. 1023).

Metrics computed by the linear learner algorithm

The linear learner algorithm reports the metrics in the following table, which are computed during training. Choose one of them as the objective metric. To avoid overfitting, we recommend tuning the model against a validation metric instead of a training metric.

<table>
<thead>
<tr>
<th>Metric Name</th>
<th>Description</th>
<th>Optimization Direction</th>
</tr>
</thead>
<tbody>
<tr>
<td>test:objective_loss</td>
<td>The mean value of the objective loss function on the test dataset after the model is trained. By default, the loss is logistic loss for binary classification and squared loss for regression. To set the loss to other types, use the loss hyperparameter.</td>
<td>Minimize</td>
</tr>
<tr>
<td>test:binary_classification_accuracy</td>
<td>The accuracy of the final model on the test dataset.</td>
<td>Maximize</td>
</tr>
<tr>
<td>test:binary_f_beta</td>
<td>The F_beta score of the final model on the test dataset. By default, it is the F1 score, which is the harmonic mean of precision and recall.</td>
<td>Maximize</td>
</tr>
<tr>
<td>Metric Name</td>
<td>Description</td>
<td>Optimization Direction</td>
</tr>
<tr>
<td>------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>-------------------------</td>
</tr>
<tr>
<td>test:precision</td>
<td>The precision of the final model on the test dataset. If you choose this metric as the objective, we recommend setting a target recall by setting the <code>binary_classifier_model_selection</code> hyperparameter to <code>precision_at_target_recall</code> and setting the value for the <code>target_recall</code> hyperparameter.</td>
<td>Maximize</td>
</tr>
<tr>
<td>test:recall</td>
<td>The recall of the final model on the test dataset. If you choose this metric as the objective, we recommend setting a target precision by setting the <code>binary_classifier_model_selection</code> hyperparameter to <code>recall_at_target_precision</code> and setting the value for the <code>target_precision</code> hyperparameter.</td>
<td>Maximize</td>
</tr>
<tr>
<td>validation:objective</td>
<td>The mean value of the objective loss function on the validation dataset every epoch. By default, the loss is logistic loss for binary classification and squared loss for regression. To set loss to other types, use the <code>loss</code> hyperparameter.</td>
<td>Minimize</td>
</tr>
<tr>
<td>validation:binary_classification_accuracy</td>
<td>The accuracy of the final model on the validation dataset.</td>
<td>Maximize</td>
</tr>
<tr>
<td>validation:binary_f_beta</td>
<td>The F_beta score of the final model on the validation dataset. By default, the F_beta score is the F1 score, which is the harmonic mean of the validation:precision and validation:recall metrics.</td>
<td>Maximize</td>
</tr>
<tr>
<td>validation:precision</td>
<td>The precision of the final model on the validation dataset. If you choose this metric as the objective, we recommend setting a target recall by setting the <code>binary_classifier_model_selection</code> hyperparameter to <code>precision_at_target_recall</code> and setting the value for the <code>target_recall</code> hyperparameter.</td>
<td>Maximize</td>
</tr>
<tr>
<td>validation:recall</td>
<td>The recall of the final model on the validation dataset. If you choose this metric as the objective, we recommend setting a target precision by setting the <code>binary_classifier_model_selection</code> hyperparameter to <code>recall_at_target_precision</code> and setting the value for the <code>target_precision</code> hyperparameter.</td>
<td>Maximize</td>
</tr>
</tbody>
</table>

**Tuning linear learner hyperparameters**

You can tune a linear learner model with the following hyperparameters.
Use built-in algorithms

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Parameter Type</th>
<th>Recommended Ranges</th>
</tr>
</thead>
<tbody>
<tr>
<td>wd</td>
<td>ContinuousParameterRanges</td>
<td>MinValue: 1e-7, MaxValue: 1</td>
</tr>
<tr>
<td>l1</td>
<td>ContinuousParameterRanges</td>
<td>MinValue: 1e-7, MaxValue: 1</td>
</tr>
<tr>
<td>learning_rate</td>
<td>ContinuousParameterRanges</td>
<td>MinValue: 1e-5, MaxValue: 1</td>
</tr>
<tr>
<td>mini_batch_size</td>
<td>IntegerParameterRanges</td>
<td>MinValue: 100, MaxValue: 5000</td>
</tr>
<tr>
<td>use_bias</td>
<td>CategoricalParameterRanges</td>
<td>[True, False]</td>
</tr>
<tr>
<td>positive_example_weight</td>
<td>ContinuousParameterRanges</td>
<td>MinValue: 1e-5, MaxValue: 1e5</td>
</tr>
</tbody>
</table>

Linear learner response formats

JSON response formats

All Amazon SageMaker built-in algorithms adhere to the common input inference format described in Common Data Formats - Inference. The following are the available output formats for the SageMaker linear learner algorithm.

**Binary Classification**

```javascript
let response = {
  "predictions": [
    {
      "score": 0.4,
      "predicted_label": 0
    }
  ]
};
```

**Multiclass Classification**

```javascript
let response = {
  "predictions": [
    {
      "score": [0.1, 0.2, 0.4, 0.3],
      "predicted_label": 2
    }
  ]
};
```

**Regression**

```javascript
let response = {
  "predictions": [
    {
      "score": 0.4
    }
  ]
};
```
JSONLINES response formats

Binary Classification

{"score": 0.4, "predicted_label": 0}

Multiclass Classification

{"score": [0.1, 0.2, 0.4, 0.3], "predicted_label": 2}

Regression

{"score": 0.4}

RECORDIO response formats

Binary Classification

[ Record = {  "features": [  "label": {  "score": {  "values": [0.4]  # float32  
  }  
  }  
  }  
}]

Multiclass Classification

[ Record = {  "features": [],  "label": {  "score": {  "values": [0.1, 0.2, 0.3, 0.4]  
  }  ,  "predicted_label": {  "values": [3]  
  }  ,  "uid": "abc123",  "metadata": "{created_at: '2017-06-03'}"
  }  
}]

Regression

[ Record = {  "features": [  "label": {  "score": {  
  }  
  }  
  }  
}]

Neural Topic Model (NTM) Algorithm

Amazon SageMaker NTM is an unsupervised learning algorithm that is used to organize a corpus of documents into *topics* that contain word groupings based on their statistical distribution. Documents that contain frequent occurrences of words such as “bike”, “car”, “train”, “mileage”, and “speed” are likely to share a topic on “transportation” for example. Topic modeling can be used to classify or summarize documents based on the topics detected or to retrieve information or recommend content based on topic similarities. The topics from documents that NTM learns are characterized as a *latent representation* because the topics are inferred from the observed word distributions in the corpus. The semantics of topics are usually inferred by examining the top ranking words they contain. Because the method is unsupervised, only the number of topics, not the topics themselves, are prespecified. In addition, the topics are not guaranteed to align with how a human might naturally categorize documents.

Topic modeling provides a way to visualize the contents of a large document corpus in terms of the learned topics. Documents relevant to each topic might be indexed or searched for based on their soft topic labels. The latent representations of documents might also be used to find similar documents in the topic space. You can also use the latent representations of documents that the topic model learns for input to another supervised algorithm such as a document classifier. Because the latent representations of documents are expected to capture the semantics of the underlying documents, algorithms based in part on these representations are expected to perform better than those based on lexical features alone.

Although you can use both the Amazon SageMaker NTM and LDA algorithms for topic modeling, they are distinct algorithms and can be expected to produce different results on the same input data.

For more information on the mathematics behind NTM, see *Neural Variational Inference for Text Processing*.

Topics
- Input/Output Interface for the NTM Algorithm (p. 748)
- EC2 Instance Recommendation for the NTM Algorithm (p. 749)
- NTM Sample Notebooks (p. 749)
- NTM Hyperparameters (p. 749)
- Tune an NTM Model (p. 752)
- NTM Response Formats (p. 753)

Input/Output Interface for the NTM Algorithm

Amazon SageMaker Neural Topic Model supports four data channels: train, validation, test, and auxiliary. The validation, test, and auxiliary data channels are optional. If you specify any of these optional channels, set the value of the `s3DataDistributionType` parameter for them to *FullyReplicated*. If you provide validation data, the loss on this data is logged at every epoch, and the model stops training as soon as it detects that the validation loss is not improving. If you don't provide validation data, the algorithm stops early based on the training data, but this can be less efficient. If you provide test data, the algorithm reports the test loss from the final model.

The train, validation, and test data channels for NTM support both *recordIO-wrapped-protobuf* (dense and sparse) and *CSV* file formats. For *CSV* format, each row must be represented densely with zero counts for words not present in the corresponding document, and have dimension equal to: (number of records) * (vocabulary size). You can use either File mode or Pipe mode to train models on...
data that is formatted as recordIO-wrapped-protobuf or as CSV. The auxiliary channel is used to supply a text file that contains vocabulary. By supplying the vocabulary file, users are able to see the top words for each of the topics printed in the log instead of their integer IDs. Having the vocabulary file also allows NTM to compute the Word Embedding Topic Coherence (WETC) scores, a new metric displayed in the log that captures similarity among the top words in each topic effectively. The ContentType for the auxiliary channel is text/plain, with each line containing a single word, in the order corresponding to the integer IDs provided in the data. The vocabulary file must be named vocab.txt and currently only UTF-8 encoding is supported.

For inference, text/csv, application/json, application/jsonlines, and application/x-recordio-protobuf content types are supported. Sparse data can also be passed for application/json and application/x-recordio-protobuf. NTM inference returns application/json or application/x-recordio-protobuf predictions, which include the topic_weights vector for each observation.

See the blog post and the companion notebook for more details on using the auxiliary channel and the WETC scores. For more information on how to compute the WETC score, see Coherence-Aware Neural Topic Modeling. We used the pairwise WETC described in this paper for the Amazon SageMaker Neural Topic Model.

For more information on input and output file formats, see NTM Response Formats (p. 753) for inference and the NTM Sample Notebooks (p. 749).

EC2 Instance Recommendation for the NTM Algorithm

NTM training supports both GPU and CPU instance types. We recommend GPU instances, but for certain workloads, CPU instances may result in lower training costs. CPU instances should be sufficient for inference.

NTM Sample Notebooks

For a sample notebook that uses the SageMaker NTM algorithm to uncover topics in documents from a synthetic data source where the topic distributions are known, see the Introduction to Basic Functionality of NTM. For instructions how to create and access Jupyter notebook instances that you can use to run the example in SageMaker, see Use Amazon SageMaker Notebook Instances (p. 124). Once you have created a notebook instance and opened it, select the SageMaker Examples tab to see a list of all the SageMaker samples. The topic modeling example notebooks using the NTM algorithms are located in the Introduction to Amazon algorithms section. To open a notebook, click on its Use tab and select Create copy.

NTM Hyperparameters

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>feature_dim</td>
<td>The vocabulary size of the dataset.</td>
</tr>
<tr>
<td></td>
<td>Required</td>
</tr>
<tr>
<td></td>
<td>Valid values: Positive integer (min: 1, max: 1,000,000)</td>
</tr>
<tr>
<td>num_topics</td>
<td>The number of required topics.</td>
</tr>
<tr>
<td></td>
<td>Required</td>
</tr>
<tr>
<td></td>
<td>Valid values: Positive integer (min: 2, max: 1000)</td>
</tr>
<tr>
<td>batch_norm</td>
<td>Whether to use batch normalization during training.</td>
</tr>
<tr>
<td></td>
<td>Optional</td>
</tr>
<tr>
<td>Parameter Name</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td></td>
<td>Valid values: true or false</td>
</tr>
<tr>
<td></td>
<td>Default value: false</td>
</tr>
<tr>
<td>clip_gradient</td>
<td>The maximum magnitude for each gradient component.</td>
</tr>
<tr>
<td></td>
<td>Optional</td>
</tr>
<tr>
<td></td>
<td>Valid values: Float (min: 1e-3)</td>
</tr>
<tr>
<td></td>
<td>Default value: Infinity</td>
</tr>
<tr>
<td>encoder_layers</td>
<td>The number of layers in the encoder and the output size of each layer.</td>
</tr>
<tr>
<td></td>
<td>When set to auto, the algorithm uses two layers of sizes 3 x num_topics and 2 x num_topics respectively.</td>
</tr>
<tr>
<td></td>
<td>Optional</td>
</tr>
<tr>
<td></td>
<td>Valid values: Comma-separated list of positive integers or auto</td>
</tr>
<tr>
<td></td>
<td>Default value: auto</td>
</tr>
<tr>
<td>encoder_layers_activation</td>
<td>The activation function to use in the encoder layers.</td>
</tr>
<tr>
<td></td>
<td>Optional</td>
</tr>
<tr>
<td></td>
<td>Valid values:</td>
</tr>
<tr>
<td></td>
<td>• sigmoid: Sigmoid function</td>
</tr>
<tr>
<td></td>
<td>• tanh: Hyperbolic tangent</td>
</tr>
<tr>
<td></td>
<td>• relu: Rectified linear unit</td>
</tr>
<tr>
<td></td>
<td>Default value: sigmoid</td>
</tr>
<tr>
<td>epochs</td>
<td>The maximum number of passes over the training data.</td>
</tr>
<tr>
<td></td>
<td>Optional</td>
</tr>
<tr>
<td></td>
<td>Valid values: Positive integer (min: 1)</td>
</tr>
<tr>
<td></td>
<td>Default value: 50</td>
</tr>
<tr>
<td>learning_rate</td>
<td>The learning rate for the optimizer.</td>
</tr>
<tr>
<td></td>
<td>Optional</td>
</tr>
<tr>
<td></td>
<td>Valid values: Float (min: 1e-6, max: 1.0)</td>
</tr>
<tr>
<td></td>
<td>Default value: 0.001</td>
</tr>
<tr>
<td>mini_batch_size</td>
<td>The number of examples in each mini batch.</td>
</tr>
<tr>
<td></td>
<td>Optional</td>
</tr>
<tr>
<td></td>
<td>Valid values: Positive integer (min: 1, max: 10000)</td>
</tr>
<tr>
<td></td>
<td>Default value: 256</td>
</tr>
<tr>
<td>Parameter Name</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>num_patience_epochs</td>
<td>The number of successive epochs over which early stopping criterion is evaluated. Early stopping is triggered when the change in the loss function drops below the specified tolerance within the last num_patience_epochs number of epochs. To disable early stopping, set num_patience_epochs to a value larger than epochs.</td>
</tr>
<tr>
<td></td>
<td>Optional</td>
</tr>
<tr>
<td></td>
<td>Valid values: Positive integer (min: 1)</td>
</tr>
<tr>
<td></td>
<td>Default value: 3</td>
</tr>
<tr>
<td>optimizer</td>
<td>The optimizer to use for training.</td>
</tr>
<tr>
<td></td>
<td>Optional</td>
</tr>
<tr>
<td></td>
<td>Valid values:                                                                                                                                  sgd: Stochastic gradient descent \nadam: Adaptive momentum estimation \nadagrad: Adaptive gradient algorithm \nadadelta: An adaptive learning rate algorithm \nrmsprop: Root mean square propagation</td>
</tr>
<tr>
<td></td>
<td>Default value: adadelta</td>
</tr>
<tr>
<td>rescale_gradient</td>
<td>The rescale factor for gradient.</td>
</tr>
<tr>
<td></td>
<td>Optional</td>
</tr>
<tr>
<td></td>
<td>Valid values: float (min: 1e-3, max: 1.0)</td>
</tr>
<tr>
<td></td>
<td>Default value: 1.0</td>
</tr>
<tr>
<td>sub_sample</td>
<td>The fraction of the training data to sample for training per epoch.</td>
</tr>
<tr>
<td></td>
<td>Optional</td>
</tr>
<tr>
<td></td>
<td>Valid values: Float (min: 0.0, max: 1.0)</td>
</tr>
<tr>
<td></td>
<td>Default value: 1.0</td>
</tr>
<tr>
<td>tolerance</td>
<td>The maximum relative change in the loss function. Early stopping is triggered when change in the loss function drops below this value within the last num_patience_epochs number of epochs.</td>
</tr>
<tr>
<td></td>
<td>Optional</td>
</tr>
<tr>
<td></td>
<td>Valid values: Float (min: 1e-6, max: 0.1)</td>
</tr>
<tr>
<td></td>
<td>Default value: 0.001</td>
</tr>
</tbody>
</table>
Tune an NTM Model

*Automatic model tuning*, also known as hyperparameter tuning, finds the best version of a model by running many jobs that test a range of hyperparameters on your dataset. You choose the tunable hyperparameters, a range of values for each, and an objective metric. You choose the objective metric from the metrics that the algorithm computes. Automatic model tuning searches the hyperparameters chosen to find the combination of values that result in the model that optimizes the objective metric.

Amazon SageMaker NTM is an unsupervised learning algorithm that learns latent representations of large collections of discrete data, such as a corpus of documents. Latent representations use inferred variables that are not directly measured to model the observations in a dataset. Automatic model tuning on NTM helps you find the model that minimizes loss over the training or validation data. *Training loss* measures how well the model fits the training data. *Validation loss* measures how well the model can generalize to data that it is not trained on. Low training loss indicates that a model is a good fit to the training data. Low validation loss indicates that a model has not overfit the training data and so should be able to model documents successfully on which is has not been trained. Usually, it’s preferable to have both losses be small. However, minimizing training loss too much might result in overfitting and increase validation loss, which would reduce the generality of the model.

For more information about model tuning, see Perform Automatic Model Tuning (p. 1023).

**Metrics Computed by the NTM Algorithm**

The NTM algorithm reports a single metric that is computed during training: *validation:total_loss*. The total loss is the sum of the reconstruction loss and Kullback-Leibler divergence. When tuning hyperparameter values, choose this metric as the objective.

<table>
<thead>
<tr>
<th>Metric Name</th>
<th>Description</th>
<th>Optimization Direction</th>
</tr>
</thead>
<tbody>
<tr>
<td>validation:total_loss</td>
<td>Total Loss on validation set</td>
<td>Minimize</td>
</tr>
</tbody>
</table>

**Tunable NTM Hyperparameters**

You can tune the following hyperparameters for the NTM algorithm. Usually setting low *mini_batch_size* and small *learning_rate* values results in lower validation losses, although it might take longer to train. Low validation losses don’t necessarily produce more coherent topics as interpreted by humans. The effect of other hyperparameters on training and validation loss can vary from dataset to dataset. To see which values are compatible, see *NTM Hyperparameters* (p. 749).

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Parameter Type</th>
<th>Recommended Ranges</th>
</tr>
</thead>
<tbody>
<tr>
<td>encoder_layers_activation</td>
<td>CategoricalParameterRange</td>
<td>['sigmoid', 'tanh', 'relu']</td>
</tr>
<tr>
<td>learning_rate</td>
<td>ContinuousParameterRange</td>
<td>MinValue: 1e-4, MaxValue: 0.1</td>
</tr>
</tbody>
</table>
Use built-in algorithms

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Parameter Type</th>
<th>Recommended Ranges</th>
</tr>
</thead>
<tbody>
<tr>
<td>mini_batch_size</td>
<td>IntegerParameterRanges</td>
<td>MinValue: 16, MaxValue: 2048</td>
</tr>
<tr>
<td>optimizer</td>
<td>CategoricalParameterRanges</td>
<td>['sgd', 'adam', 'adadelta']</td>
</tr>
<tr>
<td>rescale_gradient</td>
<td>ContinuousParameterRange</td>
<td>MinValue: 0.1, MaxValue: 1.0</td>
</tr>
<tr>
<td>weight_decay</td>
<td>ContinuousParameterRange</td>
<td>MinValue: 0.0, MaxValue: 1.0</td>
</tr>
</tbody>
</table>

NTM Response Formats

All Amazon SageMaker built-in algorithms adhere to the common input inference format described in Common Data Formats - Inference. This topic contains a list of the available output formats for the SageMaker NTM algorithm.

JSON Response Format

```json
{
   "predictions": [
       {
           "topic_weights": [0.02, 0.1, 0,...],
           "topic_weights": [0.25, 0.067, 0,...]
        }
   ]
}
```

JSONLINES Response Format

```json
{"topic_weights": [0.02, 0.1, 0,...]}
{"topic_weights": [0.25, 0.067, 0,...]}
```

RECORDIO Response Format

```json
[ Record = {
    features = {},
    label = {
        'topic_weights': {
            keys: [],
            values: [0.25, 0.067, 0, ...]  # float32
        }
    },
    Record = {
    features = {},
    label = {
        'topic_weights': {
            keys: [],
            values: [0.25, 0.067, 0, ...]  # float32
        }
    }
}
]```
Object2Vec Algorithm

The Amazon SageMaker Object2Vec algorithm is a general-purpose neural embedding algorithm that is highly customizable. It can learn low-dimensional dense embeddings of high-dimensional objects. The embeddings are learned in a way that preserves the semantics of the relationship between pairs of objects in the original space in the embedding space. You can use the learned embeddings to efficiently compute nearest neighbors of objects and to visualize natural clusters of related objects in low-dimensional space, for example. You can also use the embeddings as features of the corresponding objects in downstream supervised tasks, such as classification or regression.

Object2Vec generalizes the well-known Word2Vec embedding technique for words that is optimized in the SageMaker BlazingText algorithm (p. 656). For a blog post that discusses how to apply Object2Vec to some practical use cases, see Introduction to Amazon SageMaker Object2Vec.

Topics

- I/O Interface for the Object2Vec Algorithm (p. 754)
- EC2 Instance Recommendation for the Object2Vec Algorithm (p. 755)
- Object2Vec Sample Notebooks (p. 755)
- How Object2Vec Works (p. 755)
- Object2Vec Hyperparameters (p. 757)
- Tune an Object2Vec Model (p. 765)
- Data Formats for Object2Vec Training (p. 767)
- Data Formats for Object2Vec Inference (p. 767)
- Encoder Embeddings for Object2Vec (p. 769)

I/O Interface for the Object2Vec Algorithm

You can use Object2Vec on many input data types, including the following examples.

<table>
<thead>
<tr>
<th>Input Data Type</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sentence-sentence pairs</td>
<td>&quot;A soccer game with multiple males playing.&quot; and &quot;Some men are playing a sport.&quot;</td>
</tr>
<tr>
<td>Labels-sequence pairs</td>
<td>The genre tags of the movie &quot;Titanic,&quot; such as &quot;Romance&quot; and &quot;Drama,&quot; and its short description: &quot;James Cameron's Titanic is an epic, action-packed romance set against the ill-fated maiden voyage of the R.M.S. Titanic. She was the most luxurious liner of her era, a ship of dreams, which ultimately carried over 1,500 people to their death in the ice cold waters of the North Atlantic in the early hours of April 15, 1912.&quot;</td>
</tr>
<tr>
<td>Customer-customer pairs</td>
<td>The customer ID of Jane and customer ID of Jackie.</td>
</tr>
<tr>
<td>Product-product pairs</td>
<td>The product ID of football and product ID of basketball.</td>
</tr>
<tr>
<td>Item review user-item pairs</td>
<td>A user's ID and the items she has bought, such as apple, pear, and orange.</td>
</tr>
</tbody>
</table>

To transform the input data into the supported formats, you must preprocess it. Currently, Object2Vec natively supports two types of input:

- A discrete token, which is represented as a list of a single integer-id. For example, [10].
A sequence of discrete tokens, which is represented as a list of integer-ids. For example, \[0,12,10,13\].

The object in each pair can be asymmetric. For example, the pairs can be (token, sequence) or (token, token) or (sequence, sequence). For token inputs, the algorithm supports simple embeddings as compatible encoders. For sequences of token vectors, the algorithm supports the following as encoders:

- Average-pooled embeddings
- Hierarchical convolutional neural networks (CNNs),
- Multi-layered bidirectional long short-term memory (BiLSTMs)

The input label for each pair can be one of the following:

- A categorical label that expresses the relationship between the objects in the pair
- A score that expresses the strength of the similarity between the two objects

For categorical labels used in classification, the algorithm supports the cross-entropy loss function. For ratings/score-based labels used in regression, the algorithm supports the mean squared error (MSE) loss function. Specify these loss functions with the `output_layer` hyperparameter when you create the model training job.

**EC2 Instance Recommendation for the Object2Vec Algorithm**

The type of Amazon Elastic Compute Cloud (Amazon EC2) instance that you use depends on whether you are training or running inferences.

**Instance Recommendation for Training**

When training a model using the Object2Vec algorithm on a CPU, start with an ml.m5.2xlarge instance. For training on a GPU, start with an ml.p2.xlarge instance. If the training takes too long on this instance, you can use a larger instance, such as an ml.m5.4xlarge or an ml.m5.12xlarge instance. Currently, the Object2Vec algorithm can train only on a single machine. However, it does offer support for multiple GPUs.

**Instance Recommendation for Inference**

For inference with a trained Object2Vec model that has a deep neural network, we recommend using ml.p3.2xlarge GPU instance. Due to GPU memory scarcity, the `INFERANCE_PREFERRED_MODE` environment variable can be specified to optimize on whether the section called "GPU optimization: Classification or Regression" (p. 767) or the section called "GPU optimization: Encoder Embeddings" (p. 769) inference network is loaded into GPU.

**Object2Vec Sample Notebooks**

- Using Object2Vec to Encode Sentences into Fixed Length Embeddings
- Using Object2Vec to learn document embeddings

**Note**

To run the notebooks on a notebook instance, see Example Notebooks (p. 134). To run the notebooks on Studio, see Create or Open an Amazon SageMaker Studio Notebook (p. 87).

**How Object2Vec Works**

When using the Amazon SageMaker Object2Vec algorithm, you follow the standard workflow: process the data, train the model, and produce inferences.
Topics

- Step 1: Process Data (p. 756)
- Step 2: Train a Model (p. 756)
- Step 3: Produce Inferences (p. 757)

Step 1: Process Data

During preprocessing, convert the data to the JSON Lines text file format specified in Data Formats for Object2Vec Training (p. 767). To get the highest accuracy during training, also randomly shuffle the data before feeding it into the model. How you generate random permutations depends on the language. For python, you could use `np.random.shuffle`; for Unix, `shuf`.

Step 2: Train a Model

The SageMaker Object2Vec algorithm has the following main components:

- **Two input channels** – The input channels take a pair of objects of the same or different types as inputs, and pass them to independent and customizable encoders.

- **Two encoders** – The two encoders, enc0 and enc1, convert each object into a fixed-length embedding vector. The encoded embeddings of the objects in the pair are then passed into a comparator.

- **A comparator** – The comparator compares the embeddings in different ways and outputs scores that indicate the strength of the relationship between the paired objects. In the output score for a sentence pair. For example, 1 indicates a strong relationship between a sentence pair, and 0 represents a weak relationship.

During training, the algorithm accepts pairs of objects and their relationship labels or scores as inputs. The objects in each pair can be of different types, as described earlier. If the inputs to both encoders are composed of the same token-level units, you can use a shared token embedding layer by setting the `tied_token_embedding_weight` hyperparameter to True when you create the training job. This is possible, for example, when comparing sentences that both have word token-level units. To generate negative samples at a specified rate, set the `negative_sampling_rate` hyperparameter to the desired ratio of negative to positive samples. This hyperparameter expedites learning how to discriminate between the positive samples observed in the training data and the negative samples that are not likely to be observed.

Pairs of objects are passed through independent, customizable encoders that are compatible with the input types of corresponding objects. The encoders convert each object in a pair into a fixed-length embedding vector of equal length. The pair of vectors are passed to a comparator operator, which assembles the vectors into a single vector using the value specified in the `comparator_list` hyperparameter. The assembled vector then passes through a multilayer perceptron (MLP) layer, which produces an output that the loss function compares with the labels that you provided. This comparison evaluates the strength of the relationship between the objects in the pair as predicted by the model. The following figure shows this workflow.
Step 3: Produce Inferences

After the model is trained, you can use the trained encoder to preprocess input objects or to perform two types of inference:

- To convert singleton input objects into fixed-length embeddings using the corresponding encoder
- To predict the relationship label or score between a pair of input objects

The inference server automatically figures out which of the types is requested based on the input data. To get the embeddings as output, provide only one input. To predict the relationship label or score, provide both inputs in the pair.

Object2Vec Hyperparameters

In the CreateTrainingJob request, you specify the training algorithm. You can also specify algorithm-specific hyperparameters as string-to-string maps. The following table lists the hyperparameters for the Object2Vec training algorithm.

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>enc0_max_seq_len</td>
<td>The maximum sequence length for the enc0 encoder.</td>
</tr>
<tr>
<td></td>
<td><strong>Required</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: $1 \leq \text{integer} \leq 5000$</td>
</tr>
<tr>
<td>enc0_vocab_size</td>
<td>The vocabulary size of enc0 tokens.</td>
</tr>
<tr>
<td></td>
<td><strong>Required</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: $2 \leq \text{integer} \leq 3000000$</td>
</tr>
<tr>
<td>Parameter Name</td>
<td>Description</td>
</tr>
<tr>
<td>------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>bucket_width</td>
<td>The allowed difference between data sequence length when bucketing is enabled. To enable bucketing, specify a non-zero value for this parameter.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: $0 \leq \text{integer} \leq 100$</td>
</tr>
<tr>
<td></td>
<td>Default value: 0 (no bucketing)</td>
</tr>
<tr>
<td>comparator_list</td>
<td>A list used to customize the way in which two embeddings are compared. The Object2Vec comparator operator layer takes the encodings from both encoders as inputs and outputs a single vector. This vector is a concatenation of subvectors. The string values passed to the comparator_list and the order in which they are passed determine how these subvectors are assembled. For example, if comparator_list=&quot;hadamard, concat&quot;, then the comparator operator constructs the vector by concatenating the Hadamard product of two encodings and the concatenation of two encodings. If, on the other hand, comparator_list=&quot;hadamard&quot;, then the comparator operator constructs the vector as the hadamard product of only two encodings.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: A string that contains any combination of the names of the three binary operators: hadamard, concat, or abs_diff. The Object2Vec algorithm currently requires that the two vector encodings have the same dimension. These operators produce the subvectors as follows:</td>
</tr>
<tr>
<td></td>
<td>• hadamard: Constructs a vector as the Hadamard (element-wise) product of two encodings.</td>
</tr>
<tr>
<td></td>
<td>• concat: Constructs a vector as the concatenation of two encodings.</td>
</tr>
<tr>
<td></td>
<td>• abs_diff: Constructs a vector as the absolute difference between two encodings.</td>
</tr>
<tr>
<td></td>
<td>Default value: &quot;hadamard, concat, abs_diff&quot;</td>
</tr>
<tr>
<td>dropout</td>
<td>The dropout probability for network layers. Dropout is a form of regularization used in neural networks that reduces overfitting by trimming codependent neurons.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: $0.0 \leq \text{float} \leq 1.0$</td>
</tr>
<tr>
<td></td>
<td>Default value: 0.0</td>
</tr>
<tr>
<td>Parameter Name</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------</td>
<td>-------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>early_stopping_patience</td>
<td>The number of consecutive epochs without improvement allowed before early stopping is applied. Improvement is defined by with the early_stopping_tolerance hyperparameter.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: $1 \leq \text{integer} \leq 5$</td>
</tr>
<tr>
<td></td>
<td>Default value: 3</td>
</tr>
<tr>
<td>early_stopping_tolerance</td>
<td>The reduction in the loss function that an algorithm must achieve between consecutive epochs to avoid early stopping after the number of consecutive epochs specified in the early_stopping_patience hyperparameter concludes.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: $0.000001 \leq \text{float} \leq 0.1$</td>
</tr>
<tr>
<td></td>
<td>Default value: 0.01</td>
</tr>
<tr>
<td>enc_dim</td>
<td>The dimension of the output of the embedding layer.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: $4 \leq \text{integer} \leq 10000$</td>
</tr>
<tr>
<td></td>
<td>Default value: 4096</td>
</tr>
<tr>
<td>enc0_network</td>
<td>The network model for the enc0 encoder.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: hcnn, bilstm, or pooled_embedding</td>
</tr>
<tr>
<td></td>
<td>• hcnn: A hierarchical convolutional neural network.</td>
</tr>
<tr>
<td></td>
<td>• bilstm: A bidirectional long short-term memory network (biLSTM), in which the signal propagates backward and forward in time. This is an appropriate recurrent neural network (RNN) architecture for sequential learning tasks.</td>
</tr>
<tr>
<td></td>
<td>• pooled_embedding: Averages the embeddings of all of the tokens in the input.</td>
</tr>
<tr>
<td></td>
<td>Default value: hcnn</td>
</tr>
<tr>
<td>enc0_cnn_filter_width</td>
<td>The filter width of the convolutional neural network (CNN) enc0 encoder.</td>
</tr>
<tr>
<td></td>
<td><strong>Conditional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: $1 \leq \text{integer} \leq 9$</td>
</tr>
<tr>
<td></td>
<td>Default value: 3</td>
</tr>
<tr>
<td>Parameter Name</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>enc0_freeze_pretrained_embedding</code></td>
<td>Whether to freeze enc0 pretrained embedding weights.</td>
</tr>
<tr>
<td></td>
<td><strong>Conditional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: True or False</td>
</tr>
<tr>
<td></td>
<td>Default value: True</td>
</tr>
<tr>
<td><code>enc0_layers</code></td>
<td>The number of layers in the enc0 encoder.</td>
</tr>
<tr>
<td></td>
<td><strong>Conditional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: auto or $1 \leq \text{integer} \leq 4$</td>
</tr>
<tr>
<td></td>
<td>• For hcnn, auto means 4.</td>
</tr>
<tr>
<td></td>
<td>• For bilstm, auto means 1.</td>
</tr>
<tr>
<td></td>
<td>• For pooled_embedding, auto ignores the number of layers.</td>
</tr>
<tr>
<td></td>
<td>Default value: auto</td>
</tr>
<tr>
<td><code>enc0_pretrained_embedding_file</code></td>
<td>The filename of the pretrained enc0 token embedding file in the auxiliary data channel.</td>
</tr>
<tr>
<td></td>
<td><strong>Conditional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: String with alphanumeric characters, underscore, or period. [A-Za-z0-9._]</td>
</tr>
<tr>
<td></td>
<td>Default value: &quot;&quot; (empty string)</td>
</tr>
<tr>
<td><code>enc0_token_embedding_dim</code></td>
<td>The output dimension of the enc0 token embedding layer.</td>
</tr>
<tr>
<td></td>
<td><strong>Conditional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: $2 \leq \text{integer} \leq 1000$</td>
</tr>
<tr>
<td></td>
<td>Default value: 300</td>
</tr>
<tr>
<td><code>enc0_vocab_file</code></td>
<td>The vocabulary file for mapping pretrained enc0 token embedding vectors to numerical vocabulary IDs.</td>
</tr>
<tr>
<td></td>
<td><strong>Conditional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: String with alphanumeric characters, underscore, or period. [A-Za-z0-9._]</td>
</tr>
<tr>
<td></td>
<td>Default value: &quot;&quot; (empty string)</td>
</tr>
<tr>
<td>Parameter Name</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| enc1_network          | The network model for the enc1 encoder. If you want the enc1 encoder to use the same network model as enc0, including the hyperparameter values, set the value to enc0. **Note** Even when the enc0 and enc1 encoder networks have symmetric architectures, you can't shared parameter values for these networks. Optional Valid values: enc0, hcnn, bilstm, or pooled_embedding  
  • **enc0**: The network model for the enc0 encoder.  
  • **hcnn**: A hierarchical convolutional neural network.  
  • **bilstm**: A bidirectional LSTM, in which the signal propagates backward and forward in time. This is an appropriate recurrent neural network (RNN) architecture for sequential learning tasks.  
  • **pooled_embedding**: The averages of the embeddings of all of the tokens in the input.  
  
  Default value: enc0 |
| enc1_cnn_filter_width | The filter width of the CNN enc1 encoder. **Conditional**  
  Valid values: 1 ≤ integer ≤ 9  
  Default value: 3 |
| enc1_freeze_pretrained_embedding | Whether to freeze enc1 pretrained embedding weights. **Conditional**  
  Valid values: True or False  
  Default value: True |
| enc1_layers           | The number of layers in the enc1 encoder. **Conditional**  
  Valid values: auto or 1 ≤ integer ≤ 4  
  • For hcnn, auto means 4.  
  • For bilstm, auto means 1.  
  • For pooled_embedding, auto ignores the number of layers.  
  Default value: auto |
| enc1_max_seq_len      | The maximum sequence length for the enc1 encoder. **Conditional**  
  Valid values: 1 ≤ integer ≤ 5000 |
<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>enc1_pretrained_embedding_file</td>
<td>The name of the enc1 pretrained token embedding file in the auxiliary data channel.</td>
</tr>
<tr>
<td></td>
<td><strong>Conditional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: String with alphanumeric characters, underscore, or period.</td>
</tr>
<tr>
<td></td>
<td>[A-Za-z0-9._]</td>
</tr>
<tr>
<td></td>
<td>Default value: &quot;&quot; (empty string)</td>
</tr>
<tr>
<td>enc1_token_embedding_dim</td>
<td>The output dimension of the enc1 token embedding layer.</td>
</tr>
<tr>
<td></td>
<td><strong>Conditional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: 2 ≤ integer ≤ 1000</td>
</tr>
<tr>
<td></td>
<td>Default value: 300</td>
</tr>
<tr>
<td>enc1_vocab_file</td>
<td>The vocabulary file for mapping pretrained enc1 token embeddings to vocabulary IDs.</td>
</tr>
<tr>
<td></td>
<td><strong>Conditional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: String with alphanumeric characters, underscore, or period.</td>
</tr>
<tr>
<td></td>
<td>[A-Za-z0-9._]</td>
</tr>
<tr>
<td></td>
<td>Default value: &quot;&quot; (empty string)</td>
</tr>
<tr>
<td>enc1_vocab_size</td>
<td>The vocabulary size of enc0 tokens.</td>
</tr>
<tr>
<td></td>
<td><strong>Conditional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: 2 ≤ integer ≤ 3000000</td>
</tr>
<tr>
<td>epochs</td>
<td>The number of epochs to run for training.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: 1 ≤ integer ≤ 100</td>
</tr>
<tr>
<td></td>
<td>Default value: 30</td>
</tr>
<tr>
<td>learning_rate</td>
<td>The learning rate for training.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: 1.0E-6 ≤ float ≤ 1.0</td>
</tr>
<tr>
<td></td>
<td>Default value: 0.0004</td>
</tr>
<tr>
<td>mini_batch_size</td>
<td>The batch size that the dataset is split into for an optimizer during training.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: 1 ≤ integer ≤ 10000</td>
</tr>
<tr>
<td></td>
<td>Default value: 32</td>
</tr>
<tr>
<td>Parameter Name</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>mlp_activation</td>
<td>The type of activation function for the multilayer perceptron (MLP) layer.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: tanh, relu, or linear</td>
</tr>
<tr>
<td></td>
<td>• tanh: Hyperbolic tangent</td>
</tr>
<tr>
<td></td>
<td>• relu: Rectified linear unit (ReLU)</td>
</tr>
<tr>
<td></td>
<td>• linear: Linear function</td>
</tr>
<tr>
<td></td>
<td>Default value: linear</td>
</tr>
<tr>
<td>mlp_dim</td>
<td>The dimension of the output from MLP layers.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: 2 ≤ integer ≤ 10000</td>
</tr>
<tr>
<td></td>
<td>Default value: 512</td>
</tr>
<tr>
<td>mlp_layers</td>
<td>The number of MLP layers in the network.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: 0 ≤ integer ≤ 10</td>
</tr>
<tr>
<td></td>
<td>Default value: 2</td>
</tr>
<tr>
<td>negative_sampling_rate</td>
<td>The ratio of negative samples, generated to assist in training the algorithm, to positive samples that are provided by users. Negative samples represent data that is unlikely to occur in reality and are labelled negatively for training. They facilitate training a model to discriminate between the positive samples observed and the negative samples that are not. To specify the ratio of negative samples to positive samples used for training, set the value to a positive integer. For example, if you train the algorithm on input data in which all of the samples are positive and set negative_sampling_rate to 2, the Object2Vec algorithm internally generates two negative samples per positive sample. If you don't want to generate or use negative samples during training, set the value to 0.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: 0 ≤ integer</td>
</tr>
<tr>
<td></td>
<td>Default value: 0 (off)</td>
</tr>
<tr>
<td>num_classes</td>
<td>The number of classes for classification training. Amazon SageMaker ignores this hyperparameter for regression problems.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: 2 ≤ integer ≤ 30</td>
</tr>
<tr>
<td></td>
<td>Default value: 2</td>
</tr>
<tr>
<td>Parameter Name</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>optimizer</td>
<td>The optimizer type.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: adadelta, adagrad, adam, sgd, or rmsprop.</td>
</tr>
<tr>
<td></td>
<td>• adadelta: A per-dimension learning rate method for gradient descent</td>
</tr>
<tr>
<td></td>
<td>• adagrad: The adaptive gradient algorithm</td>
</tr>
<tr>
<td></td>
<td>• adam: The adaptive moment estimation algorithm</td>
</tr>
<tr>
<td></td>
<td>• sgd: Stochastic gradient descent</td>
</tr>
<tr>
<td></td>
<td>• rmsprop: Root mean square propagation</td>
</tr>
<tr>
<td></td>
<td>Default value: adam</td>
</tr>
<tr>
<td>output_layer</td>
<td>The type of output layer where you specify that the task is regression or classification.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: softmax or mean_squared_error</td>
</tr>
<tr>
<td></td>
<td>• softmax: The Softmax function used for classification.</td>
</tr>
<tr>
<td></td>
<td>• mean_squared_error: The MSE used for regression.</td>
</tr>
<tr>
<td></td>
<td>Default value: softmax</td>
</tr>
<tr>
<td>tied_token_embedding_weight</td>
<td>Whether to use a shared embedding layer for both encoders. If the inputs to both encoders use the same token-level units, use a shared token embedding layer. For example, for a collection of documents, if one encoder encodes sentences and another encodes whole documents, you can use a shared token embedding layer. That's because both sentences and documents are composed of word tokens from the same vocabulary.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: True or False</td>
</tr>
<tr>
<td></td>
<td>Default value: False</td>
</tr>
</tbody>
</table>
**Parameter Name** | **Description**
---|---
**token_embedding_storage_type** | The mode of gradient update used during training: when the **dense** mode is used, the optimizer calculates the full gradient matrix for the token embedding layer even if most rows of the gradient are zero-valued. When **sparse** mode is used, the optimizer only stores rows of the gradient that are actually being used in the mini-batch. If you want the algorithm to perform lazy gradient updates, which calculate the gradients only in the non-zero rows and which speed up training, specify **row_sparse**. Setting the value to **row_sparse** constrains the values available for other hyperparameters, as follows:

- The optimizer hyperparameter must be set to **adam**, **adagrad**, or **sgd**. Otherwise, the algorithm throws a **CustomerValueError**.
- The algorithm automatically disables bucketing, setting the **bucket_width** hyperparameter to 0.

Optional
Valid values: **dense** or **row_sparse**
Default value: **dense**

**weight_decay** | The weight decay parameter used for optimization.

Optional
Valid values: 0 ≤ float ≤ 10000
Default value: 0 (no decay)

---

**Tune an Object2Vec Model**

*Automatic model tuning*, also known as hyperparameter tuning, finds the best version of a model by running many jobs that test a range of hyperparameters on your dataset. You choose the tunable hyperparameters, a range of values for each, and an objective metric. For the objective metric, you use one of the metrics that the algorithm computes. Automatic model tuning searches the chosen hyperparameters to find the combination of values that result in the model that optimizes the objective metric.

For more information about model tuning, see Perform Automatic Model Tuning (p. 1023).

**Metrics Computed by the Object2Vec Algorithm**

The Object2Vec algorithm has both classification and regression metrics. The **output_layer_type** determines which metric you can use for automatic model tuning.

**Regressor Metrics Computed by the Object2Vec Algorithm**

The algorithm reports a mean squared error regressor metric, which is computed during testing and validation. When tuning the model for regression tasks, choose this metric as the objective.

<table>
<thead>
<tr>
<th>Metric Name</th>
<th>Description</th>
<th>Optimization Direction</th>
</tr>
</thead>
<tbody>
<tr>
<td>test:mean_squared_error_root</td>
<td>Mean Square Error</td>
<td>Minimize</td>
</tr>
</tbody>
</table>
Classical Metrics Computed by the Object2Vec Algorithm

The Object2Vec algorithm reports accuracy and cross-entropy classification metrics, which are computed during test and validation. When tuning the model for classification tasks, choose one of these as the objective.

<table>
<thead>
<tr>
<th>Metric Name</th>
<th>Description</th>
<th>Optimization Direction</th>
</tr>
</thead>
<tbody>
<tr>
<td>validation:mean_squared_error</td>
<td>Root Mean Square Error</td>
<td>Minimize</td>
</tr>
<tr>
<td>test:accuracy</td>
<td>Accuracy</td>
<td>Maximize</td>
</tr>
<tr>
<td>test:cross_entropy</td>
<td>Cross-entropy</td>
<td>Minimize</td>
</tr>
<tr>
<td>validation:accuracy</td>
<td>Accuracy</td>
<td>Maximize</td>
</tr>
<tr>
<td>validation:cross_entropy</td>
<td>Cross-entropy</td>
<td>Minimize</td>
</tr>
</tbody>
</table>

Tunable Object2Vec Hyperparameters

You can tune the following hyperparameters for the Object2Vec algorithm.

<table>
<thead>
<tr>
<th>Hyperparameter Name</th>
<th>Hyperparameter Type</th>
<th>Recommended Ranges and Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>dropout</td>
<td>ContinuousParameterRange</td>
<td>MinValue: 0.0, MaxValue: 1.0</td>
</tr>
<tr>
<td>early_stopping_patience</td>
<td>IntegerParameterRange</td>
<td>MinValue: 1, MaxValue: 5</td>
</tr>
<tr>
<td>early_stopping_tolerance</td>
<td>ContinuousParameterRange</td>
<td>MinValue: 0.001, MaxValue: 0.1</td>
</tr>
<tr>
<td>enc_dim</td>
<td>IntegerParameterRange</td>
<td>MinValue: 4, MaxValue: 4096</td>
</tr>
<tr>
<td>enc0_cnn_filter_width</td>
<td>IntegerParameterRange</td>
<td>MinValue: 1, MaxValue: 5</td>
</tr>
<tr>
<td>enc0_layers</td>
<td>IntegerParameterRange</td>
<td>MinValue: 1, MaxValue: 4</td>
</tr>
<tr>
<td>enc0_token_embedding_dim</td>
<td>IntegerParameterRange</td>
<td>MinValue: 5, MaxValue: 300</td>
</tr>
<tr>
<td>enc1_cnn_filter_width</td>
<td>IntegerParameterRange</td>
<td>MinValue: 1, MaxValue: 5</td>
</tr>
<tr>
<td>enc1_layers</td>
<td>IntegerParameterRange</td>
<td>MinValue: 1, MaxValue: 4</td>
</tr>
<tr>
<td>enc1_token_embedding_dim</td>
<td>IntegerParameterRange</td>
<td>MinValue: 5, MaxValue: 300</td>
</tr>
<tr>
<td>Hyperparameter Name</td>
<td>Hyperparameter Type</td>
<td>Recommended Ranges and Values</td>
</tr>
<tr>
<td>---------------------</td>
<td>--------------------------------</td>
<td>-----------------------------------------------</td>
</tr>
<tr>
<td>epochs</td>
<td>IntegerParameterRange</td>
<td>MinValue: 4, MaxValue: 20</td>
</tr>
<tr>
<td>learning_rate</td>
<td>ContinuousParameterRange</td>
<td>MinValue: 1e-6, MaxValue: 1.0</td>
</tr>
<tr>
<td>mini_batch_size</td>
<td>IntegerParameterRange</td>
<td>MinValue: 1, MaxValue: 8192</td>
</tr>
<tr>
<td>mlp_activation</td>
<td>CategoricalParameterRanges</td>
<td>[tanh, relu, linear]</td>
</tr>
<tr>
<td>mlp_dim</td>
<td>IntegerParameterRange</td>
<td>MinValue: 16, MaxValue: 1024</td>
</tr>
<tr>
<td>mlp_layers</td>
<td>IntegerParameterRange</td>
<td>MinValue: 1, MaxValue: 4</td>
</tr>
<tr>
<td>optimizer</td>
<td>CategoricalParameterRanges</td>
<td>[adagrad, adam, rmsprop, sgd, adadelta]</td>
</tr>
<tr>
<td>weight_decay</td>
<td>ContinuousParameterRange</td>
<td>MinValue: 0.0, MaxValue: 1.0</td>
</tr>
</tbody>
</table>

**Data Formats for Object2Vec Training**

**Input: JSON Lines Request Format**

Content-type: application/jsonlines

```json
{"label": 0, "in0": [6, 17, 606, 19, 53, 67, 52, 12, 5, 10, 15, 10178, 7, 33, 652, 80, 15, 69, 821, 4], "in1": [16, 21, 13, 45, 14, 9, 80, 59, 164, 4]}

{"label": 1, "in0": [22, 1016, 32, 13, 25, 11, 5, 64, 573, 45, 5, 80, 15, 67, 21, 7, 9, 107, 4], "in1": [22, 32, 13, 25, 1016, 573, 3252, 4]}

{"label": 1, "in0": [774, 14, 21, 206], "in1": [21, 366, 125]}
```

The “in0” and “in1” are the inputs for encoder0 and encoder1, respectively. The same format is valid for both classification and regression problems. For regression, the field "label" can accept real valued inputs.

**Data Formats for Object2Vec Inference**

**GPU optimization: Classification or Regression**

Due to GPU memory scarcity, the INFEERENCE_PREFERRED_MODE environment variable can be specified to optimize on whether the classification/regression or the section called “Output: Encoder Embeddings” (p. 769) inference network is loaded into GPU. If the majority of your inference is for classification or regression, specify INFEERENCE_PREFERRED_MODE=classification. The following is a Batch Transform example of using 4 instances of p3.2xlarge that optimizes for classification/regression inference:

```python
transformer = o2v.transformer(instance_count=4,
                               instance_type="ml.p2.xlarge",
                               max_concurrent_transforms=2,
                               mode='classification')
```
max_payload=1, # 1MB
strategy='MultiRecord',
env={"INFERENCE_PREFERRED_MODE": 'classification'}, # only useful with GPU
output_path=output_s3_path)

### Input: Classification or Regression Request Format

**Content-type: application/json**

```
{
  "instances": [
    {
      "in0": [6, 17, 606, 19, 53, 67, 52, 12, 5, 10, 15, 10178, 7, 33, 652, 80, 15, 69, 821, 4],
      "in1": [16, 21, 13, 45, 14, 9, 80, 59, 164, 4]
    },
    {
      "in0": [22, 1016, 32, 13, 25, 11, 5, 64, 573, 45, 5, 80, 15, 67, 21, 7, 9, 107, 4],
      "in1": [22, 32, 13, 25, 1016, 573, 3252, 4]
    },
    {
      "in0": [774, 14, 21, 206],
      "in1": [21, 366, 125]
    }
  ]
}
```

**Content-type: application/jsonlines**

```
{"in0": [6, 17, 606, 19, 53, 67, 52, 12, 5, 10, 15, 10178, 7, 33, 652, 80, 15, 69, 821, 4],
 "in1": [16, 21, 13, 45, 14, 9, 80, 59, 164, 4]}
{"in0": [22, 1016, 32, 13, 25, 11, 5, 64, 573, 45, 5, 80, 15, 67, 21, 7, 9, 107, 4],
 "in1": [22, 32, 13, 25, 1016, 573, 3252, 4]}
{"in0": [774, 14, 21, 206],
 "in1": [21, 366, 125]}
```

For classification problems, the length of the scores vector corresponds to `num_classes`. For regression problems, the length is 1.

### Output: Classification or Regression Response Format

**Accept: application/json**

```
{
  "predictions": [
    {
      "scores": [
        0.653935070037842,
        0.07582679390907288,
        0.2707797586917877
      ]
    },
    {
      "scores": [
        0.026291321963071823,
        0.6577019095420837,
        0.31600672006607056
      ]
    }
  ]
}
```

**Accept: application/jsonlines**

```
{"scores": [0.195667684078216, 0.395351558923721, 0.408980727195739]}
{"scores": [0.251988261938095, 0.258233487606048, 0.489778339862823]}
{"scores": [0.28008779833847, 0.368331134319305, 0.35158109649169]}
```
In both the classification and regression formats, the scores apply to individual labels.

**Encoder Embeddings for Object2Vec**

**GPU optimization: Encoder Embeddings**

An embedding is a mapping from discrete objects, such as words, to vectors of real numbers.

Due to GPU memory scarcity, the `INFERENCE_PREFERRED_MODE` environment variable can be specified to optimize on whether the the section called "Inference Formats: Scoring" (p. 767) or the encoder embedding inference network is loaded into GPU. If the majority of your inference is for encoder embeddings, specify `INFERENCE_PREFERRED_MODE=embedding`. The following is a Batch Transform example of using 4 instances of p3.2xlarge that optimizes for encoder embedding inference:

```python
transformer = o2v.transformer(instance_count=4,
                              instance_type="ml.p2.xlarge",
                              max_concurrent_transforms=2,
                              max_payload=1,  # 1MB
                              strategy='MultiRecord',
                              env={'INFERENCE_PREFERRED_MODE': 'embedding'},  # only useful with GPU
                              output_path=output_s3_path)
```

**Input: Encoder Embeddings**

Content-type: application/json; infer_max_seqlens=<FWD-LENGTH>,<BCK-LENGTH>

Where `<FWD-LENGTH>` and `<BCK-LENGTH>` are integers in the range [1,5000] and define the maximum sequence lengths for the forward and backward encoder.

```json
{
"instances" : [
   {"in0": [6, 17, 606, 19, 53, 67, 52, 12, 5, 10, 15, 10178, 7, 33, 652, 80, 15, 69, 821, 4]},
   {"in0": [22, 1016, 32, 13, 25, 11, 5, 64, 573, 45, 5, 80, 15, 67, 21, 7, 9, 107, 4]},
   {"in0": [774, 14, 21, 206]}
]
}
```

Content-type: application/jsonlines; infer_max_seqlens=<FWD-LENGTH>,<BCK-LENGTH>

Where `<FWD-LENGTH>` and `<BCK-LENGTH>` are integers in the range [1,5000] and define the maximum sequence lengths for the forward and backward encoder.

```
{"in0": [6, 17, 606, 19, 53, 67, 52, 12, 5, 10, 15, 10178, 7, 33, 652, 80, 15, 69, 821, 4]}
{"in0": [22, 1016, 32, 13, 25, 11, 5, 64, 573, 45, 5, 80, 15, 67, 21, 7, 9, 107, 4]}
{"in0": [774, 14, 21, 206]}
```

In both of these formats, you specify only one input type: "in0" or "in1." The inference service then invokes the corresponding encoder and outputs the embeddings for each of the instances.

**Output: Encoder Embeddings**

Content-type: application/Embeddings

```
{
   "predictions": [
      {"embeddings":
        [0.057368703186511, 0.030703511089086, 0.099890425801277, 0.063688032329082, 0.026327300816774, 0.003637571120634, 0.021305780857801, 0.004316598642617, 0.0, 0.003397724591195, 0.0, 0.000378780066967, 0.0, 0.0, 0.0, 0.007419463712722]}
```

769
The vector length of the embeddings output by the inference service is equal to the value of one of the following hyperparameters that you specify at training time: `enc0_token_embedding_dim`, `enc1_token_embedding_dim`, or `enc_dim`.

**Object Detection Algorithm**

The Amazon SageMaker Object Detection algorithm detects and classifies objects in images using a single deep neural network. It is a supervised learning algorithm that takes images as input and identifies all instances of objects within the image scene. The object is categorized into one of the classes in a specified collection with a confidence score that it belongs to the class. Its location and scale in the image are indicated by a rectangular bounding box. It uses the Single Shot multibox Detector (SSD) framework and supports two base networks: VGG and ResNet. The network can be trained from scratch, or trained with models that have been pre-trained on the ImageNet dataset.

**Topics**
- Input/Output Interface for the Object Detection Algorithm (p. 770)
- EC2 Instance Recommendation for the Object Detection Algorithm (p. 773)
- Object Detection Sample Notebooks (p. 773)
- How Object Detection Works (p. 773)
- Object Detection Hyperparameters (p. 773)
- Tune an Object Detection Model (p. 778)
- Object Detection Request and Response Formats (p. 779)

**Input/Output Interface for the Object Detection Algorithm**

The SageMaker Object Detection algorithm supports both RecordIO (application/x-recordio) and image (image/png, image/jpeg, and application/x-image) content types for training in file mode and supports RecordIO (application/x-recordio) for training in pipe mode. However you can also train in pipe mode using the image files (image/png, image/jpeg, and application/x-image), without creating RecordIO files, by using the augmented manifest format. The recommended input format for the Amazon SageMaker object detection algorithms is Apache MXNet RecordIO. However, you can also use raw images in .jpg or .png format. The algorithm supports only application/x-image for inference.

**Note**

To maintain better interoperability with existing deep learning frameworks, this differs from the protobuf data formats commonly used by other Amazon SageMaker algorithms.

See the Object Detection Sample Notebooks (p. 773) for more details on data formats.

**Train with the RecordIO Format**

If you use the RecordIO format for training, specify both train and validation channels as values for the `InputDataConfig` parameter of the `CreateTrainingJob` request. Specify one RecordIO (.rec) file in
the train channel and one RecordIO file in the validation channel. Set the content type for both channels
to application/x-recordio. An example of how to generate RecordIO file can be found in the object
detection sample notebook. You can also use tools from the MXNet's GluonCV to generate RecordIO files
for popular datasets like the PASCAL Visual Object Classes and Common Objects in Context (COCO).

Train with the Image Format

If you use the image format for training, specify train, validation, train_annotation,
and validation_annotation channels as values for the InputDataConfig parameter of
CreateTrainingJob request. Specify the individual image data (.jpg or .png) files for the train and
validation channels. For annotation data, you can use the JSON format. Specify the corresponding .json
files in the train_annotation and validation_annotation channels. Set the content type for all
four channels to image/png or image/jpeg based on the image type. You can also use the content
type application/x-image when your dataset contains both .jpg and .png images. The following is an
example of a .json file.

```json
{
  "file": "$your_image_directory/sample_image1.jpg$",
  "image_size": [
    { 
      "width": 500,
      "height": 400,
      "depth": 3
    }
  ],
  "annotations": [ 
    { 
      "class_id": 0, 
      "left": 111, 
      "top": 134, 
      "width": 61, 
      "height": 128
    }, 
    { 
      "class_id": 0, 
      "left": 161, 
      "top": 250, 
      "width": 79, 
      "height": 143
    }, 
    { 
      "class_id": 1, 
      "left": 101, 
      "top": 185, 
      "width": 42, 
      "height": 130
    }
  ],
  "categories": [ 
    { 
      "class_id": 0, 
      "name": "dog"
    }, 
    { 
      "class_id": 1, 
      "name": "cat"
    }
  ]
}
```

Each image needs a .json file for annotation, and the .json file should have the same name as the
corresponding image. The name of above .json file should be "sample_image1.json". There are four
properties in the annotation .json file. The property "file" specifies the relative path of the image file.
For example, if your training images and corresponding .json files are stored in `s3://your_bucket/train/sample_image` and `s3://your_bucket/train_annotation`, specify the path for your train and train_annotation channels as `s3://your_bucket/train` and `s3://your_bucket/train_annotation`, respectively.

In the .json file, the relative path for an image named `sample_image1.jpg` should be `sample_image/sample_image1.jpg`. The "image_size" property specifies the overall image dimensions. The SageMaker object detection algorithm currently only supports 3-channel images. The "annotations" property specifies the categories and bounding boxes for objects within the image. Each object is annotated by a "class_id" index and by four bounding box coordinates ("left", "top", "width", "height"). The "left" (x-coordinate) and "top" (y-coordinate) values represent the upper-left corner of the bounding box. The "width" (x-coordinate) and "height" (y-coordinate) values represent the dimensions of the bounding box. The origin (0, 0) is the upper-left corner of the entire image. If you have multiple objects within one image, all the annotations should be included in a single .json file. The "categories" property stores the mapping between the class index and class name. The class indices should be numbered successively and the numbering should start with 0. The "categories" property is optional for the annotation .json file.

**Train with Augmented Manifest Image Format**

The augmented manifest format enables you to do training in pipe mode using image files without needing to create RecordIO files. You need to specify both train and validation channels as values for the `InputDataConfig` parameter of the `CreateTrainingJob` request. While using the format, an S3 manifest file needs to be generated that contains the list of images and their corresponding annotations. The manifest file format should be in JSON Lines format in which each line represents one sample. The images are specified using the 'source-ref' tag that points to the S3 location of the image. The annotations are provided under the "AttributeNames" parameter value as specified in the `CreateTrainingJob` request. It can also contain additional metadata under the `metadata` tag, but these are ignored by the algorithm. In the following example, the "AttributeNames" are contained in the list ["source-ref", "bounding-box"]:}

```json
[{"source-ref": "s3://your_bucket/image1.jpg", "bounding-box":{"image_size":{"width":500, "height":400, "depth":3}}, "annotations":{"class_id":0, "left":111, "top":134, "width":61, "height":128}, {"class_id":5, "left":161, "top":250, "width":80, "height":50}}, "bounding-box-metadata":{"class-map":{"0": "dog", "5": "horse"}, "type": "groundtruth/object-detection"}]
{"source-ref": "s3://your_bucket/image2.jpg", "bounding-box":{"image_size":{"width":400, "height":300, "depth":3}}, "annotations":{"class_id":1, "left":100, "top":120, "width":43, "height":78}}, "bounding-box-metadata":{"class-map":{"1": "cat"}, "type": "groundtruth/object-detection"}}
```

The order of "AttributeNames" in the input files matters when training the Object Detection algorithm. It accepts piped data in a specific order, with image first, followed by annotations. So the "AttributeNames" in this example are provided with "source-ref" first, followed by "bounding-box". When using Object Detection with Augmented Manifest, the value of parameter RecordWrapperType must be set as "RecordIO".

For more information on augmented manifest files, see [Provide Dataset Metadata to Training Jobs with an Augmented Manifest File](p. 1124).

**Incremental Training**

You can also seed the training of a new model with the artifacts from a model that you trained previously with SageMaker. Incremental training saves training time when you want to train a new model with the same or similar data. SageMaker object detection models can be seeded only with another built-in object detection model trained in SageMaker.

To use a pretrained model, in the `CreateTrainingJob` request, specify the ChannelName as "model" in the `InputDataConfig` parameter. Set the `ContentType` for the model channel to application/
Use built-in algorithms

x-sagemaker-model. The input hyperparameters of both the new model and the pretrained model that you upload to the model channel must have the same settings for the base_network and num_classes input parameters. These parameters define the network architecture. For the pretrained model file, use the compressed model artifacts (in .tar.gz format) output by SageMaker. You can use either RecordIO or image formats for input data.

For a sample notebook that shows how to use incremental training with the SageMaker object detection algorithm, see SageMaker Object Detection Incremental Training sample notebook. For more information on incremental training and for instructions on how to use it, see Incremental Training in Amazon SageMaker (p. 1118).

EC2 Instance Recommendation for the Object Detection Algorithm

For object detection, we support the following GPU instances for training: ml.p2.xlarge, ml.p2.8xlarge, ml.p2.16xlarge, ml.p3.2xlarge, ml.p3.8xlarge and ml.p3.16xlarge. We recommend using GPU instances with more memory for training with large batch sizes. You can also run the algorithm on multi-GPU and multi-machine settings for distributed training. However, both CPU (such as C5 and M5) and GPU (such as P2 and P3) instances can be used for the inference. All the supported instance types for inference are itemized on Amazon SageMaker ML Instance Types.

Object Detection Sample Notebooks

For a sample notebook that shows how to use the SageMaker Object Detection algorithm to train and host a model on the COCO dataset using the Single Shot multibox Detector algorithm, see Object Detection using the Image and JSON format. For instructions how to create and access Jupyter notebook instances that you can use to run the example in SageMaker, see Use Amazon SageMaker Notebook Instances (p. 124). Once you have created a notebook instance and opened it, select the SageMaker Examples tab to see a list of all the SageMaker samples. The object detection example notebook using the Object Detection algorithm is located in the Introduction to Amazon Algorithms section. To open a notebook, click on its Use tab and select Create copy.

How Object Detection Works

The object detection algorithm identifies and locates all instances of objects in an image from a known collection of object categories. The algorithm takes an image as input and outputs the category that the object belongs to, along with a confidence score that it belongs to the category. The algorithm also predicts the object's location and scale with a rectangular bounding box. Amazon SageMaker Object Detection uses the Single Shot multibox Detector (SSD) algorithm that takes a convolutional neural network (CNN) pretrained for classification task as the base network. SSD uses the output of intermediate layers as features for detection.

Various CNNs such as VGG and ResNet have achieved great performance on the image classification task. Object detection in Amazon SageMaker supports both VGG-16 and ResNet-50 as a base network for SSD. The algorithm can be trained in full training mode or in transfer learning mode. In full training mode, the base network is initialized with random weights and then trained on user data. In transfer learning mode, the base network weights are loaded from pretrained models.

The object detection algorithm uses standard data augmentation operations, such as flip, rescale, and jitter, on the fly internally to help avoid overfitting.

Object Detection Hyperparameters

In the CreateTrainingJob request, you specify the training algorithm that you want to use. You can also specify algorithm-specific hyperparameters that are used to help estimate the parameters of the model from a training dataset. The following table lists the hyperparameters provided by Amazon SageMaker for training the object detection algorithm. For more information about how object training works, see How Object Detection Works (p. 773).
<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>num_classes</td>
<td>The number of output classes. This parameter defines the dimensions of the network output and is typically set to the number of classes in the dataset.</td>
</tr>
<tr>
<td></td>
<td><strong>Required</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: positive integer</td>
</tr>
<tr>
<td>num_training_samples</td>
<td>The number of training examples in the input dataset.</td>
</tr>
<tr>
<td></td>
<td><strong>Note</strong></td>
</tr>
<tr>
<td></td>
<td>If there is a mismatch between this value and the number of samples in the training set, then the behavior of the lr_scheduler_step parameter will be undefined and distributed training accuracy may be affected.</td>
</tr>
<tr>
<td></td>
<td><strong>Required</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: positive integer</td>
</tr>
<tr>
<td>base_network</td>
<td>The base network architecture to use.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: 'vgg-16' or 'resnet-50'</td>
</tr>
<tr>
<td></td>
<td>Default value: 'vgg-16'</td>
</tr>
<tr>
<td>early_stopping</td>
<td>True to use early stopping logic during training. False not to use it.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: True or False</td>
</tr>
<tr>
<td></td>
<td>Default value: False</td>
</tr>
<tr>
<td>early_stopping_min_epochs</td>
<td>The minimum number of epochs that must be run before the early stopping logic can be invoked. It is used only when early_stopping = True.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: positive integer</td>
</tr>
<tr>
<td></td>
<td>Default value: 10</td>
</tr>
<tr>
<td>early_stopping_patience</td>
<td>The number of epochs to wait before ending training if no improvement, as defined by the early_stopping_tolerance hyperparameter, is made in the relevant metric. It is used only when early_stopping = True.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: positive integer</td>
</tr>
<tr>
<td></td>
<td>Default value: 5</td>
</tr>
</tbody>
</table>
### Parameter Name | Description
--- | ---
early_stopping_tolerance | The tolerance value that the relative improvement in validation:mAP, the mean average precision (mAP), is required to exceed to avoid early stopping. If the ratio of the change in the mAP divided by the previous best mAP is smaller than the `early_stopping_tolerance` value set, early stopping considers that there is no improvement. It is used only when `early_stopping` = True.

**Optional**

Valid values: $0 \leq \text{float} \leq 1$

Default value: 0.0

image_shape | The image size for input images. We rescale the input image to a square image with this size. We recommend using 300 and 512 for better performance.

**Optional**

Valid values: positive integer $\geq 300$

Default: 300

epochs | The number of training epochs.

**Optional**

Valid values: positive integer

Default: 30

freeze_layer_pattern | The regular expression (regex) for freezing layers in the base network. For example, if we set `freeze_layer_pattern = "^\{conv1_|conv2_\}.\*"`, then any layers with a name that contains "conv1_" or "conv2_" are frozen, which means that the weights for these layers are not updated during training. The layer names can be found in the network symbol files vgg16-symbol.json and resnet-50-symbol.json. Freezing a layer means that its weights can not be modified further. This can reduce training time significantly in exchange for modest losses in accuracy. This technique is commonly used in transfer learning where the lower layers in the base network do not need to be retrained.

**Optional**

Valid values: string

Default: No layers frozen.
<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
</table>
| kv_store       | The weight update synchronization mode used for distributed training. The weights can be updated either synchronously or asynchronously across machines. Synchronous updates typically provide better accuracy than asynchronous updates but can be slower. See the Distributed Training MXNet tutorial for details.  
**Note**  
This parameter is not applicable to single machine training.  
**Optional**  
Valid values: 'dist_sync' or 'dist_async'  
- 'dist_sync': The gradients are synchronized after every batch with all the workers. With 'dist_sync', batch-size now means the batch size used on each machine. So if there are n machines and we use batch size b, then dist_sync behaves like a single machine with batch size n*b.  
- 'dist_async': Performs asynchronous updates. The weights are updated whenever gradients are received from any machine and the weight updates are atomic. However, the order is not guaranteed.  
Default: - |
| label_width    | The force padding label width used to sync across training and validation data. For example, if one image in the data contains at most 10 objects, and each object's annotation is specified with 5 numbers, [class_id, left, top, width, height], then the label_width should be no smaller than (10*5 + header information length). The header information length is usually 2. We recommend using a slightly larger label_width for the training, such as 60 for this example.  
**Optional**  
Valid values: Positive integer large enough to accommodate the largest annotation information length in the data.  
Default: 350 |
| learning_rate  | The initial learning rate.  
**Optional**  
Valid values: float in (0, 1]  
Default: 0.001 |
<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>lr_scheduler_factor</td>
<td>The ratio to reduce learning rate. Used in conjunction with the lr_scheduler_step parameter defined as ( \text{lr_new} = \text{lr_old} \times \text{lr_scheduler_factor} ).</td>
</tr>
<tr>
<td></td>
<td>Optional</td>
</tr>
<tr>
<td></td>
<td>Valid values: float in (0, 1)</td>
</tr>
<tr>
<td></td>
<td>Default: 0.1</td>
</tr>
<tr>
<td>lr_scheduler_step</td>
<td>The epochs at which to reduce the learning rate. The learning rate is reduced by ( \text{lr_scheduler_factor} ) at epochs listed in a comma-delimited string: &quot;epoch1, epoch2, ...&quot;. For example, if the value is set to &quot;10, 20&quot; and the lr_scheduler_factor is set to 1/2, then the learning rate is halved after 10th epoch and then halved again after 20th epoch.</td>
</tr>
<tr>
<td></td>
<td>Optional</td>
</tr>
<tr>
<td></td>
<td>Valid values: string</td>
</tr>
<tr>
<td></td>
<td>Default: empty string</td>
</tr>
<tr>
<td>mini_batch_size</td>
<td>The batch size for training. In a single-machine multi-gpu setting, each GPU handles ( \text{mini_batch_size} / \text{num_gpu} ) training samples. For the multi-machine training in dist_sync mode, the actual batch size is ( \text{mini_batch_size} \times \text{number of machines} ). A large mini_batch_size generally leads to faster training, but it may cause out of memory problem. The memory usage is related to mini_batch_size, image_shape, and base_network architecture. For example, on a single p3.2xlarge instance, the largest mini_batch_size without an out of memory error is 32 with the base_network set to &quot;resnet-50&quot; and an image_shape of 300. With the same instance, you can use 64 as the mini_batch_size with the base network vgg-16 and an image_shape of 300.</td>
</tr>
<tr>
<td></td>
<td>Optional</td>
</tr>
<tr>
<td></td>
<td>Valid values: positive integer</td>
</tr>
<tr>
<td></td>
<td>Default: 32</td>
</tr>
<tr>
<td>momentum</td>
<td>The momentum for sgd. Ignored for other optimizers.</td>
</tr>
<tr>
<td></td>
<td>Optional</td>
</tr>
<tr>
<td></td>
<td>Valid values: float in (0, 1)</td>
</tr>
<tr>
<td></td>
<td>Default: 0.9</td>
</tr>
<tr>
<td>nms_threshold</td>
<td>The non-maximum suppression threshold.</td>
</tr>
<tr>
<td></td>
<td>Optional</td>
</tr>
<tr>
<td></td>
<td>Valid values: float in (0, 1)</td>
</tr>
<tr>
<td></td>
<td>Default: 0.45</td>
</tr>
<tr>
<td>Parameter Name</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>optimizer</strong></td>
<td>The optimizer types. For details on optimizer values, see MXNet’s API.</td>
</tr>
<tr>
<td><strong>overlap_threshold</strong></td>
<td>The evaluation overlap threshold.</td>
</tr>
<tr>
<td><strong>use_pretrained_model</strong></td>
<td>Indicates whether to use a pre-trained model for training. If set to 1, then the pre-trained model with corresponding architecture is loaded and used for training. Otherwise, the network is trained from scratch.</td>
</tr>
<tr>
<td><strong>weight_decay</strong></td>
<td>The weight decay coefficient for sgd and rmsprop. Ignored for other optimizers.</td>
</tr>
</tbody>
</table>

Tune an Object Detection Model

*Automatic model tuning*, also known as hyperparameter tuning, finds the best version of a model by running many jobs that test a range of hyperparameters on your dataset. You choose the tunable hyperparameters, a range of values for each, and an objective metric. You choose the objective metric from the metrics that the algorithm computes. Automatic model tuning searches the hyperparameters chosen to find the combination of values that result in the model that optimizes the objective metric.

For more information about model tuning, see Perform Automatic Model Tuning (p. 1023).

Metrics Computed by the Object Detection Algorithm

The object detection algorithm reports on a single metric during training: validation:mAP. When tuning a model, choose this metric as the objective metric.
Tunable Object Detection Hyperparameters

Tune the Amazon SageMaker object detection model with the following hyperparameters. The hyperparameters that have the greatest impact on the object detection objective metric are: mini_batch_size, learning_rate, and optimizer.

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Parameter Type</th>
<th>Recommended Ranges</th>
</tr>
</thead>
<tbody>
<tr>
<td>learning_rate</td>
<td>ContinuousParameterRange</td>
<td>MinValue: 1e-6, MaxValue: 0.5</td>
</tr>
<tr>
<td>mini_batch_size</td>
<td>IntegerParameterRanges</td>
<td>MinValue: 8, MaxValue: 64</td>
</tr>
<tr>
<td>momentum</td>
<td>ContinuousParameterRange</td>
<td>MinValue: 0.0, MaxValue: 0.999</td>
</tr>
<tr>
<td>optimizer</td>
<td>CategoricalParameterRanges</td>
<td>['sgd', 'adam', 'rmsprop', 'adadelta']</td>
</tr>
<tr>
<td>weight_decay</td>
<td>ContinuousParameterRange</td>
<td>MinValue: 0.0, MaxValue: 0.999</td>
</tr>
</tbody>
</table>

Object Detection Request and Response Formats

Request Format

Query a trained model by using the model's endpoint. The endpoint takes .jpg and .png image formats with image/jpeg and image/png content-types.

Response Formats

The response is the class index with a confidence score and bounding box coordinates for all objects within the image encoded in JSON format. The following is an example of response .json file:

```json
{"prediction": [
    [4.0, 0.86419455409049988, 0.3088374733924866, 0.07030484080314636, 0.7110607028007507, 0.9345266819000244],
    [0.0, 0.73376623392105103, 0.5714187026023865, 0.40427327156066895, 0.827075183391571, 0.9712159633636475],
    [4.0, 0.32643985450267792, 0.3677481412887573, 0.034883320331573486, 0.6318609714508057, 0.5967587828636169],
    [8.0, 0.22552496790885925, 0.6152569651603699, 0.5722782611846924, 0.882301390171051, 0.8985623121261597],
    [3.0, 0.42260299175977707, 0.19305512309074402, 0.08386176824569702, 0.39093565940856934, 0.9574796557426453]
]}
```

Each row in this .json file contains an array that represents a detected object. Each of these object arrays consists of a list of six numbers. The first number is the predicted class label. The second number is the associated confidence score for the detection. The last four numbers represent the bounding box coordinates [xmin, ymin, xmax, ymax]. These output bounding box corner indices are normalized by the overall image size. Note that this encoding is different than that use by the input .json format. For example, in the first entry of the detection result, 0.3088374733924866 is the left coordinate (x-coordinate of upper-left corner) of the bounding box as a ratio of the overall image width, 0.07030484080314636 is the top coordinate (y-coordinate of upper-left corner) of the bounding box as a ratio of the overall image height.
Use built-in algorithms

To avoid unreliable detection results, you might want to filter out the detection results with low confidence scores. In the object detection sample notebook, we provide scripts to remove the low confidence detections. Scripts are also provided to plot the bounding boxes on the original image.

For batch transform, the response is in JSON format, where the format is identical to the JSON format described above. The detection results of each image is represented as a JSON file. For example:

```json
{"prediction": [[label_id, confidence_score, xmin, ymin, xmax, ymax], [label_id, confidence_score, xmin, ymin, xmax, ymax]]}
```

For more details on training and inference, see the Object Detection Sample Notebooks (p. 773).

**OUTPUT: JSON Response Format**

```
accept: application/json;annotation=1
```

```json
{
  "image_size": [
    {
      "width": 500,
      "height": 400,
      "depth": 3
    }
  ],
  "annotations": [
    {
      "class_id": 0,
      "score": 0.943,
      "left": 111,
      "top": 134,
      "width": 61,
      "height": 128
    },
    {
      "class_id": 0,
      "score": 0.0013,
      "left": 161,
      "top": 250,
      "width": 79,
      "height": 143
    },
    {
      "class_id": 1,
      "score": 0.0133,
      "left": 101,
      "top": 185,
      "width": 42,
      "height": 130
    }
  ]
}
```

**Principal Component Analysis (PCA) Algorithm**

PCA is an unsupervised machine learning algorithm that attempts to reduce the dimensionality (number of features) within a dataset while still retaining as much information as possible. This is done by
finding a new set of features called components, which are composites of the original features that are uncorrelated with one another. They are also constrained so that the first component accounts for the largest possible variability in the data, the second component the second most variability, and so on.

In Amazon SageMaker, PCA operates in two modes, depending on the scenario:

- **regular**: For datasets with sparse data and a moderate number of observations and features.
- **randomized**: For datasets with both a large number of observations and features. This mode uses an approximation algorithm.

PCA uses tabular data.

The rows represent observations you want to embed in a lower dimensional space. The columns represent features that you want to find a reduced approximation for. The algorithm calculates the covariance matrix (or an approximation thereof in a distributed manner), and then performs the singular value decomposition on this summary to produce the principal components.

**Topics**
- Input/Output Interface for the PCA Algorithm (p. 781)
- EC2 Instance Recommendation for the PCA Algorithm (p. 781)
- PCA Sample Notebooks (p. 781)
- How PCA Works (p. 782)
- PCA Hyperparameters (p. 783)
- PCA Response Formats (p. 784)

**Input/Output Interface for the PCA Algorithm**

For training, PCA expects data provided in the train channel, and optionally supports a dataset passed to the test dataset, which is scored by the final algorithm. Both recordIO-wrapped-protobuf and CSV formats are supported for training. You can use either File mode or Pipe mode to train models on data that is formatted as recordIO-wrapped-protobuf or as CSV.

For inference, PCA supports text/csv, application/json, and application/x-recordio-protobuf. Results are returned in either application/json or application/x-recordio-protobuf format with a vector of "projections."

For more information on input and output file formats, see PCA Response Formats (p. 784) for inference and the PCA Sample Notebooks (p. 781).

**EC2 Instance Recommendation for the PCA Algorithm**

PCA supports both GPU and CPU computation. Which instance type is most performant depends heavily on the specifics of the input data.

**PCA Sample Notebooks**

For a sample notebook that shows how to use the SageMaker Principal Component Analysis algorithm to analyze the images of handwritten digits from zero to nine in the MNIST dataset, see An Introduction to PCA with MNIST. For instructions how to create and access Jupyter notebook instances that you can use to run the example in SageMaker, see Use Amazon SageMaker Notebook Instances (p. 124). Once you have created a notebook instance and opened it, select the SageMaker Examples tab to see a list of all the SageMaker samples. The topic modeling example notebooks using the NTM algorithms are located in the Introduction to Amazon algorithms section. To open a notebook, click on its Use tab and select Create copy.
How PCA Works

Principal Component Analysis (PCA) is a learning algorithm that reduces the dimensionality (number of features) within a dataset while still retaining as much information as possible.

PCA reduces dimensionality by finding a new set of features called components, which are composites of the original features, but are uncorrelated with one another. The first component accounts for the largest possible variability in the data, the second component the second most variability, and so on.

It is an unsupervised dimensionality reduction algorithm. In unsupervised learning, labels that might be associated with the objects in the training dataset aren't used.

Given the input of a matrix with rows \( x_1, \ldots, x_n \) each of dimension \( 1 \times d \), the data is partitioned into mini-batches of rows and distributed among the training nodes (workers). Each worker then computes a summary of its data. The summaries of the different workers are then unified into a single solution at the end of the computation.

Modes

The Amazon SageMaker PCA algorithm uses either of two modes to calculate these summaries, depending on the situation:

- **regular**: for datasets with sparse data and a moderate number of observations and features.
- **randomized**: for datasets with both a large number of observations and features. This mode uses an approximation algorithm.

As the algorithm’s last step, it performs the singular value decomposition on the unified solution, from which the principal components are then derived.

**Mode 1: Regular**

The workers jointly compute both \( \sum x_i^T x_i \) and \( \sum x_i \).

**Note**

Because \( x_i \) are \( 1 \times d \) row vectors, \( x_i^T x_i \) is a matrix (not a scalar). Using row vectors within the code allows us to obtain efficient caching.

The covariance matrix is computed as \( \sum x_i^T x_i - \frac{1}{n} (\sum x_i)^T \sum x_i \), and its top num_components singular vectors form the model.

**Note**

If subtract_mean is False, we avoid computing and subtracting \( \sum x_i \).

Use this algorithm when the dimension \( d \) of the vectors is small enough so that \( d^2 \) can fit in memory.

**Mode 2: Randomized**

When the number of features in the input dataset is large, we use a method to approximate the covariance metric. For every mini-batch \( X_i \) of dimension \( b \times d \), we randomly initialize a \( (\text{num_components} + \text{extra_components}) \times b \) matrix that we multiply by each mini-batch, to create a \( (\text{num_components} + \text{extra_components}) \times d \) matrix. The sum of these matrices is computed by the workers, and the servers perform SVD on the final \( (\text{num_components} + \text{extra_components}) \times d \) matrix. The top right num_components singular vectors of it are the approximation of the top singular vectors of the input matrix.

Let \( \ell = \text{num_components} + \text{extra_components} \). Given a mini-batch \( X_i \) of dimension \( b \times d \), the worker draws a random matrix \( H_i \) of dimension \( \ell \times b \). Depending on whether the environment uses a
GPU or CPU and the dimension size, the matrix is either a random sign matrix where each entry is $\pm 1$ or a FJLT (fast Johnson Lindenstrauss transform; for information, see FJLT Transforms and the follow-up papers). The worker then computes $H_i X_i$ and maintains $B = \sum_i H_i X_i$. The worker also maintains $h^T$, the sum of columns of $H_1, \ldots, H_T$ ($T$ being the total number of mini-batches), and $s$, the sum of all input rows. After processing the entire shard of data, the worker sends the server $B$, $h$, $s$, and $n$ (the number of input rows).

Denote the different inputs to the server as $B^i, h^i, s^i, n^i$. The server computes $B$, $h$, $s$, $n$ the sums of the respective inputs. It then computes $C = B - \frac{1}{n} h h^T s$, and finds its singular value decomposition. The top-right singular vectors and singular values of $C$ are used as the approximate solution to the problem.

**PCA Hyperparameters**

In the CreateTrainingJob request, you specify the training algorithm. You can also specify algorithm-specific HyperParameters as string-to-string maps. The following table lists the hyperparameters for the PCA training algorithm provided by Amazon SageMaker. For more information about how PCA works, see How PCA Works (p. 782).

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>feature_dim</td>
<td>Input dimension.</td>
</tr>
<tr>
<td></td>
<td><strong>Required</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: positive integer</td>
</tr>
<tr>
<td>mini_batch_size</td>
<td>Number of rows in a mini-batch.</td>
</tr>
<tr>
<td></td>
<td><strong>Required</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: positive integer</td>
</tr>
<tr>
<td>num_components</td>
<td>The number of principal components to compute.</td>
</tr>
<tr>
<td></td>
<td><strong>Required</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: positive integer</td>
</tr>
<tr>
<td>algorithm_mode</td>
<td>Mode for computing the principal components.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: regular or randomized</td>
</tr>
<tr>
<td></td>
<td>Default value: regular</td>
</tr>
<tr>
<td>extra_components</td>
<td>As the value increases, the solution becomes more accurate but the runtime and memory consumption increase linearly. The default, -1, means the maximum of 10 and num_components. Valid for randomized mode only.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: Non-negative integer or -1</td>
</tr>
<tr>
<td></td>
<td>Default value: -1</td>
</tr>
<tr>
<td>subtract_mean</td>
<td>Indicates whether the data should be unbiased both during training and at inference.</td>
</tr>
<tr>
<td>Parameter Name</td>
<td>Description</td>
</tr>
<tr>
<td>----------------</td>
<td>-------------</td>
</tr>
<tr>
<td>Optional</td>
<td>Valid values: One of true or false</td>
</tr>
<tr>
<td></td>
<td>Default value: true</td>
</tr>
</tbody>
</table>

**PCA Response Formats**

All Amazon SageMaker built-in algorithms adhere to the common input inference format described in [Common Data Formats - Inference](#). This topic contains a list of the available output formats for the SageMaker PCA algorithm.

**JSON Response Format**

Accept—application/json

```json
{
    "projections": [
        {
            "projection": [1.0, 2.0, 3.0, 4.0, 5.0]
        },
        {
            "projection": [6.0, 7.0, 8.0, 9.0, 0.0]
        },
        ...
    ]
}
```

**JSONLINES Response Format**

Accept—application/jsonlines

```json
{ "projection": [1.0, 2.0, 3.0, 4.0, 5.0] }
{ "projection": [6.0, 7.0, 8.0, 9.0, 0.0] }
```

**RECORDIO Response Format**

Accept—application/x-recordio-protobuf

```json
[{
    Record = {
        features = {},
        label = {
            'projection': {
                keys: [],
                values: [1.0, 2.0, 3.0, 4.0, 5.0]
            }
        }
    },
    Record = {
        features = {},
        label = {
            'projection': {
                keys: [],
                values: [1.0, 2.0, 3.0, 4.0, 5.0]
            }
        }
    }
}
```
Random Cut Forest (RCF) Algorithm

Amazon SageMaker Random Cut Forest (RCF) is an unsupervised algorithm for detecting anomalous data points within a data set. These are observations which diverge from otherwise well-structured or patterned data. Anomalies can manifest as unexpected spikes in time series data, breaks in periodicity, or unclassifiable data points. They are easy to describe in that, when viewed in a plot, they are often easily distinguishable from the "regular" data. Including these anomalies in a data set can drastically increase the complexity of a machine learning task since the "regular" data can often be described with a simple model.

With each data point, RCF associates an anomaly score. Low score values indicate that the data point is considered "normal." High values indicate the presence of an anomaly in the data. The definitions of "low" and "high" depend on the application but common practice suggests that scores beyond three standard deviations from the mean score are considered anomalous.

While there are many applications of anomaly detection algorithms to one-dimensional time series data such as traffic volume analysis or sound volume spike detection, RCF is designed to work with arbitrary-dimensional input. Amazon SageMaker RCF scales well with respect to number of features, data set size, and number of instances.

Topics

- Input/Output Interface for the RCF Algorithm (p. 785)
- Instance Recommendations for the RCF Algorithm (p. 786)
- RCF Sample Notebooks (p. 786)
- How RCF Works (p. 786)
- RCF Hyperparameters (p. 789)
- Tune an RCF Model (p. 790)
- RCF Response Formats (p. 790)

Input/Output Interface for the RCF Algorithm

Amazon SageMaker Random Cut Forest supports the train and test data channels. The optional test channel is used to compute accuracy, precision, recall, and F1-score metrics on labeled data. Train and test data content types can be either application/x-recordio-protobuf or text/csv formats. For the test data, when using text/csv format, the content must be specified as text/csv;label_size=1 where the first column of each row represents the anomaly label: "1" for an anomalous data point and "0" for a normal data point. You can use either File mode or Pipe mode to train RCF models on data that is formatted as recordIO-wrapped-protobuf or as CSV.

Also note that the train channel only supports S3DataDistributionType=ShardedByS3Key and the test channel only supports S3DataDistributionType=FullyReplicated. The S3 distribution type can be specified using the Amazon SageMaker Python SDK as follows:

```python
import sagemaker

# specify Random Cut Forest training job information and hyperparameters
rcf = sagemaker.estimator.Estimator(...)

# explicitly specify "ShardedByS3Key" distribution type
train_data = sagemaker.inputs.s3_input(
    s3_data=s3_training_data_location,
    content_type='text/csv;label_size=0',
    distribution='ShardedByS3Key')
```
# run the training job on input data stored in S3
rcf.fit({'train': train_data})

See the S3DataSource for more information on customizing the S3 data source attributes. Finally, in order to take advantage of multi-instance training the training data must be partitioned into at least as many files as instances.

For inference, RCF supports application/x-recordio-protobuf, text/csv and application/json input data content types. See the Common Data Formats for Built-in Algorithms (p. 646) documentation for more information. RCF inference returns application/x-recordio-protobuf or application/json formatted output. Each record in these output data contains the corresponding anomaly scores for each input data point. See Common Data Formats--Inference for more information.

For more information on input and output file formats, see RCF Response Formats (p. 790) for inference and the RCF Sample Notebooks (p. 786).

**Instance Recommendations for the RCF Algorithm**

For training, we recommend the ml.m4, ml.c4, and ml.c5 instance families. For inference we recommend using a ml.c5.xl instance type in particular, for maximum performance as well as minimized cost per hour of usage. Although the algorithm could technically run on GPU instance types it does not take advantage of GPU hardware.

**RCF Sample Notebooks**

For an example of how to train an RCF model and perform inferences with it, see the Introduction to SageMaker Random Cut Forests notebook. For a sample notebook that uses the SageMaker Random Cut Forest algorithm for anomaly detection, see An Introduction to SageMaker Random Cut Forests. For instructions how to create and access Jupyter notebook instances that you can use to run the example in SageMaker, see Use Amazon SageMaker Notebook Instances (p. 124). Once you have created a notebook instance and opened it, select the SageMaker Examples tab to see a list of all the SageMaker samples. To open a notebook, click on its Use tab and select Create copy.

**How RCF Works**

Amazon SageMaker Random Cut Forest (RCF) is an unsupervised algorithm for detecting anomalous data points within a dataset. These are observations which diverge from otherwise well-structured or patterned data. Anomalies can manifest as unexpected spikes in time series data, breaks in periodicity, or unclassifiable data points. They are easy to describe in that, when viewed in a plot, they are often easily distinguishable from the "regular" data. Including these anomalies in a dataset can drastically increase the complexity of a machine learning task since the "regular" data can often be described with a simple model.

The main idea behind the RCF algorithm is to create a forest of trees where each tree is obtained using a partition of a sample of the training data. For example, a random sample of the input data is first determined. The random sample is then partitioned according to the number of trees in the forest. Each tree is given such a partition and organizes that subset of points into a k-d tree. The anomaly score assigned to a data point by the tree is defined as the expected change in complexity of the tree as a result adding that point to the tree; which, in approximation, is inversely proportional to the resulting depth of the point in the tree. The random cut forest assigns an anomaly score by computing the average score from each constituent tree and scaling the result with respect to the sample size. The RCF algorithm is based on the one described in reference [1].

**Sample Data Randomly**

The first step in the RCF algorithm is to obtain a random sample of the training data. In particular, suppose we want a sample of size $K$ from $N$ total data points. If the training data is small enough, the entire dataset can be used, and we could randomly draw $K$ elements from this set. However,
frequently the training data is too large to fit all at once, and this approach isn’t feasible. Instead, we use a technique called reservoir sampling.

Reservoir sampling is an algorithm for efficiently drawing random samples from a dataset $S = \{S_1, \ldots, S_N\}$ where the elements in the dataset can only be observed one at a time or in batches. In fact, reservoir sampling works even when $N$ is not known a priori. If only one sample is requested, such as when $K = 1$, the algorithm is like this:

**Algorithm: Reservoir Sampling**

- Input: dataset or data stream $S = \{S_1, \ldots, S_N\}$
- Initialize the random sample $X = S_1$
- For each observed sample $S_n, n = 2, \ldots, N$:
  - Pick a uniform random number $\xi \in [0, 1]$
  - If $\xi < 1/n$
    - Set $X = S_n$
  - Return $X$

This algorithm selects a random sample such that $P(X = S_n) = 1/N$ for all $n = 1, \ldots, N$. When $K > 1$ the algorithm is more complicated. Additionally, a distinction must be made between random sampling that is with and without replacement. RCF performs an augmented reservoir sampling without replacement on the training data based on the algorithms described in [2].

**Train a RCF Model and Produce Inferences**

The next step in RCF is to construct a random cut forest using the random sample of data. First, the sample is partitioned into a number of equal-sized partitions equal to the number of trees in the forest. Then, each partition is sent to an individual tree. The tree recursively organizes its partition into a binary tree by partitioning the data domain into bounding boxes.

This procedure is best illustrated with an example. Suppose a tree is given the following two-dimensional dataset. The corresponding tree is initialized to the root node:

![Root](Root)

A two-dimensional dataset where the majority of data lies in a cluster (blue) except for one anomalous data point (orange). The tree is initialized with a root node.
The RCF algorithm organizes these data in a tree by first computing a bounding box of the data, selecting a random dimension (giving more weight to dimensions with higher "variance"), and then randomly determining the position of a hyperplane "cut" through that dimension. The two resulting subspaces define their own sub tree. In this example, the cut happens to separate a lone point from the remainder of the sample. The first level of the resulting binary tree consists of two nodes, one which will consist of the subtree of points to the left of the initial cut and the other representing the single point on the right.

A random cut partitioning the two-dimensional dataset. An anomalous data point is more likely to lie isolated in a bounding box at a smaller tree depth than other points.

Bounding boxes are then computed for the left and right halves of the data and the process is repeated until every leaf of the tree represents a single data point from the sample. Note that if the lone point is sufficiently far away then it is more likely that a random cut would result in point isolation. This observation provides the intuition that tree depth is, loosely speaking, inversely proportional to the anomaly score.

When performing inference using a trained RCF model the final anomaly score is reported as the average across scores reported by each tree. Note that it is often the case that the new data point does not already reside in the tree. To determine the score associated with the new point the data point is inserted into the given tree and the tree is efficiently (and temporarily) reassembled in a manner equivalent to the training process described above. That is, the resulting tree is as if the input data point were a member of the sample used to construct the tree in the first place. The reported score is inversely proportional to the depth of the input point within the tree.

Choose Hyperparameters

The primary hyperparameters used to tune the RCF model are num_trees and num_samples_per_tree. Increasing num_trees has the effect of reducing the noise observed in anomaly scores since the final score is the average of the scores reported by each tree. While the optimal value is application-dependent we recommend using 100 trees to begin with as a balance between score noise and model complexity. Note that inference time is proportional to the number of trees. Although training time is also affected it is dominated by the reservoir sampling algorithm describe above.

The parameter num_samples_per_tree is related to the expected density of anomalies in the dataset. In particular, num_samples_per_tree should be chosen such that 1/num_samples_per_tree approximates the ratio of anomalous data to normal data. For example, if 256 samples are used in each
tree then we expect our data to contain anomalies 1/256 or approximately 0.4% of the time. Again, an optimal value for this hyperparameter is dependent on the application.

References


**RCF Hyperparameters**

In the `CreateTrainingJob` request, you specify the training algorithm. You can also specify algorithm-specific hyperparameters as string-to-string maps. The following table lists the hyperparameters for the Amazon SageMaker RCF algorithm. For more information, including recommendations on how to choose hyperparameters, see *How RCF Works* (p. 786).

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>feature_dim</td>
<td>The number of features in the data set. (If you use the Random Cut Forest estimator, this value is calculated for you and need not be specified.)</td>
</tr>
<tr>
<td></td>
<td><strong>Required</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: Positive integer (min: 1, max: 10000)</td>
</tr>
<tr>
<td>eval_metrics</td>
<td>A list of metrics used to score a labeled test data set. The following metrics can be selected for output:</td>
</tr>
<tr>
<td></td>
<td>• accuracy - returns fraction of correct predictions.</td>
</tr>
<tr>
<td></td>
<td>• precision_recall_fscore - returns the positive and negative precision, recall, and F1-scores.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: a list with possible values taken from accuracy or precision_recall_fscore.</td>
</tr>
<tr>
<td></td>
<td>Default value: Both accuracy, precision_recall_fscore are calculated.</td>
</tr>
<tr>
<td>num_samples_per_tree</td>
<td>Number of random samples given to each tree from the training data set.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: Positive integer (min: 1, max: 2048)</td>
</tr>
<tr>
<td></td>
<td>Default value: 256</td>
</tr>
<tr>
<td>num_trees</td>
<td>Number of trees in the forest.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: Positive integer (min: 50, max: 1000)</td>
</tr>
</tbody>
</table>
Use built-in algorithms

Parameter Name | Description | Default value: 100
--- | --- | ---

Tune an RCF Model

Automatic model tuning, also known as hyperparameter tuning or hyperparameter optimization, finds the best version of a model by running many jobs that test a range of hyperparameters on your dataset. You choose the tunable hyperparameters, a range of values for each, and an objective metric. You choose the objective metric from the metrics that the algorithm computes. Automatic model tuning searches the hyperparameters chosen to find the combination of values that result in the model that optimizes the objective metric.

The Amazon SageMaker RCF algorithm is an unsupervised anomaly-detection algorithm that requires a labeled test dataset for hyperparameter optimization. RCF calculates anomaly scores for test datapoints and then labels the datapoints as anomalous if their scores are beyond three standard deviations from the mean score. This is known as the three-sigma limit heuristic. The F1-score is based on the difference between calculated labels and actual labels. The hyperparameter tuning job finds the model that maximizes that score. The success of hyperparameter optimization depends on the applicability of the three-sigma limit heuristic to the test dataset.

For more information about model tuning, see Perform Automatic Model Tuning (p. 1023).

Metrics Computed by the RCF Algorithm

The RCF algorithm computes the following metric during training. When tuning the model, choose this metric as the objective metric.

<table>
<thead>
<tr>
<th>Metric Name</th>
<th>Description</th>
<th>Optimization Direction</th>
</tr>
</thead>
<tbody>
<tr>
<td>test:f1</td>
<td>F1-score on the test dataset, based on the difference between calculated labels and actual labels.</td>
<td>Maximize</td>
</tr>
</tbody>
</table>

Tunable RCF Hyperparameters

You can tune a RCF model with the following hyperparameters.

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Parameter Type</th>
<th>Recommended Ranges</th>
</tr>
</thead>
<tbody>
<tr>
<td>num_samples_per_tree</td>
<td>IntegerParameterRanges</td>
<td>MinValue: 1, MaxValue: 2048</td>
</tr>
<tr>
<td>num_trees</td>
<td>IntegerParameterRanges</td>
<td>MinValue: 50, MaxValue: 1000</td>
</tr>
</tbody>
</table>

RCF Response Formats

All Amazon SageMaker built-in algorithms adhere to the common input inference format described in Common Data Formats - Inference. Note that SageMaker Random Cut Forest supports both dense and sparse JSON and RecordIO formats. This topic contains a list of the available output formats for the SageMaker RCF algorithm.
JSON Response Format

ACCEPT: application/json.

```json
{
    "scores": [
        {
            "score": 0.02
        },
        {
            "score": 0.25
        }
    ]
}
```

JSONLINES Response Format

ACCEPT: application/jsonlines.

```jsonlines
{"score": 0.02},
{"score": 0.25}
```

RECORDIO Response Format

ACCEPT: application/x-recordio-protobuf.

```protobuf
[
  Record = {
    features = {},
    label = {
      'score': {
        keys: [],
        values: [0.25] # float32
      }
    }
  }
]```
Semantic Segmentation Algorithm

The SageMaker semantic segmentation algorithm provides a fine-grained, pixel-level approach to developing computer vision applications. It tags every pixel in an image with a class label from a predefined set of classes. Tagging is fundamental for understanding scenes, which is critical to an increasing number of computer vision applications, such as self-driving vehicles, medical imaging diagnostics, and robot sensing.

For comparison, the SageMaker Image Classification Algorithm (p. 690) is a supervised learning algorithm that analyzes only whole images, classifying them into one of multiple output categories. The Object Detection Algorithm (p. 770) is a supervised learning algorithm that detects and classifies all
instances of an object in an image. It indicates the location and scale of each object in the image with a rectangular bounding box.

Because the semantic segmentation algorithm classifies every pixel in an image, it also provides information about the shapes of the objects contained in the image. The segmentation output is represented as a grayscale image, called a segmentation mask. A segmentation mask is a grayscale image with the same shape as the input image.

The SageMaker semantic segmentation algorithm is built using the MXNet Gluon framework and the Gluon CV toolkit, and provides you with a choice of three build-in algorithms to train a deep neural network. You can use the Fully-Convolutional Network (FCN) algorithm, Pyramid Scene Parsing (PSP) algorithm, or DeepLabV3.

Each of the three algorithms has two distinct components:

- The **backbone** (or encoder)—A network that produces reliable activation maps of features.
- The **decoder**—A network that constructs the segmentation mask from the encoded activation maps.

You also have a choice of backbones for the FCN, PSP, and DeepLabV3 algorithms: ResNet50 or ResNet101. These backbones include pretrained artifacts that were originally trained on the ImageNet classification task. You can fine-tune these backbones for segmentation using your own data. Or, you can initialize and train these networks from scratch using only your own data. The decoders are never pretrained.

To deploy the trained model for inference, use the SageMaker hosting service. During inference, you can request the segmentation mask either as a PNG image or as a set of probabilities for each class for each pixel. You can use these masks as part of a larger pipeline that includes additional downstream image processing or other applications.

**Topics**

- **Semantic Segmentation Sample Notebooks (p. 793)**
- **Input/Output Interface for the Semantic Segmentation Algorithm (p. 793)**
- **EC2 Instance Recommendation for the Semantic Segmentation Algorithm (p. 796)**
- **Semantic Segmentation Hyperparameters (p. 796)**

**Semantic Segmentation Sample Notebooks**

For a sample Jupyter notebook that uses the SageMaker semantic segmentation algorithm to train a model and deploy it to perform inferences, see the Semantic Segmentation Example. For instructions on how to create and access Jupyter notebook instances that you can use to run the example in SageMaker, see Use Amazon SageMaker Notebook Instances (p. 124).

To see a list of all of the SageMaker samples, create and open a notebook instance, and choose the **SageMaker Examples** tab. The example semantic segmentation notebooks are located under Introduction to Amazon algorithms. To open a notebook, choose its **Use** tab, and choose **Create copy**.

**Input/Output Interface for the Semantic Segmentation Algorithm**

SageMaker semantic segmentation expects the customer's training dataset to be on Amazon Simple Storage Service (Amazon S3). Once trained, it produces the resulting model artifacts on Amazon S3. The input interface format for the SageMaker semantic segmentation is similar to that of most standardized semantic segmentation benchmarking datasets. The dataset in Amazon S3 is expected to be presented in two channels, one for train and one for validation using four directories, two for images and two for annotations. Annotations are expected to be uncompressed PNG images. The dataset might also have a label map that describes how the annotation mappings are established. If not, the algorithm uses a
default. It also supports the augmented manifest image format (application/x-image) for training in Pipe input mode straight from Amazon S3. For inference, an endpoint accepts images with an image/jpeg content type.

**How Training Works**

The training data is split into four directories: train, train_annotation, validation, and validation_annotation. There is a channel for each of these directories. The dataset also expected to have one label_map.json file per channel for train_annotation and validation_annotation respectively. If you don’t provide these JSON files, SageMaker provides the default set label map.

The dataset specifying these files should look similar to the following example:

```
s3://bucket_name
   | - train
   |    | - 0000.jpg
   |    | - coffee.jpg
   | - validation
   |    | - 00a0.jpg
   |    | - banana.jpg
   | - train_annotation
   |    | - 0000.png
   |    | - coffee.png
   | - validation_annotation
   |    | - 00a0.png
   |    | - banana.png
   | - label_map
   |    | - train_label_map.json
   |    | - validation_label_map.json
```

Every JPG image in the train and validation directories has a corresponding PNG label image with the same name in the train_annotation and validation_annotation directories. This naming convention helps the algorithm to associate a label with its corresponding image during training. The train, train_annotation, validation, and validation_annotation channels are mandatory. The annotations are single-channel PNG images. The format works as long as the metadata (modes) in the image helps the algorithm read the annotation images into a single-channel 8-bit unsigned integer. For more information on our support for modes, see the [Python Image Library documentation](https://docs.python.org/3/library/images). We recommend using the 8-bit pixel, true color P mode.

The image that is encoded is a simple 8-bit integer when using modes. To get from this mapping to a map of a label, the algorithm uses one mapping file per channel, called the *label map*. The label map is used to map the values in the image with actual label indices. In the default label map, which is provided by default if you don’t provide one, the pixel value in an annotation matrix (image) directly index the label. These images can be grayscale PNG files or 8-bit indexed PNG files. The label map file for the unscaled default case is the following:

```
{
    "scale": "1"
}
```

To provide some contrast for viewing, some annotation software scales the label images by a constant amount. To support this, the SageMaker semantic segmentation algorithm provides a rescaling option to scale down the values to actual label values. When scaling down doesn’t convert the value to an appropriate integer, the algorithm defaults to the greatest integer less than or equal to the scale value. The following code shows how to set the scale value to rescale the label values:
The following example shows how this "scale" value is used to rescale the encoded_label values of the input annotation image when they are mapped to the mapped_label values to be used in training. The label values in the input annotation image are 0, 3, 6, with scale 3, so they are mapped to 0, 1, 2 for training:

encoded_label = [0, 3, 6]
mapped_label = [0, 1, 2]

In some cases, you might need to specify a particular color mapping for each class. Use the map option in the label mapping as shown in the following example of a label_map file:

```json
{
  "map": {
    "0": 5,
    "1": 0,
    "2": 2
  }
}
```

This label mapping for this example is:

encoded_label = [0, 5, 2]
mapped_label = [1, 0, 2]

With label mappings, you can use different annotation systems and annotation software to obtain data without a lot of preprocessing. You can provide one label map per channel. The files for a label map in the label_map channel must follow the naming conventions for the four directory structure. If you don't provide a label map, the algorithm assumes a scale of 1 (the default).

**Training with the Augmented Manifest Format**

The augmented manifest format enables you to do training in Pipe mode using image files without needing to create RecordIO files. The augmented manifest file contains data objects and should be in JSON Lines format, as described in the CreateTrainingJob request. Each line in the manifest is an entry containing the Amazon S3 URI for the image and the URI for the annotation image.

Each JSON object in the manifest file must contain a source-ref key. The source-ref key should contain the value of the Amazon S3 URI to the image. The labels are provided under the AttributeNames parameter value as specified in the CreateTrainingJob request. It can also contain additional metadata under the metadata tag, but these are ignored by the algorithm. In the example below, the AttributeNames are contained in the list of image and annotation references ["source-ref", "city-streets-ref"]. These names must have -ref appended to them. When using the Semantic Segmentation algorithm with Augmented Manifest, the value of the RecordWrapperType parameter must be "RecordIO" and value of the ContentType parameter must be application/x-recordio.

```json
{"source-ref": "S3 bucket location", "city-streets-ref": "S3 bucket location", "city-streets-metadata": {"job-name": "label-city-streets", }}
```

For more information on augmented manifest files, see [Provide Dataset Metadata to Training Jobs with an Augmented Manifest File](p. 1124).
Incremental Training

You can also seed the training of a new model with a model that you trained previously using SageMaker. This incremental training saves training time when you want to train a new model with the same or similar data. Currently, incremental training is supported only for models trained with the built-in SageMaker Semantic Segmentation.

To use your own pre-trained model, specify the ChannelName as "model" in the InputDataConfig for the CreateTrainingJob request. Set the ContentType for the model channel to application/x-sagemaker-model. The backbone, algorithm, crop_size, and num_classes input parameters that define the network architecture must be consistently specified in the input hyperparameters of the new model and the pre-trained model that you upload to the model channel. For the pretrained model file, you can use the compressed (.tar.gz) artifacts from SageMaker outputs. You can only use Image formats for input data. For more information on incremental training and for instructions on how to use it, see Incremental Training in Amazon SageMaker (p. 1118).

Produce Inferences

To query a trained model that is deployed to an endpoint, you need to provide an image and an AcceptType that denotes the type of output required. The endpoint takes JPEG images with an image/jpeg content type. If you request an AcceptType of image/png, the algorithm outputs a PNG file with a segmentation mask in the same format as the labels themselves. If you request an accept type of application/x-recordio-protobuf, the algorithm returns class probabilities encoded in recordio-protobuf format. The latter format outputs a 3D tensor where the third dimension is the same size as the number of classes. This component denotes the probability of each class label for each pixel.

EC2 Instance Recommendation for the Semantic Segmentation Algorithm

The SageMaker semantic segmentation algorithm only supports GPU instances for training, and we recommend using GPU instances with more memory for training with large batch sizes. The algorithm can be trained using P2/P3 EC2 Amazon Elastic Compute Cloud (Amazon EC2) instances in single machine configurations. It supports the following GPU instances for training:

- ml.p2.xlarge
- ml.p2.8xlarge
- ml.p2.16xlarge
- ml.p3.2xlarge
- ml.p3.8xlarge
- ml.p3.16xlarge

For inference, you can use either CPU instances (such as c5 and m5) and GPU instances (such as p2 and p3) or both. For information about the instance types that provide varying combinations of CPU, GPU, memory, and networking capacity for inference, see Amazon SageMaker ML Instance Types.

Semantic Segmentation Hyperparameters

The following tables list the hyperparameters supported by the Amazon SageMaker semantic segmentation algorithm for network architecture, data inputs, and training. You specify Semantic Segmentation for training in the AlgorithmName of the CreateTrainingJob request.

Network Architecture Hyperparameters

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>backbone</td>
<td>The backbone to use for the algorithm's encoder component. Optional</td>
</tr>
</tbody>
</table>
### Use built-in algorithms

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Valid values: resnet-50, resnet-101</td>
</tr>
<tr>
<td></td>
<td>Default value: resnet-50</td>
</tr>
<tr>
<td>use_pretrained_model</td>
<td>Whether a pretrained model is to be used for the backbone.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: True, False</td>
</tr>
<tr>
<td></td>
<td>Default value: True</td>
</tr>
<tr>
<td>algorithm</td>
<td>The algorithm to use for semantic segmentation.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values:</td>
</tr>
<tr>
<td></td>
<td>• fcn: Fully-Convolutional Network (FCN) algorithm</td>
</tr>
<tr>
<td></td>
<td>• psp: Pyramid Scene Parsing (PSP) algorithm</td>
</tr>
<tr>
<td></td>
<td>• deeplab: DeepLab V3 algorithm</td>
</tr>
<tr>
<td></td>
<td>Default value: fcn</td>
</tr>
</tbody>
</table>

### Data Hyperparameters

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>num_classes</td>
<td>The number of classes to segment.</td>
</tr>
<tr>
<td></td>
<td><strong>Required</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: 2 ≤ positive integer ≤ 254</td>
</tr>
<tr>
<td>num_training_samples</td>
<td>The number of samples in the training data. The algorithm uses this value</td>
</tr>
<tr>
<td></td>
<td>to set up the learning rate scheduler.</td>
</tr>
<tr>
<td></td>
<td><strong>Required</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: positive integer</td>
</tr>
<tr>
<td>base_size</td>
<td>Defines how images are rescaled before cropping. Images are rescaled such</td>
</tr>
<tr>
<td></td>
<td>that the long size length is set to base_size multiplied by a random number</td>
</tr>
<tr>
<td></td>
<td>from 0.5 to 2.0, and the short size is computed to preserve the aspect ratio</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: positive integer &gt; 16</td>
</tr>
<tr>
<td></td>
<td>Default value: 520</td>
</tr>
<tr>
<td>crop_size</td>
<td>The image size for input during training. We randomly rescale the input</td>
</tr>
<tr>
<td></td>
<td>image based on base_size, and then take a random square crop with side</td>
</tr>
<tr>
<td></td>
<td>length equal to crop_size. The crop_size will be automatically rounded up</td>
</tr>
<tr>
<td></td>
<td>to multiples of 8.</td>
</tr>
</tbody>
</table>
### Training Hyperparameters

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>early_stopping</td>
<td>Whether to use early stopping logic during training.</td>
</tr>
<tr>
<td></td>
<td>Optional</td>
</tr>
<tr>
<td></td>
<td>Valid values: True, False</td>
</tr>
<tr>
<td></td>
<td>Default value: False</td>
</tr>
<tr>
<td>early_stopping_min_epochs</td>
<td>The minimum number of epochs that must be run.</td>
</tr>
<tr>
<td></td>
<td>Optional</td>
</tr>
<tr>
<td></td>
<td>Valid values: integer</td>
</tr>
<tr>
<td></td>
<td>Default value: 5</td>
</tr>
<tr>
<td>early_stopping_patience</td>
<td>The number of epochs that meet the tolerance for lower performance before the algorithm enforces an early stop.</td>
</tr>
<tr>
<td></td>
<td>Optional</td>
</tr>
<tr>
<td></td>
<td>Valid values: integer</td>
</tr>
<tr>
<td></td>
<td>Default value: 4</td>
</tr>
<tr>
<td>early_stopping_tolerance</td>
<td>If the relative improvement of the score of the training job, the mIOU, is smaller than this value, early stopping considers the epoch as not improved. This is used only when early_stopping = True.</td>
</tr>
<tr>
<td></td>
<td>Optional</td>
</tr>
<tr>
<td></td>
<td>Valid values: $0 \leq \text{float} \leq 1$</td>
</tr>
<tr>
<td></td>
<td>Default value: 0.0</td>
</tr>
<tr>
<td>epochs</td>
<td>The number of epochs with which to train.</td>
</tr>
<tr>
<td></td>
<td>Optional</td>
</tr>
<tr>
<td></td>
<td>Valid values: positive integer</td>
</tr>
<tr>
<td></td>
<td>Default value: 10</td>
</tr>
<tr>
<td>gamma1</td>
<td>The decay factor for the moving average of the squared gradient for rmsprop. Used only for rmsprop.</td>
</tr>
<tr>
<td></td>
<td>Optional</td>
</tr>
<tr>
<td>Parameter Name</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>gamma2</td>
<td>The momentum factor for rmsprop.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: $0 \leq \text{float} \leq 1</td>
</tr>
<tr>
<td></td>
<td>Default value: 0.9</td>
</tr>
<tr>
<td>learning_rate</td>
<td>The initial learning rate.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: $0 &lt; \text{float} \leq 1</td>
</tr>
<tr>
<td></td>
<td>Default value: 0.001</td>
</tr>
<tr>
<td>lr_scheduler</td>
<td>The shape of the learning rate schedule that controls its decrease over time.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values:</td>
</tr>
<tr>
<td></td>
<td>• <strong>step</strong>: A stepwise decay, where the learning rate is reduced (multiplied) by the lr_scheduler_factor after epochs specified by lr_scheduler_step.</td>
</tr>
<tr>
<td></td>
<td>• <strong>poly</strong>: A smooth decay using a polynomial function.</td>
</tr>
<tr>
<td></td>
<td>• <strong>cosine</strong>: A smooth decay using a cosine function.</td>
</tr>
<tr>
<td></td>
<td>Default value: poly</td>
</tr>
<tr>
<td>lr_scheduler_factor</td>
<td>If lr_scheduler is set to step, the ratio by which to reduce (multiply) the learning_rate after each of the epochs specified by the lr_scheduler_step. Otherwise, ignored.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: $0 \leq \text{float} \leq 1</td>
</tr>
<tr>
<td></td>
<td>Default value: 0.1</td>
</tr>
<tr>
<td>lr_scheduler_step</td>
<td>A comma delimited list of the epochs after which the learning_rate is reduced (multiplied) by an lr_scheduler_factor. For example, if the value is set to &quot;10, 20&quot;, then the learning-rate is reduced by lr_scheduler_factor after the 10th epoch and again by this factor after 20th epoch.</td>
</tr>
<tr>
<td></td>
<td><strong>Conditionally Required</strong> if lr_scheduler is set to step. Otherwise, ignored.</td>
</tr>
<tr>
<td></td>
<td>Valid values: string</td>
</tr>
<tr>
<td></td>
<td>Default value: (No default, as the value is required when used.)</td>
</tr>
<tr>
<td>Parameter Name</td>
<td>Description</td>
</tr>
<tr>
<td>----------------</td>
<td>-------------</td>
</tr>
</tbody>
</table>
| mini_batch_size | The batch size for training. Using a large mini_batch_size usually results in faster training, but it might cause you to run out of memory. Memory usage is affected by the values of the mini_batch_size and image_shape parameters, and the backbone architecture.  
**Optional**  
Valid values: positive integer  
Default value: 16 |
| momentum | The momentum for the sgd optimizer. When you use other optimizers, the semantic segmentation algorithm ignores this parameter.  
**Optional**  
Valid values: 0 < float ≤ 1  
Default value: 0.9 |
| optimizer | The type of optimizer. For more information about an optimizer, choose the appropriate link:  
- adam: Adaptive momentum estimation  
- adagrad: Adaptive gradient descent  
- nag: Nesterov accelerated gradient  
- rmsprop: Root mean square propagation  
- sgd: Stochastic gradient descent  
**Optional**  
Valid values: adam, adagrad, nag, rmsprop, sgd  
Default value: sgd |
| syncbn | If set to True, the batch normalization mean and variance are computed over all the samples processed across the GPUs.  
**Optional**  
Valid values: True, False  
Default value: False |
### Parameter Name

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
</table>
| **validation_mini_batch_size** | The batch size for validation. A large mini_batch_size usually results in faster training, but it might cause you to run out of memory. Memory usage is affected by the values of the mini_batch_size and image_shape parameters, and the backbone architecture.  

- To score the validation on the entire image without cropping the images, set this parameter to 1. Use this option if you want to measure performance on the entire image as a whole.  
  **Note**  
  Setting the validation_mini_batch_size parameter to 1 causes the algorithm to create a new network model for every image. This might slow validation and training.  

- To crop images to the size specified in the crop_size parameter, even during evaluation, set this parameter to a value greater than 1.  

  **Optional**  
  Valid values: positive integer  
  Default value: 16 |
| **weight_decay** | The weight decay coefficient for the sgd optimizer. When you use other optimizers, the algorithm ignores this parameter.  

  **Optional**  
  Valid values: 0 < float < 1  
  Default value: 0.0001 |

---

**Sequence-to-Sequence Algorithm**

Amazon SageMaker Sequence to Sequence is a supervised learning algorithm where the input is a sequence of tokens (for example, text, audio) and the output generated is another sequence of tokens. Example applications include: machine translation (input a sentence from one language and predict what that sentence would be in another language), text summarization (input a longer string of words and predict a shorter string of words that is a summary), speech-to-text (audio clips converted into output sentences in tokens). Recently, problems in this domain have been successfully modeled with deep neural networks that show a significant performance boost over previous methodologies. Amazon SageMaker seq2seq uses Recurrent Neural Networks (RNNs) and Convolutional Neural Network (CNN) models with attention as encoder-decoder architectures.

**Topics**

- Input/Output Interface for the Sequence-to-Sequence Algorithm (p. 802)
- EC2 Instance Recommendation for the Sequence-to-Sequence Algorithm (p. 803)
- Sequence-to-Sequence Sample Notebooks (p. 803)
- How Sequence-to-Sequence Works (p. 803)
- Sequence-to-Sequence Hyperparameters (p. 804)
- Tune a Sequence-to-Sequence Model (p. 811)
Input/Output Interface for the Sequence-to-Sequence Algorithm

Training

SageMaker seq2seq expects data in RecordIO-Protobuf format. However, the tokens are expected as integers, not as floating points, as is usually the case.

A script to convert data from tokenized text files to the protobuf format is included in the seq2seq example notebook. In general, it packs the data into 32-bit integer tensors and generates the necessary vocabulary files, which are needed for metric calculation and inference.

After preprocessing is done, the algorithm can be invoked for training. The algorithm expects three channels:

- **train**: It should contain the training data (for example, the train.rec file generated by the preprocessing script).
- **validation**: It should contain the validation data (for example, the val.rec file generated by the preprocessing script).
- **vocab**: It should contain two vocabulary files (vocab.src.json and vocab.trg.json)

If the algorithm doesn't find data in any of these three channels, training results in an error.

Inference

For hosted endpoints, inference supports two data formats. To perform inference using space separated text tokens, use the application/json format. Otherwise, use the recordio-protobuf format to work with the integer encoded data. Both mode supports batching of input data. application/json format also allows you to visualize the attention matrix.

- **application/json**: Expects the input in JSON format and returns the output in JSON format. Both content and accept types should be application/json. Each sequence is expected to be a string with whitespace separated tokens. This format is recommended when the number of source sequences in the batch is small. It also supports the following additional configuration options:

  configuration: {attention_matrix: true}: Returns the attention matrix for the particular input sequence.

- **application/x-recordio-protobuf**: Expects the input in recordio-protobuf format and returns the output in recordio-protobuf format. Both content and accept types should be applications/x-recordio-protobuf. For this format, the source sequences must be converted into a list of integers for subsequent protobuf encoding. This format is recommended for bulk inference.

For batch transform, inference supports JSON Lines format. Batch transform expects the input in JSON Lines format and returns the output in JSON Lines format. Both content and accept types should be application/jsonlines. The format for input is as follows:

```
content-type: application/jsonlines

{"source": "source_sequence_0"}
{"source": "source_sequence_1"}
```

The format for response is as follows:

```
accept: application/jsonlines
```
For additional details on how to serialize and deserialize the inputs and outputs to specific formats for inference, see the Sequence-to-Sequence Sample Notebooks (p. 803).

**EC2 Instance Recommendation for the Sequence-to-Sequence Algorithm**

Currently Amazon SageMaker seq2seq is only supported on GPU instance types and is only set up to train on a single machine. But it does also offer support for multiple GPUs.

**Sequence-to-Sequence Sample Notebooks**

For a sample notebook that shows how to use the SageMaker Sequence to Sequence algorithm to train a English-German translation model, see Machine Translation English-German Example Using SageMaker Seq2Seq. For instructions how to create and access Jupyter notebook instances that you can use to run the example in SageMaker, see Use Amazon SageMaker Notebook Instances (p. 124). Once you have created a notebook instance and opened it, select the SageMaker Examples tab to see a list of all the SageMaker samples. The topic modeling example notebooks using the NTM algorithms are located in the Introduction to Amazon algorithms section. To open a notebook, click on its Use tab and select Create copy.

**How Sequence-to-Sequence Works**

Typically, a neural network for sequence-to-sequence modeling consists of a few layers, including:

- **An embedding layer.** In this layer, the input matrix, which is input tokens encoded in a sparse way (for example, one-hot encoded) are mapped to a dense feature layer. This is required because a high-dimensional feature vector is more capable of encoding information regarding a particular token (word for text corpora) than a simple one-hot-encoded vector. It is also a standard practice to initialize this embedding layer with a pre-trained word vector like FastText or Glove or to initialize it randomly and learn the parameters during training.

- **An encoder layer.** After the input tokens are mapped into a high-dimensional feature space, the sequence is passed through an encoder layer to compress all the information from the input embedding layer (of the entire sequence) into a fixed-length feature vector. Typically, an encoder is made of RNN-type networks like long short-term memory (LSTM) or gated recurrent units (GRU). (Colah's blog explains LSTM in a great detail.)

- **A decoder layer.** The decoder layer takes this encoded feature vector and produces the output sequence of tokens. This layer is also usually built with RNN architectures (LSTM and GRU).

The whole model is trained jointly to maximize the probability of the target sequence given the source sequence. This model was first introduced by Sutskever et al. in 2014.

**Attention mechanism.** The disadvantage of an encoder-decoder framework is that model performance decreases as and when the length of the source sequence increases because of the limit of how much information the fixed-length encoded feature vector can contain. To tackle this problem, in 2015, Bahdanau et al. proposed the attention mechanism. In an attention mechanism, the decoder tries to find the location in the encoder sequence where the most important information could be located and uses that information and previously decoded words to predict the next token in the sequence.

For more in details, see the whitepaper Effective Approaches to Attention-based Neural Machine Translation by Luong, et al. that explains and simplifies calculations for various attention mechanisms. Additionally, the whitepaper Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation by Wu, et al. describes Google's architecture for machine translation, which uses skip connections between encoder and decoder layers.
## Sequence-to-Sequence Hyperparameters

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>batch_size</td>
<td>Mini batch size for gradient descent. Optional</td>
</tr>
<tr>
<td></td>
<td>Valid values: positive integer</td>
</tr>
<tr>
<td></td>
<td>Default value: 64</td>
</tr>
<tr>
<td>beam_size</td>
<td>Length of the beam for beam search. Used during training for computing bleu and used during inference. Optional</td>
</tr>
<tr>
<td></td>
<td>Valid values: positive integer</td>
</tr>
<tr>
<td></td>
<td>Default value: 5</td>
</tr>
<tr>
<td>bleu_sample_size</td>
<td>Number of instances to pick from validation dataset to decode and compute bleu score during training. Set to -1 to use full validation set (if bleu is chosen as optimized_metric). Optional</td>
</tr>
<tr>
<td></td>
<td>Valid values: integer</td>
</tr>
<tr>
<td></td>
<td>Default value: 0</td>
</tr>
<tr>
<td>bucket_width</td>
<td>Returns (source,target) buckets up to (max_seq_len_source, max_seq_len_target). The longer side of the data uses steps of bucket_width while the shorter side uses steps scaled down by the average target/source length ratio. If one sided reaches its maximum length before the other, width of extra buckets on that side is fixed to that side of max_len. Optional</td>
</tr>
<tr>
<td></td>
<td>Valid values: positive integer</td>
</tr>
<tr>
<td></td>
<td>Default value: 10</td>
</tr>
<tr>
<td>bucketing_enabled</td>
<td>Set to false to disable bucketing, unroll to maximum length. Optional</td>
</tr>
<tr>
<td></td>
<td>Valid values: true or false</td>
</tr>
<tr>
<td></td>
<td>Default value: true</td>
</tr>
<tr>
<td>checkpoint_frequency_num_batches</td>
<td>Checkpoint and evaluate every x batches. Optional</td>
</tr>
<tr>
<td></td>
<td>Valid values: positive integer</td>
</tr>
<tr>
<td>Parameter Name</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>checkpoint_threshold</td>
<td>Maximum number of checkpoints model is allowed to not improve in <code>optimized_metric</code> on validation dataset before training is stopped.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: positive integer</td>
</tr>
<tr>
<td></td>
<td>Default value: 3</td>
</tr>
<tr>
<td>clip_gradient</td>
<td>Clip absolute gradient values greater than this. Set to negative to disable.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: float</td>
</tr>
<tr>
<td></td>
<td>Default value: 1</td>
</tr>
<tr>
<td>cnn_activation_type</td>
<td>The <code>cnn</code> activation type to be used.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: String. One of glu, relu, softrelu, sigmoid, or tanh.</td>
</tr>
<tr>
<td></td>
<td>Default value: glu</td>
</tr>
<tr>
<td>cnn_hidden_dropout</td>
<td>Dropout probability for dropout between convolutional layers.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: Float. Range in [0,1].</td>
</tr>
<tr>
<td></td>
<td>Default value: 0</td>
</tr>
<tr>
<td>cnn_kernel_width_decoder</td>
<td>Kernel width for the <code>cnn</code> decoder.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: positive integer</td>
</tr>
<tr>
<td></td>
<td>Default value: 5</td>
</tr>
<tr>
<td>cnn_kernel_width_encoder</td>
<td>Kernel width for the <code>cnn</code> encoder.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: positive integer</td>
</tr>
<tr>
<td></td>
<td>Default value: 3</td>
</tr>
<tr>
<td>Parameter Name</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>cnn_num_hidden</code></td>
<td>Number of <code>cnn</code> hidden units for encoder and decoder.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: positive integer</td>
</tr>
<tr>
<td></td>
<td>Default value: 512</td>
</tr>
<tr>
<td><code>decoder_type</code></td>
<td>Decoder type.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: String. Either <code>rnn</code> or <code>cnn</code>.</td>
</tr>
<tr>
<td></td>
<td>Default value: <code>rnn</code></td>
</tr>
<tr>
<td><code>embed_dropout_source</code></td>
<td>Dropout probability for source side embeddings.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: Float. Range in [0,1].</td>
</tr>
<tr>
<td></td>
<td>Default value: 0</td>
</tr>
<tr>
<td><code>embed_dropout_target</code></td>
<td>Dropout probability for target side embeddings.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: Float. Range in [0,1].</td>
</tr>
<tr>
<td></td>
<td>Default value: 0</td>
</tr>
<tr>
<td><code>encoder_type</code></td>
<td>Encoder type. The <code>rnn</code> architecture is based on attention mechanism by Bahdanau et al. and <code>cnn</code> architecture is based on Gehring et al.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: String. Either <code>rnn</code> or <code>cnn</code>.</td>
</tr>
<tr>
<td></td>
<td>Default value: <code>rnn</code></td>
</tr>
<tr>
<td><code>fixed_rate_lr_half_life</code></td>
<td>Half life for learning rate in terms of number of checkpoints for <code>fixed_rate_*</code> schedulers.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: positive integer</td>
</tr>
<tr>
<td></td>
<td>Default value: 10</td>
</tr>
<tr>
<td><code>learning_rate</code></td>
<td>Initial learning rate.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: float</td>
</tr>
<tr>
<td></td>
<td>Default value: 0.0003</td>
</tr>
<tr>
<td>Parameter Name</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>loss_type</td>
<td>Loss function for training.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: String. cross-entropy</td>
</tr>
<tr>
<td></td>
<td>Default value: cross-entropy</td>
</tr>
<tr>
<td>lr_scheduler_type</td>
<td>Learning rate scheduler type. plateau_reduce means reduce the learning rate whenever optimized_metric on validation_accuracy plateaus. inv_t is inverse time decay. learning_rate/(1+decay_rate*t)</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: String. One of plateau_reduce, fixed_rate_inv_t, or fixed_rate_inv_sqrt_t.</td>
</tr>
<tr>
<td></td>
<td>Default value: plateau_reduce</td>
</tr>
<tr>
<td>max_num_batches</td>
<td>Maximum number of updates/batches to process. -1 for infinite.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: integer</td>
</tr>
<tr>
<td></td>
<td>Default value: -1</td>
</tr>
<tr>
<td>max_num_epochs</td>
<td>Maximum number of epochs to pass through training data before fitting is stopped. Training continues until this number of epochs even if validation accuracy is not improving if this parameter is passed. Ignored if not passed.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: Positive integer and less than or equal to max_num_epochs.</td>
</tr>
<tr>
<td></td>
<td>Default value: none</td>
</tr>
<tr>
<td>max_seq_len_source</td>
<td>Maximum length for the source sequence length. Sequences longer than this length are truncated to this length.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: positive integer</td>
</tr>
<tr>
<td></td>
<td>Default value: 100</td>
</tr>
<tr>
<td>max_seq_len_target</td>
<td>Maximum length for the target sequence length. Sequences longer than this length are truncated to this length.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: positive integer</td>
</tr>
<tr>
<td></td>
<td>Default value: 100</td>
</tr>
<tr>
<td>Parameter Name</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>min_num_epochs</td>
<td>Minimum number of epochs the training must run before it is stopped via early_stopping conditions.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: positive integer</td>
</tr>
<tr>
<td></td>
<td>Default value: 0</td>
</tr>
<tr>
<td>momentum</td>
<td>Momentum constant used for sgd. Don't pass this parameter if you are using adam or rmsprop.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: float</td>
</tr>
<tr>
<td></td>
<td>Default value: none</td>
</tr>
<tr>
<td>num_embed_source</td>
<td>Embedding size for source tokens.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: positive integer</td>
</tr>
<tr>
<td></td>
<td>Default value: 512</td>
</tr>
<tr>
<td>num_embed_target</td>
<td>Embedding size for target tokens.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: positive integer</td>
</tr>
<tr>
<td></td>
<td>Default value: 512</td>
</tr>
<tr>
<td>num_layers_decoder</td>
<td>Number of layers for Decoder \textit{rnn} or \textit{cnn}.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: positive integer</td>
</tr>
<tr>
<td></td>
<td>Default value: 1</td>
</tr>
<tr>
<td>num_layers_encoder</td>
<td>Number of layers for Encoder \textit{rnn} or \textit{cnn}.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: positive integer</td>
</tr>
<tr>
<td></td>
<td>Default value: 1</td>
</tr>
<tr>
<td>optimized_metric</td>
<td>Metrics to optimize with early stopping.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: String. One of perplexity, accuracy, or bleu.</td>
</tr>
<tr>
<td></td>
<td>Default value: perplexity</td>
</tr>
<tr>
<td>Parameter Name</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>optimizer_type</td>
<td>Optimizer to choose from.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: String. One of adam, sgd, or rmsprop.</td>
</tr>
<tr>
<td></td>
<td>Default value: adam</td>
</tr>
<tr>
<td>plateau_reduce_lr_factor</td>
<td>Factor to multiply learning rate with (for plateau_reduce).</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: float</td>
</tr>
<tr>
<td></td>
<td>Default value: 0.5</td>
</tr>
<tr>
<td>plateau_reduce_lr_threshold</td>
<td>For plateau_reduce scheduler, multiply learning rate with reduce factor if optimized_metric didn't improve for this many checkpoints.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: positive integer</td>
</tr>
<tr>
<td></td>
<td>Default value: 3</td>
</tr>
<tr>
<td>rnn_attention_in_upper_layers</td>
<td>Pass the attention to upper layers of rnn, like Google NMT paper. Only applicable if more than one layer is used.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: boolean (true or false)</td>
</tr>
<tr>
<td></td>
<td>Default value: true</td>
</tr>
<tr>
<td>rnn_attention_num_hidden</td>
<td>Number of hidden units for attention layers. defaults to rnn_num_hidden.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: positive integer</td>
</tr>
<tr>
<td></td>
<td>Default value: rnn_num_hidden</td>
</tr>
<tr>
<td>rnn_attention_type</td>
<td>Attention model for encoders. mlp refers to concat and bilinear refers to general from the Luong et al. paper.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: String. One of dot, fixed, mlp, or bilinear.</td>
</tr>
<tr>
<td></td>
<td>Default value: mlp</td>
</tr>
<tr>
<td>Parameter Name</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>rnn_cell_type</code></td>
<td>Specific type of <code>rnn</code> architecture.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: String. Either <code>lstm</code> or <code>gru</code>.</td>
</tr>
<tr>
<td></td>
<td>Default value: <code>lstm</code></td>
</tr>
<tr>
<td><code>rnn_decoder_state_init</code></td>
<td>How to initialize <code>rnn</code> decoder states from encoders.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: String. One of <code>last</code>, <code>avg</code>, or <code>zero</code>.</td>
</tr>
<tr>
<td></td>
<td>Default value: <code>last</code></td>
</tr>
<tr>
<td><code>rnn_first_residual_layer</code></td>
<td>First <code>rnn</code> layer to have a residual connection, only applicable if number of layers in encoder or decoder is more than 1.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: positive integer</td>
</tr>
<tr>
<td></td>
<td>Default value: 2</td>
</tr>
<tr>
<td><code>rnn_num_hidden</code></td>
<td>The number of <code>rnn</code> hidden units for encoder and decoder. This must be a multiple of 2 because the algorithm uses bi-directional <code>LSTM</code> by default.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: positive even integer</td>
</tr>
<tr>
<td></td>
<td>Default value: 1024</td>
</tr>
<tr>
<td><code>rnn_residual_connections</code></td>
<td>Add residual connection to stacked <code>rnn</code>. Number of layers should be more than 1.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: boolean (<code>true</code> or <code>false</code>)</td>
</tr>
<tr>
<td></td>
<td>Default value: <code>false</code></td>
</tr>
<tr>
<td><code>rnn_decoder_hidden_dropout</code></td>
<td>Dropout probability for hidden state that combines the context with the <code>rnn</code> hidden state in the decoder.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: Float. Range in [0,1].</td>
</tr>
<tr>
<td></td>
<td>Default value: 0</td>
</tr>
<tr>
<td>Parameter Name</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>training_metric</td>
<td>Metrics to track on training on validation data.</td>
</tr>
<tr>
<td>Optional</td>
<td>Valid values: String. Either perplexity or accuracy.</td>
</tr>
<tr>
<td>Default value:</td>
<td>perplexity</td>
</tr>
<tr>
<td>weight_decay</td>
<td>Weight decay constant.</td>
</tr>
<tr>
<td>Optional</td>
<td>Valid values: float</td>
</tr>
<tr>
<td>Default value:</td>
<td>0</td>
</tr>
<tr>
<td>weight_init_scale</td>
<td>Weight initialization scale (for uniform and xavier initialization).</td>
</tr>
<tr>
<td>Optional</td>
<td>Valid values: float</td>
</tr>
<tr>
<td>Default value:</td>
<td>2.34</td>
</tr>
<tr>
<td>weight_init_type</td>
<td>Type of weight initialization.</td>
</tr>
<tr>
<td>Optional</td>
<td>Valid values: String. Either uniform or xavier.</td>
</tr>
<tr>
<td>Default value:</td>
<td>xavier</td>
</tr>
<tr>
<td>xavier_factor_type</td>
<td>Xavier factor type.</td>
</tr>
<tr>
<td>Optional</td>
<td>Valid values: String. One of in, out, or avg.</td>
</tr>
<tr>
<td>Default value:</td>
<td>in</td>
</tr>
</tbody>
</table>

### Tune a Sequence-to-Sequence Model

*Automatic model tuning*, also known as hyperparameter tuning, finds the best version of a model by running many jobs that test a range of hyperparameters on your dataset. You choose the tunable hyperparameters, a range of values for each, and an objective metric. You choose the objective metric from the metrics that the algorithm computes. Automatic model tuning searches the hyperparameters chosen to find the combination of values that result in the model that optimizes the objective metric.

For more information about model tuning, see Perform Automatic Model Tuning (p. 1023).

### Metrics Computed by the Sequence-to-Sequence Algorithm

The sequence to sequence algorithm reports three metrics that are computed during training. Choose one of them as an objective to optimize when tuning the hyperparameter values.
## Use built-in algorithms

<table>
<thead>
<tr>
<th>Metric Name</th>
<th>Description</th>
<th>Optimization Direction</th>
</tr>
</thead>
<tbody>
<tr>
<td>validation:accuracy</td>
<td>Accuracy computed on the validation dataset.</td>
<td>Maximize</td>
</tr>
<tr>
<td>validation:bleu</td>
<td>Bleu score computed on the validation dataset. Because BLEU computation is expensive, you can choose to compute BLEU on a random subsample of the validation dataset to speed up the overall training process. Use the <code>bleu_sample_size</code> parameter to specify the subsample.</td>
<td>Maximize</td>
</tr>
<tr>
<td>validation:perplexity</td>
<td>Perplexity, is a loss function computed on the validation dataset. Perplexity measures the cross-entropy between an empirical sample and the distribution predicted by a model and so provides a measure of how well a model predicts the sample values, Models that are good at predicting a sample have a low perplexity.</td>
<td>Minimize</td>
</tr>
</tbody>
</table>

### Tunable Sequence-to-Sequence Hyperparameters

You can tune the following hyperparameters for the SageMaker Sequence to Sequence algorithm. The hyperparameters that have the greatest impact on sequence to sequence objective metrics are: `batch_size`, `optimizer_type`, `learning_rate`, `num_layers_encoder`, and `num_layers_decoder`.

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Parameter Type</th>
<th>Recommended Ranges</th>
</tr>
</thead>
<tbody>
<tr>
<td>num_layers_encoder</td>
<td>IntegerParameterRange</td>
<td>[1-10]</td>
</tr>
<tr>
<td>num_layers_decoder</td>
<td>IntegerParameterRange</td>
<td>[1-10]</td>
</tr>
<tr>
<td>batch_size</td>
<td>CategoricalParameterRange</td>
<td>[16,32,64,128,256,512,1024,2048]</td>
</tr>
<tr>
<td>optimizer_type</td>
<td>CategoricalParameterRange</td>
<td>[‘adam’, ‘sgd’, ‘rmsprop’]</td>
</tr>
<tr>
<td>weight_init_type</td>
<td>CategoricalParameterRange</td>
<td>[‘xavier’, ‘uniform’]</td>
</tr>
<tr>
<td>weight_init_scale</td>
<td>ContinuousParameterRange</td>
<td>For the <code>xavier</code> type: MinValue: 2.0, MaxValue: 3.0 For the <code>uniform</code> type: MinValue: -1.0, MaxValue: 1.0</td>
</tr>
<tr>
<td>learning_rate</td>
<td>ContinuousParameterRange</td>
<td>MinValue: 0.00005, MaxValue: 0.2</td>
</tr>
<tr>
<td>weight_decay</td>
<td>ContinuousParameterRange</td>
<td>MinValue: 0.0, MaxValue: 0.1</td>
</tr>
<tr>
<td>momentum</td>
<td>ContinuousParameterRange</td>
<td>MinValue: 0.5, MaxValue: 0.9</td>
</tr>
<tr>
<td>clip_gradient</td>
<td>ContinuousParameterRange</td>
<td>MinValue: 1.0, MaxValue: 5.0</td>
</tr>
<tr>
<td>rnn_num_hidden</td>
<td>CategoricalParameterRange</td>
<td>Applicable only to recurrent neural</td>
</tr>
</tbody>
</table>
### XGBoost Algorithm

The XGBoost (eXtreme Gradient Boosting) is a popular and efficient open-source implementation of the gradient boosted trees algorithm. Gradient boosting is a supervised learning algorithm that attempts to accurately predict a target variable by combining an ensemble of estimates from a set of simpler and weaker models. The XGBoost algorithm performs well in machine learning competitions because of its robust handling of a variety of data types, relationships, distributions, and the variety of hyperparameters that you can fine-tune. You can use XGBoost for regression, classification (binary and multiclass), and ranking problems.

You can use the new release of the XGBoost algorithm either as a Amazon SageMaker built-in algorithm or as a framework to run training scripts in your local environments. This implementation has a smaller memory footprint, better logging, improved hyperparameter validation, and an expanded set of metrics than the original versions. It provides an XGBoost estimator that executes a training script in a managed XGBoost environment. The current release of SageMaker XGBoost is based on the original XGBoost versions 0.90, 1.0, and 1.2.

#### Supported versions

- Framework (open source) mode: 0.90-1, 0.90-2, 1.0-1, 1.2-1
- Algorithm mode: 0.90-1, 0.90-2, 1.0-1, 1.2-1
Note
XGBoost 1.1 is not supported on SageMaker because XGBoost 1.1 has a broken capability to run prediction when the test input has fewer features than the training data in LIBSVM inputs. This capability has been restored in XGBoost 1.2. Consider using SageMaker XGBoost 1.2-1.

How to Use SageMaker XGBoost

With SageMaker, you can use XGBoost as a built-in algorithm or framework. By using XGBoost as a framework, you have more flexibility and access to more advanced scenarios, such as k-fold cross-validation, because you can customize your own training scripts.

- Use XGBoost as a framework

Use XGBoost as a framework to run your customized training scripts that can incorporate additional data processing into your training jobs. In the following code example, you can find how SageMaker Python SDK provides the XGBoost API as a framework in the same way it provides other framework APIs, such as TensorFlow, MXNet, and PyTorch.

SageMaker Python SDK v1

```python
import boto3
import sagemaker
from sagemaker.xgboost.estimator import XGBoost
from sagemaker.session import s3_input, Session

# initialize hyperparameters
hyperparameters = {
    "max_depth":"5",
    "eta":"0.2",
    "gamma":"4",
    "min_child_weight":"6",
    "subsample":"0.7",
    "verbose":"1",
    "objective":"reg:linear",
    "num_round":"50"
}

# set an output path where the trained model will be saved
bucket = sagemaker.Session().default_bucket()
prefix = 'DEMO-xgboost-as-a-framework'
output_path = 's3://{}/{}/{}/output'.format(bucket, prefix, 'abalone-xgb-framework')

# construct a SageMaker XGBoost estimator
# specify the entry_point to your xgboost training script
estimator = XGBoost(entry_point = "your_xgboost_abalone_script.py",
    framework_version='1.2-1',
    hyperparameters=hyperparameters,
    role=sagemaker.get_execution_role(),
    train_instance_count=1,
    train_instance_type='ml.m5.2xlarge',
    output_path=output_path)

# define the data type and paths to the training and validation datasets
content_type = "libsvm"
train_input = s3_input("s3://{}/{}/{}".format(bucket, prefix, 'train'),
    content_type=content_type)
validation_input = s3_input("s3://{}/{}/{}".format(bucket, prefix, 'validation'),
    content_type=content_type)

# execute the XGBoost training job
estimator.fit({'train': train_input, 'validation': validation_input})
```

SageMaker Python SDK v2

```python
import boto3
import sagemaker
from sagemaker.xgboost.estimator import XGBoost
from sagemaker.session import Session
from sagemaker.inputs import TrainingInput

# initialize hyperparameters
hyperparameters = {
    "max_depth":"5",
    "eta":"0.2",
    "gamma":"4",
    "min_child_weight":"6",
    "subsample":"0.7",
    "verbose":"1",
    "objective":"reg:linear",
    "num_round":"50"}

# set an output path where the trained model will be saved
bucket = sagemaker.Session().default_bucket()
prefix = 'DEMO-xgboost-as-a-framework'
output_path = 's3://{}/{}/{}/output'.format(bucket, prefix, 'abalone-xgb-framework')

# construct a SageMaker XGBoost estimator
estimator = XGBoost(entry_point = "your_xgboost_abalone_script.py",
                     framework_version='1.2-1',
                     hyperparameters=hyperparameters,
                     role=sagemaker.get_execution_role(),
                     instance_count=1,
                     instance_type='ml.m5.2xlarge',
                     output_path=output_path)

# define the data type and paths to the training and validation datasets
content_type = "libsvm"
train_input = TrainingInput("s3://{}/{}/{}/".format(bucket, prefix, 'train'),
                           content_type=content_type)
validation_input = TrainingInput("s3://{}/{}/{}/".format(bucket, prefix, 'validation'),
                                 content_type=content_type)

# execute the XGBoost training job
estimator.fit({'train': train_input, 'validation': validation_input})
```

For an end-to-end example of using SageMaker XGBoost as a framework, see Regression with Amazon SageMaker XGBoost

- **Use XGBoost as a built-in algorithm**

Use the XGBoost built-in algorithm to build an XGBoost training container as shown in the following code example. You can automatically spot the XGBoost built-in algorithm image URI using the SageMaker image_uris.retrieve API (or the get_image_uri API if using Amazon SageMaker Python SDK version 1). If you want to ensure if the image_uris.retrieve API finds the correct URI, see Common parameters for built-in algorithms and look up xgboost from the full list of built-in algorithm image URIs and available regions.

After specifying the XGBoost image URI, you can use the XGBoost container to construct an estimator using the SageMaker Estimator API and initiate a training job. This XGBoost built-in algorithm mode does not incorporate your own XGBoost training script and runs directly on the input datasets.
SageMaker Python SDK v1

```python
import sagemaker
import boto3
from sagemaker.amazon.amazon_estimator import get_image_uri
from sagemaker.session import Session

# initialize hyperparameters
hyperparameters = {
    "max_depth":"5",
    "eta":"0.2",
    "gamma":"4",
    "min_child_weight":"6",
    "subsample":"0.7",
    "objective":"reg:squarederror",
    "num_round":50"
}

# set an output path where the trained model will be saved
bucket = sagemaker.Session().default_bucket()
prefix = 'DEMO-xgboost-as-a-built-in-algo'
output_path = 's3://{}/{}/{}', output'.format(bucket, prefix, 'abalone-xgb-built-in-algo')

# this line automatically looks for the XGBoost image URI and builds an XGBoost container.
# specify the repo_version depending on your preference.
xgboost_container = get_image_uri(boto3.Session().region_name, 'xgboost', repo_version='1.2-1')

# construct a SageMaker estimator that calls the xgboost-container
estimator = sagemaker.estimator.Estimator(image_name=xgboost_container,
                                          hyperparameters=hyperparameters,
                                          role=sagemaker.get_execution_role(),
                                          train_instance_count=1,
                                          train_instance_type='ml.m5.2xlarge',
                                          train_volume_size=5, # 5 GB
                                          output_path=output_path)

# define the data type and paths to the training and validation datasets
content_type = "libsvm"
train_input = s3_input("s3://{}/{}/{}' format(bucket, prefix, 'train'),
validation_input = s3_input("s3://{}/{}/{}' format(bucket, prefix, 'validation'),

# execute the XGBoost training job
estimator.fit({'train': train_input, 'validation': validation_input})
```

SageMaker Python SDK v2

```python
import sagemaker
import boto3
from sagemaker import image_uris
from sagemaker import inputs
from sagemaker import session

# initialize hyperparameters
hyperparameters = {
    "max_depth":5,
    "eta":0.2,
    "gamma":4,
    "min_child_weight":6,
    "subsample":0.7,
    "objective":"reg:squarederror",
    "num_round":50"
}

# set an output path where the trained model will be saved
bucket = sagemaker.Session().default_bucket()
prefix = 'DEMO-xgboost-as-a-built-in-algo'
output_path = 's3://{}/{}/{}', output'.format(bucket, prefix, 'abalone-xgb-built-in-algo')

# this line automatically looks for the XGBoost image URI and builds an XGBoost container.
# specify the repo_version depending on your preference.
xgboost_container = get_image_uri(boto3.Session().region_name, 'xgboost', repo_version='1.2-1')

# construct a SageMaker estimator that calls the xgboost-container
estimator = sagemaker.estimator.Estimator(image=Image(image=xgboost_container,
                                          hyperparameters=hyperparameters,
                                          role=sagemaker.get_execution_role(),
                                          train_instance_count=1,
                                          train_instance_type='ml.m5.2xlarge',
                                          train_volume_size=5, # 5 GB
                                          output_path=output_path)

# define the data type and paths to the training and validation datasets
content_type = "libsvm"
train_input = s3_input("s3://{}/{}/{}' format(bucket, prefix, 'train'),
validation_input = s3_input("s3://{}/{}/{}' format(bucket, prefix, 'validation'),

# execute the XGBoost training job
estimator.fit({'train': train_input, 'validation': validation_input})
```
"subsample":"0.7",
"objective":"reg:squarederror",
"num_round":"50"}

# set an output path where the trained model will be saved
bucket = sagemaker.Session().default_bucket()
prefix = 'DEMO-xgboost-as-a-built-in-algo'
output_path = 's3://{}/{}/{}/output'.format(bucket, prefix, 'abalone-xgb-built-in-algo')

# this line automatically looks for the XGBoost image URI and builds an XGBoost container.
# specify the repo_version depending on your preference.
xgboost_container = sagemaker.image_uris.retrieve("xgboost", region, "1.2-1")

# construct a SageMaker estimator that calls the xgboost-container
estimator = sagemaker.estimator.Estimator(image_uri=xgboost_container,
hyperparameters=hyperparameters,
role=sagemaker.get_execution_role(),
instance_count=1,
inference_volumes=5, # 5 GB
output_path=output_path)

# define the data type and paths to the training and validation datasets
content_type = "libsvm"
train_input = TrainingInput("s3://{}/{}/{}/train".format(bucket, prefix, content_type=content_type)
validation_input = TrainingInput("s3://{}/{}/{}/validation".format(bucket, prefix, 'validation', content_type=content_type)

# execute the XGBoost training job
estimator.fit({'train': train_input, 'validation': validation_input})

For more information about how to set up the XGBoost as a built-in algorithm, see the following notebook examples.
- Managed Spot Training for XGBoost
- Regression with Amazon SageMaker XGBoost (Parquet input)

Input/Output Interface for the XGBoost Algorithm

Gradient boosting operates on tabular data, with the rows representing observations, one column representing the target variable or label, and the remaining columns representing features.

The SageMaker implementation of XGBoost supports CSV and libsvm formats for training and inference:

- For Training ContentType, valid inputs are text/libsvm (default) or text/csv.
- For Inference ContentType, valid inputs are text/libsvm (default) or text/csv.

Note
For CSV training, the algorithm assumes that the target variable is in the first column and that the CSV does not have a header record.
For CSV inference, the algorithm assumes that CSV input does not have the label column.
For libsvm training, the algorithm assumes that the label is in the first column. Subsequent columns contain the zero-based index value pairs for features. So each row has the format: <label> <index0>:<value0> <index1>:<value1> ... Inference requests for libsvm might not have labels in the libsvm format.
This differs from other SageMaker algorithms, which use the protobuf training input format to maintain greater consistency with standard XGBoost data formats.

For CSV training input mode, the total memory available to the algorithm (Instance Count * the memory available in the InstanceType) must be able to hold the training dataset. For libsvm training input mode, it's not required, but we recommend it.

SageMaker XGBoost uses the Python pickle module to serialize/deserialize the model, which can be used for saving/loading the model.

To use a model trained with SageMaker XGBoost in open source XGBoost

- Use the following Python code:

```python
import pickle as pkl
import tarfile
import xgboost

t = tarfile.open('model.tar.gz', 'r:gz')
t.extractall()

model = pkl.load(open(model_file_path, 'rb'))

# prediction with test data
pred = model.predict(dtest)
```

To differentiate the importance of labelled data points use Instance Weight Supports

- SageMaker XGBoost allows customers to differentiate the importance of labelled data points by assigning each instance a weight value. For text/libsvm input, customers can assign weight values to data instances by attaching them after the labels. For example, `label:weight idx_0:val_0 idx_1:val_1...`. For text/csv input, customers need to turn on the `csv_weights` flag in the parameters and attach weight values in the column after labels. For example: `label,weight,val_0,val_1,...`.

EC2 Instance Recommendation for the XGBoost Algorithm

SageMaker XGBoost 1.0-1 or earlier currently only trains using CPUs. It is a memory-bound (as opposed to compute-bound) algorithm. So, a general-purpose compute instance (for example, M5) is a better choice than a compute-optimized instance (for example, C4). Further, we recommend that you have enough total memory in selected instances to hold the training data. Although it supports the use of disk space to handle data that does not fit into main memory (the out-of-core feature available with the libsvm input mode), writing cache files onto disk slows the algorithm processing time.

SageMaker XGBoost version 1.2 or later supports single-instance GPU training. Despite higher per-instance costs, GPUs train more quickly, making them more cost effective. To take advantage of GPU training, specify the instance type as one of the GPU instances (for example, P3) and set the `tree_method` hyperparameter to `gpu_hist` in your existing XGBoost script. SageMaker currently does not support multi-GPU training.

XGBoost Sample Notebooks

The following table outlines a variety of sample notebooks that address different use cases of Amazon SageMaker XGBoost algorithm.
### Notebook Title | Description
--- | ---
How to Create a Custom XGBoost container?  | This notebook shows you how to build a custom XGBoost Container with Amazon SageMaker Batch Transform.
An Introduction to Feature Processing, Training and Inference  | This notebook shows you how to build a Machine Learning (ML) Pipeline using Spark Feature Transformers and do real-time inference using Amazon SageMaker Batch Transform.
Regression with XGBoost using Parquet  | This notebook shows you how to use the Abalone dataset in Parquet to train a XGBoost model.
How to Train and Host a Multiclass Classification Model?  | This notebook shows how to use the MNIST dataset to train and host a multiclass classification model.
How to train a Model for Customer Churn Prediction?  | This notebook shows you how to train a model to Predict Mobile Customer Departure in an effort to identify unhappy customers.
An Introduction to Amazon SageMaker Managed Spot infrastructure for XGBoost Training  | This notebook shows you how to use Spot Instances for training with a XGBoost Container.
How to use Amazon SageMaker Debugger to debug XGBoost Training Jobs?  | This notebook shows you how to use Amazon SageMaker Debugger to monitor training jobs to detect inconsistencies.
How to use Amazon SageMaker Debugger to debug XGBoost Training Jobs in Real-Time?  | This notebook shows you how to use the MNIST dataset and Amazon SageMaker Debugger to perform real-time analysis of XGBoost training jobs while training jobs are running.

For instructions on how to create and access Jupyter notebook instances that you can use to run the example in SageMaker, see Use Amazon SageMaker Notebook Instances (p. 124). After you have created a notebook instance and opened it, choose the SageMaker Examples tab to see a list of all of the SageMaker samples. The topic modeling example notebooks using the linear learning algorithm are located in the Introduction to Amazon algorithms section. To open a notebook, choose its Use tab and choose Create copy.

**How XGBoost Works**

**XGBoost** is a popular and efficient open-source implementation of the gradient boosted trees algorithm. Gradient boosting is a supervised learning algorithm, which attempts to accurately predict a target variable by combining the estimates of a set of simpler, weaker models.

When using gradient boosting for regression, the weak learners are regression trees, and each regression tree maps an input data point to one of its leafs that contains a continuous score. XGBoost minimizes a regularized (L1 and L2) objective function that combines a convex loss function (based on the difference between the predicted and target outputs) and a penalty term for model complexity (in other words, the regression tree functions). The training proceeds iteratively, adding new trees that predict the residuals or errors of prior trees that are then combined with previous trees to make the final prediction. It's called gradient boosting because it uses a gradient descent algorithm to minimize the loss when adding new models.

For more detail on XGBoost, see:
XGBoost Hyperparameters

The following table contains the subset of hyperparameters that are required or most commonly used for the Amazon SageMaker XGBoost algorithm. These are parameters that are set by users to facilitate the estimation of model parameters from data. The required hyperparameters that must be set are listed first, in alphabetical order. The optional hyperparameters that can be set are listed next, also in alphabetical order. The SageMaker XGBoost algorithm is an implementation of the open-source DMLC XGBoost package. Currently SageMaker supports version 0.90. For details about full set of hyperparameter that can be configured for this version of XGBoost, see XGBoost Parameters.

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>num_class</td>
<td>The number of classes.</td>
</tr>
<tr>
<td></td>
<td><strong>Required</strong> if objective is set to multi:softmax or multi:softprob.</td>
</tr>
<tr>
<td></td>
<td>Valid values: integer</td>
</tr>
<tr>
<td>num_round</td>
<td>The number of rounds to run the training.</td>
</tr>
<tr>
<td></td>
<td><strong>Required</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: integer</td>
</tr>
<tr>
<td>alpha</td>
<td>L1 regularization term on weights. Increasing this value makes models more</td>
</tr>
<tr>
<td></td>
<td>conservative.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: float</td>
</tr>
<tr>
<td></td>
<td>Default value: 0</td>
</tr>
<tr>
<td>base_score</td>
<td>The initial prediction score of all instances, global bias.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: float</td>
</tr>
<tr>
<td></td>
<td>Default value: 0.5</td>
</tr>
<tr>
<td>booster</td>
<td>Which booster to use. The gbtree and dart values use a tree-based model,</td>
</tr>
<tr>
<td></td>
<td>while gblinear uses a linear function.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: String. One of gbtree, gblinear, or dart.</td>
</tr>
<tr>
<td></td>
<td>Default value: gbtree</td>
</tr>
<tr>
<td>(colsample_bylevel)</td>
<td>Subsample ratio of columns for each split, in each level.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: Float. Range: [0,1].</td>
</tr>
<tr>
<td>Parameter Name</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td></td>
<td>Default value: 1</td>
</tr>
<tr>
<td><strong>colsample_bynode</strong></td>
<td>Subsample ratio of columns from each node.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: Float. Range: (0,1].</td>
</tr>
<tr>
<td></td>
<td>Default value: 1</td>
</tr>
<tr>
<td><strong>colsample_bytree</strong></td>
<td>Subsample ratio of columns when constructing each tree.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: Float. Range: [0,1].</td>
</tr>
<tr>
<td></td>
<td>Default value: 1</td>
</tr>
<tr>
<td><strong>csv_weights</strong></td>
<td>When this flag is enabled, XGBoost differentiates the importance of instances for csv input by taking the second column (the column after labels) in training data as the instance weights.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: 0 or 1</td>
</tr>
<tr>
<td></td>
<td>Default value: 0</td>
</tr>
<tr>
<td><strong>early_stopping_rounds</strong></td>
<td>The model trains until the validation score stops improving. Validation error needs to decrease at least every <code>early_stopping_rounds</code> to continue training. SageMaker hosting uses the best model for inference.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: integer</td>
</tr>
<tr>
<td></td>
<td>Default value: -</td>
</tr>
<tr>
<td><strong>eta</strong></td>
<td>Step size shrinkage used in updates to prevent overfitting. After each boosting step, you can directly get the weights of new features. The <code>eta</code> parameter actually shrinks the feature weights to make the boosting process more conservative.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: Float. Range: [0,1].</td>
</tr>
<tr>
<td></td>
<td>Default value: 0.3</td>
</tr>
<tr>
<td>Parameter Name</td>
<td>Description</td>
</tr>
<tr>
<td>----------------</td>
<td>-------------</td>
</tr>
</tbody>
</table>
| **eval_metric** | Evaluation metrics for validation data. A default metric is assigned according to the objective:  
  - *rmse*: for regression  
  - *error*: for classification  
  - *map*: for ranking  
  For a list of valid inputs, see XGBoost Learning Task Parameters.  
  Optional  
  Valid values: string  
  Default value: Default according to objective. |
| **gamma** | Minimum loss reduction required to make a further partition on a leaf node of the tree. The larger, the more conservative the algorithm is.  
  Optional  
  Valid values: Float. Range: [0,∞).  
  Default value: 0 |
| **grow_policy** | Controls the way that new nodes are added to the tree. Currently supported only if tree_method is set to hist.  
  Optional  
  Valid values: String. Either depthwise or lossguide.  
  Default value: depthwise |
| **lambda** | L2 regularization term on weights. Increasing this value makes models more conservative.  
  Optional  
  Valid values: float  
  Default value: 1 |
| **lambda_bias** | L2 regularization term on bias.  
  Optional  
  Valid values: Float. Range: [0.0, 1.0].  
  Default value: 0 |
<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>max_bin</td>
<td>Maximum number of discrete bins to bucket continuous features. Used only if <code>tree_method</code> is set to <code>hist</code>.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: integer</td>
</tr>
<tr>
<td></td>
<td>Default value: 256</td>
</tr>
<tr>
<td>max_delta_step</td>
<td>Maximum delta step allowed for each tree's weight estimation. When a positive integer is used, it helps make the update more conservative. The preferred option is to use it in logistic regression. Set it to 1-10 to help control the update.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: Integer. Range: ([0, \infty)).</td>
</tr>
<tr>
<td></td>
<td>Default value: 0</td>
</tr>
<tr>
<td>max_depth</td>
<td>Maximum depth of a tree. Increasing this value makes the model more complex and likely to be overfit. 0 indicates no limit. A limit is required when <code>grow_policy</code>=depth-wise.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: Integer. Range: ([0, \infty))</td>
</tr>
<tr>
<td></td>
<td>Default value: 6</td>
</tr>
<tr>
<td>max_leaves</td>
<td>Maximum number of nodes to be added. Relevant only if <code>grow_policy</code> is set to <code>lossguide</code>.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: integer</td>
</tr>
<tr>
<td></td>
<td>Default value: 0</td>
</tr>
<tr>
<td>min_child_weight</td>
<td>Minimum sum of instance weight (hessian) needed in a child. If the tree partition step results in a leaf node with the sum of instance weight less than <code>min_child_weight</code>, the building process gives up further partitioning. In linear regression models, this simply corresponds to a minimum number of instances needed in each node. The larger the algorithm, the more conservative it is.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: Float. Range: ([0, \infty)).</td>
</tr>
<tr>
<td></td>
<td>Default value: 1</td>
</tr>
<tr>
<td>normalize_type</td>
<td>Type of normalization algorithm.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: Either <code>tree</code> or <code>forest</code>.</td>
</tr>
<tr>
<td></td>
<td>Default value: <code>tree</code></td>
</tr>
<tr>
<td>Parameter Name</td>
<td>Description</td>
</tr>
<tr>
<td>----------------</td>
<td>-------------</td>
</tr>
</tbody>
</table>
| nthread        | Number of parallel threads used to run xgboost.  
|                | Optional    
|                | Valid values: integer  
|                | Default value: Maximum number of threads. |
| objective      | Specifies the learning task and the corresponding learning objective. Examples: reg:logistic, multi:softmax, reg:squarederror. For a full list of valid inputs, refer to XGBoost Learning Task Parameters.  
|                | Optional    
|                | Valid values: string  
|                | Default value: reg:squarederror |
| one_drop       | When this flag is enabled, at least one tree is always dropped during the dropout.  
|                | Optional    
|                | Valid values: 0 or 1  
|                | Default value: 0 |
| process_type   | The type of boosting process to run.  
|                | Optional    
|                | Valid values: String. Either default or update.  
|                | Default value: default |
| rate_drop      | The dropout rate that specifies the fraction of previous trees to drop during the dropout.  
|                | Optional    
|                | Valid values: Float. Range: [0.0, 1.0].  
|                | Default value: 0.0 |
| refresh_leaf   | This is a parameter of the 'refresh' updater plug-in. When set to true (1), tree leaves and tree node stats are updated. When set to false(0), only tree node stats are updated.  
|                | Optional    
|                | Valid values: 0/1  
<p>|                | Default value: 1 |</p>
<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sample_type</td>
<td>Type of sampling algorithm.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: Either uniform or weighted.</td>
</tr>
<tr>
<td></td>
<td>Default value: uniform</td>
</tr>
<tr>
<td>scale_pos_weight</td>
<td>Controls the balance of positive and negative weights. It's useful for unbalanced classes. A typical value to consider: ( \text{sum(negative cases)} / \text{sum(positive cases)} ).</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: float</td>
</tr>
<tr>
<td></td>
<td>Default value: 1</td>
</tr>
<tr>
<td>seed</td>
<td>Random number seed.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: integer</td>
</tr>
<tr>
<td></td>
<td>Default value: 0</td>
</tr>
<tr>
<td>silent</td>
<td>0 means print running messages, 1 means silent mode.</td>
</tr>
<tr>
<td></td>
<td>Valid values: 0 or 1</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Default value: 0</td>
</tr>
<tr>
<td>sketch_eps</td>
<td>Used only for approximate greedy algorithm. This translates into ( O(1 / \text{sketch_eps}) ) number of bins. Compared to directly select number of bins, this comes with theoretical guarantee with sketch accuracy.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: Float, Range: [0, 1].</td>
</tr>
<tr>
<td></td>
<td>Default value: 0.03</td>
</tr>
<tr>
<td>skip_drop</td>
<td>Probability of skipping the dropout procedure during a boosting iteration.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: Float, Range: [0.0, 1.0].</td>
</tr>
<tr>
<td></td>
<td>Default value: 0.0</td>
</tr>
<tr>
<td>Parameter Name</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>subsample</td>
<td>Subsample ratio of the training instance. Setting it to 0.5 means that XGBoost randomly collects half of the data instances to grow trees. This prevents overfitting.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: Float. Range: [0, 1].</td>
</tr>
<tr>
<td></td>
<td>Default value: 1</td>
</tr>
<tr>
<td>tree_method</td>
<td>The tree construction algorithm used in XGBoost.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: One of <code>auto</code>, <code>exact</code>, <code>approx</code>, <code>hist</code>, or <code>gpu_hist</code>.</td>
</tr>
<tr>
<td></td>
<td>Default value: <code>auto</code></td>
</tr>
<tr>
<td>tweedie_variance_power</td>
<td>Parameter that controls the variance of the Tweedie distribution.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: Float. Range: (1, 2).</td>
</tr>
<tr>
<td></td>
<td>Default value: 1.5</td>
</tr>
<tr>
<td>updater</td>
<td>A comma-separated string that defines the sequence of tree updaters to run. This provides a modular way to construct and to modify the trees.</td>
</tr>
<tr>
<td></td>
<td>For a full list of valid inputs, please refer to XGBoost Parameters.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: comma-separated string.</td>
</tr>
<tr>
<td></td>
<td>Default value: <code>grow_colmaker</code>, <code>prune</code></td>
</tr>
<tr>
<td>single_precision_histogram</td>
<td>When this flag is enabled, XGBoost uses single precision to build histograms instead of double precision. Used only if <code>tree_method</code> is set to <code>hist</code> or <code>gpu_hist</code>.</td>
</tr>
<tr>
<td></td>
<td>For a full list of valid inputs, please refer to XGBoost Parameters.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: String. Range: <code>true</code> or <code>false</code></td>
</tr>
<tr>
<td></td>
<td>Default value: <code>false</code></td>
</tr>
</tbody>
</table>
Use built-in algorithms

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>deterministic_histogram</td>
<td>When this flag is enabled, XGBoost builds histogram on GPU deterministically. Used only if tree_method is set to gpu_hist. For a full list of valid inputs, please refer to XGBoost Parameters.</td>
</tr>
<tr>
<td>Optional</td>
<td>Valid values: String. Range: true or false</td>
</tr>
<tr>
<td>Default value</td>
<td>true</td>
</tr>
</tbody>
</table>

**Tune an XGBoost Model**

*Automated model tuning*, also known as hyperparameter tuning, finds the best version of a model by running many jobs that test a range of hyperparameters on your training and validation datasets. You choose three types of hyperparameters:

- a learning *objective* function to optimize during model training
- an *eval_metric* to use to evaluate model performance during validation
- a set of hyperparameters and a range of values for each to use when tuning the model automatically

You choose the evaluation metric from set of evaluation metrics that the algorithm computes. Automated model tuning searches the hyperparameters chosen to find the combination of values that result in the model that optimizes the evaluation metric.

**Note**

Automated model tuning for XGBoost 0.90 is only available from the Amazon SageMaker SDKs, not from the SageMaker console.

For more information about model tuning, see Perform Automatic Model Tuning (p. 1023).

**Evaluation Metrics Computed by the XGBoost Algorithm**

The XGBoost algorithm computes the following metrics to use for model validation. When tuning the model, choose one of these metrics to evaluate the model. For full list of valid eval_metric values, refer to XGBoost Learning Task Parameters

<table>
<thead>
<tr>
<th>Metric Name</th>
<th>Description</th>
<th>Optimization Direction</th>
</tr>
</thead>
<tbody>
<tr>
<td>validation:accuracy</td>
<td>Classification rate, calculated as #(right)/#(all cases).</td>
<td>Maximize</td>
</tr>
<tr>
<td>validation:auc</td>
<td>Area under the curve.</td>
<td>Maximize</td>
</tr>
<tr>
<td>validation: error</td>
<td>Binary classification error rate, calculated as #(wrong cases)/#(all cases).</td>
<td>Minimize</td>
</tr>
<tr>
<td>validation: f1</td>
<td>Indicator of classification accuracy, calculated as the harmonic mean of precision and recall.</td>
<td>Maximize</td>
</tr>
<tr>
<td>validation: logloss</td>
<td>Negative log-likelihood.</td>
<td>Minimize</td>
</tr>
<tr>
<td>validation: mae</td>
<td>Mean absolute error.</td>
<td>Minimize</td>
</tr>
<tr>
<td>validation: map</td>
<td>Mean average precision.</td>
<td>Maximize</td>
</tr>
<tr>
<td>Metric Name</td>
<td>Description</td>
<td>Optimization Direction</td>
</tr>
<tr>
<td>-------------</td>
<td>------------------------------------------------------------------------------</td>
<td>-------------------------</td>
</tr>
<tr>
<td>validation:merror</td>
<td>Multiclass classification error rate, calculated as #(wrong cases)/#(all cases).</td>
<td>Minimize</td>
</tr>
<tr>
<td>validation:mlogloss</td>
<td>Negative log-likelihood for multiclass classification.</td>
<td>Minimize</td>
</tr>
<tr>
<td>validation:mse</td>
<td>Mean squared error.</td>
<td>Minimize</td>
</tr>
<tr>
<td>validation:ndcg</td>
<td>Normalized Discounted Cumulative Gain.</td>
<td>Maximize</td>
</tr>
<tr>
<td>validation:rmse</td>
<td>Root mean square error.</td>
<td>Minimize</td>
</tr>
</tbody>
</table>

**Tunable XGBoost Hyperparameters**

Tune the XGBoost model with the following hyperparameters. The hyperparameters that have the greatest effect on optimizing the XGBoost evaluation metrics are: alpha, min_child_weight, subsample, eta, and num_round.

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Parameter Type</th>
<th>Recommended Ranges</th>
</tr>
</thead>
<tbody>
<tr>
<td>alpha</td>
<td>ContinuousParameterRanges</td>
<td>MinValue: 0, MaxValue: 1000</td>
</tr>
<tr>
<td>colsample_bylevel</td>
<td>ContinuousParameterRanges</td>
<td>MinValue: 0.1, MaxValue: 1</td>
</tr>
<tr>
<td>colsample_bynode</td>
<td>ContinuousParameterRanges</td>
<td>MinValue: 0.1, MaxValue: 1</td>
</tr>
<tr>
<td>colsample_bytree</td>
<td>ContinuousParameterRanges</td>
<td>MinValue: 0.5, MaxValue: 1</td>
</tr>
<tr>
<td>eta</td>
<td>ContinuousParameterRanges</td>
<td>MinValue: 0.1, MaxValue: 0.5</td>
</tr>
<tr>
<td>gamma</td>
<td>ContinuousParameterRanges</td>
<td>MinValue: 0, MaxValue: 5</td>
</tr>
<tr>
<td>lambda</td>
<td>ContinuousParameterRanges</td>
<td>MinValue: 0, MaxValue: 1000</td>
</tr>
<tr>
<td>max_delta_step</td>
<td>IntegerParameterRanges</td>
<td>[0, 10]</td>
</tr>
<tr>
<td>max_depth</td>
<td>IntegerParameterRanges</td>
<td>[0, 10]</td>
</tr>
<tr>
<td>min_child_weight</td>
<td>ContinuousParameterRanges</td>
<td>MinValue: 0, MaxValue: 120</td>
</tr>
<tr>
<td>num_round</td>
<td>IntegerParameterRanges</td>
<td>[1, 4000]</td>
</tr>
<tr>
<td>subsample</td>
<td>ContinuousParameterRanges</td>
<td>MinValue: 0.5, MaxValue: 1</td>
</tr>
</tbody>
</table>

**XGBoost Previous Versions**

This page contains links to the documentation for previous versions of Amazon SageMaker XGBoost.
Use built-in algorithms

Topics
- XGBoost Version 0.72 (p. 829)

XGBoost Version 0.72

Important
The XGBoost 0.72 is deprecated by Amazon SageMaker. You can still use this old version of XGBoost (as a built-in algorithm) by pulling its image URI as shown in the following code sample. For XGBoost, the image URI ending with :1 is for the old version.

SageMaker Python SDK v1

```python
import boto3
from sagemaker.amazon.amazon_estimator import get_image_uri

xgb_image_uri = get_image_uri(boto3.Session().region_name, "xgboost", repo_version="1")
```

SageMaker Python SDK v2

```python
import boto3
from sagemaker import image_uris

xgb_image_uri = image_uris.retrieve("xgboost", boto3.Session().region_name, "1")
```

If you want to use newer versions, you have to explicitly specify the image URI tags (see Supported versions (p. 813)).

This previous release of the Amazon SageMaker XGBoost algorithm is based on the 0.72 release. XGBoost (eXtreme Gradient Boosting) is a popular and efficient open-source implementation of the gradient boosted trees algorithm. Gradient boosting is a supervised learning algorithm that attempts to accurately predict a target variable by combining the estimates of a set of simpler, weaker models. XGBoost has done remarkably well in machine learning competitions because it robustly handles a variety of data types, relationships, and distributions, and because of the large number of hyperparameters that can be tweaked and tuned for improved fits. This flexibility makes XGBoost a solid choice for problems in regression, classification (binary and multiclass), and ranking.

Customers should consider using the new release of XGBoost Algorithm (p. 813). They can use it as a SageMaker built-in algorithm or as a framework to run scripts in their local environments as they would typically, for example, do with a Tensorflow deep learning framework. The new implementation has a smaller memory footprint, better logging, improved hyperparameter validation, and an expanded set of metrics. The earlier implementation of XGBoost remains available to customers if they need to postpone migrating to the new version. But this previous implementation will remain tied to the 0.72 release of XGBoost.

Input/Output Interface for the XGBoost Release 0.72

Gradient boosting operates on tabular data, with the rows representing observations, one column representing the target variable or label, and the remaining columns representing features.

The SageMaker implementation of XGBoost supports CSV and libsvm formats for training and inference:

- For Training ContentType, valid inputs are `text/libsvm` (default) or `text/csv`.
- For Inference ContentType, valid inputs are `text/libsvm` or (the default) `text/csv`. 
Note
For CSV training, the algorithm assumes that the target variable is in the first column and that the CSV does not have a header record. For CSV inference, the algorithm assumes that CSV input does not have the label column.
For libsvm training, the algorithm assumes that the label is in the first column. Subsequent columns contain the zero-based index value pairs for features. So each row has the format: <label> <index0>:<value0> <index1>:<value1> ... Inference requests for libsvm may or may not have labels in the libsvm format.

This differs from other SageMaker algorithms, which use the protobuf training input format to maintain greater consistency with standard XGBoost data formats.

For CSV training input mode, the total memory available to the algorithm (Instance Count * the memory available in the InstanceType) must be able to hold the training dataset. For libsvm training input mode, it's not required, but we recommend it.

SageMaker XGBoost uses the Python pickle module to serialize/deserialize the model, which can be used for saving/loading the model.

To use a model trained with SageMaker XGBoost in open source XGBoost

- Use the following Python code:

```python
import pickle as pkl
import tarfile
import xgboost
t = tarfile.open('model.tar.gz', 'r:gz')
t.extractall()
model = pkl.load(open(model_file_path, 'rb'))
# prediction with test data
pred = model.predict(dtest)
```

To differentiate the importance of labelled data points use Instance Weight Supports

- SageMaker XGBoost allows customers to differentiate the importance of labelled data points by assigning each instance a weight value. For text/libsvm input, customers can assign weight values to data instances by attaching them after the labels. For example, label:weight idx_0:val_0 idx_1:val_1.... For text/csv input, customers need to turn on the csv_weights flag in the parameters and attach weight values in the column after labels. For example: label,weight,val_0,val_1,...).

EC2 Instance Recommendation for the XGBoost Release 0.72

SageMaker XGBoost currently only trains using CPUs. It is a memory-bound (as opposed to compute-bound) algorithm. So, a general-purpose compute instance (for example, M4) is a better choice than a compute-optimized instance (for example, C4). Further, we recommend that you have enough total memory in selected instances to hold the training data. Although it supports the use of disk space to handle data that does not fit into main memory (the out-of-core feature available with the libsvm input mode), writing cache files onto disk slows the algorithm processing time.

XGBoost Release 0.72 Sample Notebooks

For a sample notebook that shows how to use the latest version of SageMaker XGBoost as a built-in algorithm to train and host a regression model, see Regression with Amazon SageMaker XGBoost algorithm. To use the 0.72 version of XGBoost, you need to change the version in the sample code to
Use built-in algorithms

0.72. For instructions how to create and access Jupyter notebook instances that you can use to run the example in SageMaker, see Use Amazon SageMaker Notebook Instances (p. 124). Once you have created a notebook instance and opened it, select the SageMaker Examples tab to see a list of all the SageMaker samples. The topic modeling example notebooks using the XGBoost algorithms are located in the Introduction to Amazon algorithms section. To open a notebook, click on its Use tab and select Create copy.

**XGBoost Release 0.72 Hyperparameters**

The following table contains the hyperparameters for the XGBoost algorithm. These are parameters that are set by users to facilitate the estimation of model parameters from data. The required hyperparameters that must be set are listed first, in alphabetical order. The optional hyperparameters that can be set are listed next, also in alphabetical order. The SageMaker XGBoost algorithm is an implementation of the open-source XGBoost package. Currently SageMaker supports version 0.72. For more detail about hyperparameter configuration for this version of XGBoost, see XGBoost Parameters.

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>num_class</td>
<td>The number of classes. Required if objective is set to multi:softmax or multi:softprob. Valid values: integer</td>
</tr>
<tr>
<td>num_round</td>
<td>The number of rounds to run the training. Required. Valid values: integer</td>
</tr>
<tr>
<td>alpha</td>
<td>L1 regularization term on weights. Increasing this value makes models more conservative. Optional. Valid values: float</td>
</tr>
<tr>
<td>base_score</td>
<td>The initial prediction score of all instances, global bias. Optional. Valid values: float. Default value: 0</td>
</tr>
<tr>
<td>booster</td>
<td>Which booster to use. The gbtree and dart values use a tree-based model, while gblinear uses a linear function. Optional. Valid values: String. One of gbtree, gblinear, or dart. Default value: gbtree</td>
</tr>
<tr>
<td>colsample_bylevel</td>
<td>Subsample ratio of columns for each split, in each level. Optional. Valid values: Float. Range: [0,1].</td>
</tr>
<tr>
<td>Parameter Name</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td></td>
<td><strong>Default value:</strong> 1</td>
</tr>
<tr>
<td><strong>colsample_bytree</strong></td>
<td>Subsample ratio of columns when constructing each tree.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: Float. Range: [0,1].</td>
</tr>
<tr>
<td></td>
<td>Default value: 1</td>
</tr>
<tr>
<td><strong>csv_weights</strong></td>
<td>When this flag is enabled, XGBoost differentiates the importance of instances for csv input by taking the second column (the column after labels) in training data as the instance weights.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: 0 or 1</td>
</tr>
<tr>
<td></td>
<td>Default value: 0</td>
</tr>
<tr>
<td><strong>early_stopping_rounds</strong></td>
<td>The model trains until the validation score stops improving. Validation error needs to decrease at least every \texttt{early_stopping_rounds} to continue training. SageMaker hosting uses the best model for inference.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: integer</td>
</tr>
<tr>
<td></td>
<td>Default value: -</td>
</tr>
<tr>
<td><strong>eta</strong></td>
<td>Step size shrinkage used in updates to prevent overfitting. After each boosting step, you can directly get the weights of new features. The \texttt{eta} parameter actually shrinks the feature weights to make the boosting process more conservative.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: Float. Range: [0,1].</td>
</tr>
<tr>
<td></td>
<td>Default value: 0.3</td>
</tr>
<tr>
<td><strong>eval_metric</strong></td>
<td>Evaluation metrics for validation data. A default metric is assigned according to the objective:</td>
</tr>
<tr>
<td></td>
<td>• \texttt{rmse}: for regression</td>
</tr>
<tr>
<td></td>
<td>• \texttt{error}: for classification</td>
</tr>
<tr>
<td></td>
<td>• \texttt{map}: for ranking</td>
</tr>
<tr>
<td></td>
<td>For a list of valid inputs, see \texttt{XGBoost Parameters}.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: string</td>
</tr>
<tr>
<td></td>
<td>Default value: Default according to objective.</td>
</tr>
<tr>
<td>Parameter Name</td>
<td>Description</td>
</tr>
<tr>
<td>----------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>gamma</td>
<td>Minimum loss reduction required to make a further partition on a leaf node of the tree. The larger, the more conservative the algorithm is.</td>
</tr>
<tr>
<td></td>
<td>Optional</td>
</tr>
<tr>
<td></td>
<td>Valid values: Float. Range: [0,\infty).</td>
</tr>
<tr>
<td></td>
<td>Default value: 0</td>
</tr>
<tr>
<td>grow_policy</td>
<td>Controls the way that new nodes are added to the tree. Currently supported only if <code>tree_method</code> is set to <code>hist</code>.</td>
</tr>
<tr>
<td></td>
<td>Optional</td>
</tr>
<tr>
<td></td>
<td>Valid values: String. Either <code>depthwise</code> or <code>losguide</code>.</td>
</tr>
<tr>
<td></td>
<td>Default value: <code>depthwise</code></td>
</tr>
<tr>
<td>lambda</td>
<td>L2 regularization term on weights. Increasing this value makes models more conservative.</td>
</tr>
<tr>
<td></td>
<td>Optional</td>
</tr>
<tr>
<td></td>
<td>Valid values: float</td>
</tr>
<tr>
<td></td>
<td>Default value: 1</td>
</tr>
<tr>
<td>lambda_bias</td>
<td>L2 regularization term on bias.</td>
</tr>
<tr>
<td></td>
<td>Optional</td>
</tr>
<tr>
<td></td>
<td>Valid values: Float. Range: [0.0, 1.0].</td>
</tr>
<tr>
<td></td>
<td>Default value: 0</td>
</tr>
<tr>
<td>max_bin</td>
<td>Maximum number of discrete bins to bucket continuous features. Used only if <code>tree_method</code> is set to <code>hist</code>.</td>
</tr>
<tr>
<td></td>
<td>Optional</td>
</tr>
<tr>
<td></td>
<td>Valid values: integer</td>
</tr>
<tr>
<td></td>
<td>Default value: 256</td>
</tr>
<tr>
<td>max_delta_step</td>
<td>Maximum delta step allowed for each tree's weight estimation. When a positive integer is used, it helps make the update more conservative. The preferred option is to use it in logistic regression. Set it to 1-10 to help control the update.</td>
</tr>
<tr>
<td></td>
<td>Optional</td>
</tr>
<tr>
<td></td>
<td>Valid values: Integer. Range: [0,\infty).</td>
</tr>
<tr>
<td></td>
<td>Default value: 0</td>
</tr>
<tr>
<td>Parameter Name</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>max_depth</td>
<td>Maximum depth of a tree. Increasing this value makes the model more complex and likely to be overfit. 0 indicates no limit. A limit is required when <code>grow_policy=depth-wise</code>.</td>
</tr>
<tr>
<td></td>
<td>Optional</td>
</tr>
<tr>
<td></td>
<td>Valid values: Integer. Range: [0,∞)</td>
</tr>
<tr>
<td></td>
<td>Default value: 6</td>
</tr>
<tr>
<td>max_leaves</td>
<td>Maximum number of nodes to be added. Relevant only if <code>grow_policy</code> is set to <code>lossguide</code>.</td>
</tr>
<tr>
<td></td>
<td>Optional</td>
</tr>
<tr>
<td></td>
<td>Valid values: integer</td>
</tr>
<tr>
<td></td>
<td>Default value: 0</td>
</tr>
<tr>
<td>min_child_weight</td>
<td>Minimum sum of instance weight (hessian) needed in a child. If the tree partition step results in a leaf node with the sum of instance weight less than <code>min_child_weight</code>, the building process gives up further partitioning. In linear regression models, this simply corresponds to a minimum number of instances needed in each node. The larger the algorithm, the more conservative it is.</td>
</tr>
<tr>
<td></td>
<td>Optional</td>
</tr>
<tr>
<td></td>
<td>Valid values: Float. Range: [0,∞).</td>
</tr>
<tr>
<td></td>
<td>Default value: 1</td>
</tr>
<tr>
<td>normalize_type</td>
<td>Type of normalization algorithm.</td>
</tr>
<tr>
<td></td>
<td>Optional</td>
</tr>
<tr>
<td></td>
<td>Valid values: Either <code>tree</code> or <code>forest</code>.</td>
</tr>
<tr>
<td></td>
<td>Default value: <code>tree</code></td>
</tr>
<tr>
<td>nthread</td>
<td>Number of parallel threads used to run <code>xgboost</code>.</td>
</tr>
<tr>
<td></td>
<td>Optional</td>
</tr>
<tr>
<td></td>
<td>Valid values: integer</td>
</tr>
<tr>
<td></td>
<td>Default value: Maximum number of threads.</td>
</tr>
<tr>
<td>objective</td>
<td>Specifies the learning task and the corresponding learning objective. Examples: <code>reg:logistic</code>, <code>reg:softmax</code>, <code>multi:squarederror</code>. For a full list of valid inputs, refer to <code>XGBoost Parameters</code>.</td>
</tr>
<tr>
<td></td>
<td>Optional</td>
</tr>
<tr>
<td></td>
<td>Valid values: string</td>
</tr>
<tr>
<td></td>
<td>Default value: <code>reg:squarederror</code></td>
</tr>
<tr>
<td>Parameter Name</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>one_drop</td>
<td>When this flag is enabled, at least one tree is always dropped during the dropout.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: 0 or 1</td>
</tr>
<tr>
<td></td>
<td>Default value: 0</td>
</tr>
<tr>
<td>process_type</td>
<td>The type of boosting process to run.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: String. Either <code>default</code> or <code>update</code>.</td>
</tr>
<tr>
<td></td>
<td>Default value: <code>default</code></td>
</tr>
<tr>
<td>rate_drop</td>
<td>The dropout rate that specifies the fraction of previous trees to drop during the dropout.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: Float. Range: [0.0, 1.0].</td>
</tr>
<tr>
<td></td>
<td>Default value: 0.0</td>
</tr>
<tr>
<td>refresh_leaf</td>
<td>This is a parameter of the 'refresh' updater plug-in. When set to true (1), tree leaves and tree node stats are updated. When set to false (0), only tree node stats are updated.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: 0/1</td>
</tr>
<tr>
<td></td>
<td>Default value: 1</td>
</tr>
<tr>
<td>sample_type</td>
<td>Type of sampling algorithm.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: Either <code>uniform</code> or <code>weighted</code>.</td>
</tr>
<tr>
<td></td>
<td>Default value: <code>uniform</code></td>
</tr>
<tr>
<td>scale_pos_weight</td>
<td>Controls the balance of positive and negative weights. A typical value to consider: ( \frac{\text{sum(negative cases)}}{\text{sum(positive cases)}} ).</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: float</td>
</tr>
<tr>
<td></td>
<td>Default value: 1</td>
</tr>
<tr>
<td>Parameter Name</td>
<td>Description</td>
</tr>
<tr>
<td>----------------</td>
<td>-------------</td>
</tr>
</tbody>
</table>
| seed           | Random number seed.  
  **Optional**  
  Valid values: integer  
  Default value: 0 |
| silent         | 0 means print running messages, 1 means silent mode.  
  Valid values: 0 or 1  
  **Optional**  
  Default value: 0 |
| sketch_eps     | Used only for approximate greedy algorithm. This translates into $O(1 / \text{sketch\_eps})$ number of bins. Compared to directly select number of bins, this comes with theoretical guarantee with sketch accuracy.  
  **Optional**  
  Valid values: Float, Range: [0, 1].  
  Default value: 0.03 |
| skip_drop      | Probability of skipping the dropout procedure during a boosting iteration.  
  **Optional**  
  Valid values: Float. Range: [0.0, 1.0].  
  Default value: 0.0 |
| subsample      | Subsample ratio of the training instance. Setting it to 0.5 means that XGBoost randomly collects half of the data instances to grow trees. This prevents overfitting.  
  **Optional**  
  Valid values: Float. Range: [0,1].  
  Default value: 1 |
| tree_method    | The tree construction algorithm used in XGBoost.  
  **Optional**  
  Valid values: One of auto, exact, approx, or hist.  
  Default value: auto |
Use built-in algorithms

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tweedie_variance_power</td>
<td>Parameter that controls the variance of the Tweedie distribution.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: Float. Range: (1, 2).</td>
</tr>
<tr>
<td></td>
<td>Default value: 1.5</td>
</tr>
<tr>
<td>updater</td>
<td>A comma-separated string that defines the sequence of tree updaters to run.</td>
</tr>
<tr>
<td></td>
<td>This provides a modular way to construct and to modify the trees.</td>
</tr>
<tr>
<td></td>
<td>For a full list of valid inputs, please refer to XGBoost Parameters.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: comma-separated string.</td>
</tr>
<tr>
<td></td>
<td>Default value: grow_colmaker, prune</td>
</tr>
</tbody>
</table>

Tune an XGBoost Release 0.72 Model

*Automatic model tuning*, also known as hyperparameter tuning, finds the best version of a model by running many jobs that test a range of hyperparameters on your training and validation datasets. You choose three types of hyperparameters:

- a **learning objective** function to optimize during model training
- an **eval_metric** to use to evaluate model performance during validation
- a set of hyperparameters and a range of values for each to use when tuning the model automatically

You choose the evaluation metric from set of evaluation metrics that the algorithm computes. Automatic model tuning searches the hyperparameters chosen to find the combination of values that result in the model that optimizes the evaluation metric.

For more information about model tuning, see [Perform Automatic Model Tuning](p. 1023).

Metrics Computed by the XGBoost Release 0.72 Algorithm

The XGBoost algorithm based on version 0.72 computes the following nine metrics to use for model validation. When tuning the model, choose one of these metrics to evaluate the model. For full list of valid eval_metric values, refer to XGBoost Learning Task Parameters

<table>
<thead>
<tr>
<th>Metric Name</th>
<th>Description</th>
<th>Optimization Direction</th>
</tr>
</thead>
<tbody>
<tr>
<td>validation:auc</td>
<td>Area under the curve.</td>
<td>Maximize</td>
</tr>
<tr>
<td>validation:error</td>
<td>Binary classification error rate, calculated as #/(wrong cases)/#(all cases).</td>
<td>Minimize</td>
</tr>
<tr>
<td>validation:logloss</td>
<td>Negative log-likelihood.</td>
<td>Minimize</td>
</tr>
<tr>
<td>validation:mae</td>
<td>Mean absolute error.</td>
<td>Minimize</td>
</tr>
<tr>
<td>validation:map</td>
<td>Mean average precision.</td>
<td>Maximize</td>
</tr>
</tbody>
</table>
Tunable XGBoost Release 0.72 Hyperparameters

Tune the XGBoost model with the following hyperparameters. The hyperparameters that have the greatest effect on optimizing the XGBoost evaluation metrics are: `alpha`, `min_child_weight`, `subsample`, `eta`, and `num_round`.

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Parameter Type</th>
<th>Recommended Ranges</th>
</tr>
</thead>
<tbody>
<tr>
<td>alpha</td>
<td>ContinuousParameterRanges</td>
<td>MinValue: 0, MaxValue: 1000</td>
</tr>
<tr>
<td>colsample_bylevel</td>
<td>ContinuousParameterRanges</td>
<td>MinValue: 0.1, MaxValue: 1</td>
</tr>
<tr>
<td>colsample_bytree</td>
<td>ContinuousParameterRanges</td>
<td>MinValue: 0.5, MaxValue: 1</td>
</tr>
<tr>
<td>eta</td>
<td>ContinuousParameterRanges</td>
<td>MinValue: 0.1, MaxValue: 0.5</td>
</tr>
<tr>
<td>gamma</td>
<td>ContinuousParameterRanges</td>
<td>MinValue: 0, MaxValue: 5</td>
</tr>
<tr>
<td>lambda</td>
<td>ContinuousParameterRanges</td>
<td>MinValue: 0, MaxValue: 1000</td>
</tr>
<tr>
<td>max_delta_step</td>
<td>IntegerParameterRanges</td>
<td>[0, 10]</td>
</tr>
<tr>
<td>max_depth</td>
<td>IntegerParameterRanges</td>
<td>[0, 10]</td>
</tr>
<tr>
<td>min_child_weight</td>
<td>ContinuousParameterRanges</td>
<td>MinValue: 0, MaxValue: 120</td>
</tr>
<tr>
<td>num_round</td>
<td>IntegerParameterRanges</td>
<td>[1, 4000]</td>
</tr>
<tr>
<td>subsample</td>
<td>ContinuousParameterRanges</td>
<td>MinValue: 0.5, MaxValue: 1</td>
</tr>
</tbody>
</table>

Use reinforcement learning with Amazon SageMaker

Reinforcement learning (RL) is a machine learning technique that attempts to learn a strategy, called a policy, that optimizes an objective for an agent acting in an environment. For example, the agent might be a robot, the environment might be a maze, and the goal might be to successfully navigate the maze in the smallest amount of time. In RL, the agent takes an action, observes the state of the environment, and gets a reward based on the value of the current state of the environment. The goal is to maximize
the long-term reward that the agent receives as a result of its actions. RL is well-suited for solving problems in which an agent can make autonomous decisions.

Topics
- Why is Reinforcement Learning Important? (p. 839)
- Markov Decision Process (MDP) (p. 839)
- Key Features of Amazon SageMaker RL (p. 840)
- Reinforcement Learning Sample Notebooks (p. 842)
- Sample RL Workflow Using Amazon SageMaker RL (p. 842)
- RL Environments in Amazon SageMaker (p. 843)
- Distributed Training with Amazon SageMaker RL (p. 845)
- Hyperparameter Tuning with Amazon SageMaker RL (p. 845)

Why is Reinforcement Learning Important?

RL is well-suited for solving large, complex problems, such as supply chain management, HVAC systems, industrial robotics, game artificial intelligence, dialog systems, and autonomous vehicles. Because RL models learn by a continuous process of receiving rewards and punishments for every action taken by the agent, it is possible to train systems to make decisions under uncertainty and in dynamic environments.

Markov Decision Process (MDP)

RL is based on models called Markov Decision Processes (MDPs). An MDP consists of a series of time steps. Each time step consists of the following:

Environment

Defines the space in which the RL model operates. This can be either a real-world environment or a simulator. For example, if you train a physical autonomous vehicle on a physical road, that would be a real-world environment. If you train a computer program that models an autonomous vehicle driving on a road, that would be a simulator.

State

Specifies all information about the environment and past steps that is relevant to the future. For example, in an RL model in which a robot can move in any direction at any time step, the position of the robot at the current time step is the state, because if we know where the robot is, it isn't necessary to know the steps it took to get there.

Action

What the agent does. For example, the robot takes a step forward.

Reward

A number that represents the value of the state that resulted from the last action that the agent took. For example, if the goal is for a robot to find treasure, the reward for finding treasure might be 5, and the reward for not finding treasure might be 0. The RL model attempts to find a strategy that optimizes the cumulative reward over the long term. This strategy is called a policy.

Observation

Information about the state of the environment that is available to the agent at each step. This might be the entire state, or it might be just a part of the state. For example, the agent in a chess-playing model would be able to observe the entire state of the board at any step, but a robot in a maze might only be able to observe a small portion of the maze that it currently occupies.
Typically, training in RL consists of many episodes. An episode consists of all of the time steps in an MDP from the initial state until the environment reaches the terminal state.

**Key Features of Amazon SageMaker RL**

To train RL models in SageMaker RL, use the following components:

- An RL environment. You can use custom environments, open-source environments, or commercial environments. For information, see [RL Environments in Amazon SageMaker](#) (p. 843).

The following diagram shows the RL components that are supported in SageMaker RL.
Use reinforcement learning
Reinforcement Learning Sample Notebooks

The following table outlines a variety of sample notebooks that address different use cases of Amazon SageMaker reinforcement learning.

<table>
<thead>
<tr>
<th>Notebook Title</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>How to Train Batch RL Policies?</td>
<td>This notebook shows how to use batch RL to train a new policy from an offline dataset.</td>
</tr>
<tr>
<td>How to Solve the Cart-pole Balancing Problem?</td>
<td>This notebook shows how to solve the cart-pole balancing problem with RL.</td>
</tr>
<tr>
<td>How to Solve the Knapsack Problem?</td>
<td>This notebook shows how to use RL to solve the knapsack problem, and how SageMaker Managed Spot Training can be used to run training at a lower cost.</td>
</tr>
<tr>
<td>How to Solve the Mountain Car Problem?</td>
<td>This notebook shows how to solve the mountain car control problem with RL.</td>
</tr>
<tr>
<td>How to Train a Distributed Object Tracker with RL?</td>
<td>This notebook shows how to train a Distributed Object Tracker that learns to track and follow another Robot using Robomaker.</td>
</tr>
</tbody>
</table>

Sample RL Workflow Using Amazon SageMaker RL

The following example describes the steps for developing RL models using Amazon SageMaker RL.

For complete code examples, see the sample notebooks at https://github.com/awslabs/amazon-sagemaker-examples/tree/master/reinforcement-learning.

1. **Formulate the RL problem**—First, formulate the business problem into an RL problem. For example, auto scaling enables services to dynamically increase or decrease capacity depending on conditions that you define. Currently, this requires setting up alarms, scaling policies, thresholds, and other manual steps. To solve this with RL, we define the components of the Markov Decision Process:
   a. **Objective**—Scale instance capacity so that it matches the desired load profile.
   b. **Environment**—A custom environment that includes the load profile. It generates a simulated load with daily and weekly variations and occasional spikes. The simulated system has a delay between when new resources are requested and when they become available for serving requests.
   c. **State**—The current load, number of failed jobs, and number of active machines.
   d. **Action**—Remove, add, or keep the same number of instances.
   e. **Reward**—A positive reward for successful transactions and a high penalty for failing transactions beyond a specified threshold.

2. **Define the RL environment**—The RL environment can be the real world where the RL agent interacts or a simulation of the real world. You can connect open source and custom environments developed using Gym interfaces and commercial simulation environments such as MATLAB and Simulink.

3. **Define the presets**—The presets configure the RL training jobs and define the hyperparameters for the RL algorithms.

4. **Write the training code**—Write training code as a Python script and pass the script to a SageMaker training job. In your training code, import the environment files and the preset files, and then define the `main()` function.
5. **Train the RL Model**—Use the SageMaker RLEstimator in the Amazon SageMaker Python SDK to start an RL training job. If you are using local mode, the training job runs on the notebook instance. When you use SageMaker for training, you can select GPU or CPU instances. Store the output from the training job in a local directory if you train in local mode, or on Amazon S3 if you use SageMaker training.

The RLEstimator requires the following information as parameters.

a. The source directory where the environment, presets, and training code are uploaded.

b. The path to the training script.

c. The RL toolkit and deep learning framework you want to use. This automatically resolves to the Amazon ECR path for the RL container.

d. The training parameters, such as the instance count, job name, and S3 path for output.

e. Metric definitions that you want to capture in your logs. These can also be visualized in CloudWatch and in SageMaker notebooks.

6. **Visualize training metrics and output**—After a training job that uses an RL model completes, you can view the metrics you defined in the training jobs in CloudWatch. You can also plot the metrics in a notebook by using the Amazon SageMaker Python SDK analytics library. Visualizing metrics helps you understand how the performance of the model as measured by the reward improves over time.

   **Note**
   If you train in local mode, you can't visualize metrics in CloudWatch.

7. **Evaluate the model**—Checkpointed data from the previously trained models can be passed on for evaluation and inference in the checkpoint channel. In local mode, use the local directory. In SageMaker training mode, you need to upload the data to S3 first.

8. **Deploy RL models**—Finally, deploy the trained model on an endpoint hosted on SageMaker containers or on an edge device by using AWS IoT Greengrass.

For more information on RL with SageMaker, see Using RL with the SageMaker Python SDK.

**RL Environments in Amazon SageMaker**

Amazon SageMaker RL uses environments to mimic real-world scenarios. Given the current state of the environment and an action taken by the agent or agents, the simulator processes the impact of the action, and returns the next state and a reward. Simulators are useful in cases where it is not safe to train an agent in the real world (for example, flying a drone) or if the RL algorithm takes a long time to converge (for example, when playing chess).

The following diagram shows an example of the interactions with a simulator for a car racing game.
The simulation environment consists of an agent and a simulator. Here, a convolutional neural network (CNN) consumes images from the simulator and generates actions to control the game controller. With multiple simulations, this environment generates training data of the form \( \text{state}_t, \text{action}, \text{state}_{t+1} \), and \( \text{reward}_{t+1} \). Defining the reward is not trivial and impacts the RL model quality. We want to provide a few examples of reward functions, but would like to make it user-configurable.

**Topics**
- Use OpenAI Gym Interface for Environments in SageMaker RL (p. 844)
- Use Open-Source Environments (p. 845)
- Use Commercial Environments (p. 845)

**Use OpenAI Gym Interface for Environments in SageMaker RL**

To use OpenAI Gym environments in SageMaker RL, use the following API elements. For more information about OpenAI Gym, see [https://gym.openai.com/docs/](https://gym.openai.com/docs/).

- env.action_space—Defines the actions the agent can take, specifies whether each action is continuous or discrete, and specifies the minimum and maximum if the action is continuous.
- env.observation_space—Defines the observations the agent receives from the environment, as well as minimum and maximum for continuous observations.
- env.reset()—Initializes a training episode. The reset() function returns the initial state of the environment, and the agent uses the initial state to take its first action. The action is then sent to step() repeatedly until the episode reaches a terminal state. When step() returns done = True, the episode ends. The RL toolkit re-initializes the environment by calling reset().
- step()—Takes the agent action as input and outputs the next state of the environment, the reward, whether the episode has terminated, and an info dictionary to communicate debugging information. It is the responsibility of the environment to validate the inputs.
- env.render()—Used for environments that have visualization. The RL toolkit calls this function to capture visualizations of the environment after each call to the step() function.
Use Open-Source Environments

You can use open-source environments, such as EnergyPlus and RoboSchool, in SageMaker RL by building your own container. For more information about EnergyPlus, see https://energyplus.net/. For more information about RoboSchool, see https://github.com/openai/roboschool. The HVAC and RoboSchool examples in the SageMaker examples repository show how to build a custom container to use with SageMaker RL:

Use Commercial Environments

You can use commercial environments, such as MATLAB and Simulink, in SageMaker RL by building your own container. You need to manage your own licenses.

Distributed Training with Amazon SageMaker RL

Amazon SageMaker RL supports multi-core and multi-instance distributed training. Depending on your use case, training and/or environment rollout can be distributed. For example, SageMaker RL works for the following distributed scenarios:

- Single training instance and multiple rollout instances of the same instance type. For an example, see the Neural Network Compression example in the SageMaker examples repository.
- Single trainer instance and multiple rollout instances, where different instance types for training and rollouts. For an example, see the AWS DeepRacer / AWS RoboMaker example in the SageMaker examples repository.
- Single trainer instance that uses multiple cores for rollout. For an example, see the Roboschool example in the SageMaker examples repository. This is useful if the simulation environment is lightweight and can run on a single thread.
- Multiple instances for training and rollouts. For an example, see the Roboschool example in the SageMaker examples repository.

Hyperparameter Tuning with Amazon SageMaker RL

You can run a hyperparameter tuning job to optimize hyperparameters for Amazon SageMaker RL. The Roboschool example in the sample notebooks in the SageMaker examples repository shows how you can do this with RL Coach. The launcher script shows how you can abstract parameters from the Coach preset file and optimize them.

Manage Machine Learning with Amazon SageMaker Experiments

Amazon SageMaker Experiments is a capability of Amazon SageMaker that lets you organize, track, compare, and evaluate your machine learning experiments.

Machine learning is an iterative process. You need to experiment with multiple combinations of data, algorithm and parameters, all the while observing the impact of incremental changes on model accuracy. Over time this iterative experimentation can result in thousands of model training runs and model versions. This makes it hard to track the best performing models and their input configurations. It’s also difficult to compare active experiments with past experiments to identify opportunities for further incremental improvements.

SageMaker Experiments automatically tracks the inputs, parameters, configurations, and results of your iterations as trials. You can assign, group, and organize these trials into experiments. SageMaker
Experiments is integrated with Amazon SageMaker Studio providing a visual interface to browse your active and past experiments, compare trials on key performance metrics, and identify the best performing models.

SageMaker Experiments comes with its own Experiments SDK which makes the analytics capabilities easily accessible in Amazon SageMaker Notebooks. Because SageMaker Experiments enables tracking of all the steps and artifacts that went into creating a model, you can quickly revisit the origins of a model when you are troubleshooting issues in production, or auditing your models for compliance verifications.

**Topics**
- SageMaker Experiments Features (p. 846)
- Create an Amazon SageMaker Experiment (p. 847)
- View and Compare Amazon SageMaker Experiments, Trials, and Trial Components (p. 850)
- Track and Compare Tutorial (p. 853)
- Search Experiments Using Amazon SageMaker Studio (p. 859)
- Clean Up Amazon SageMaker Experiment Resources (p. 864)
- Search Using the Amazon SageMaker Console and API (p. 866)

### SageMaker Experiments Features

The following sections provide a brief overview of the features provided by SageMaker Experiments.

**Topics**
- Organize Experiments (p. 846)
- Track Experiments (p. 846)
- Compare and Evaluate Experiments (p. 847)
- Amazon SageMaker Autopilot (p. 847)

### Organize Experiments

Amazon SageMaker Experiments offers a structured organization scheme to help users group and organize their machine learning iterations. The top level entity, an experiment, is a collection of trials that are observed, compared, and evaluated as a group. A trial is a set of steps called trial components. Each trial component can include a combination of inputs such as datasets, algorithms, and parameters, and produce specific outputs such as models, metrics, datasets, and checkpoints. Examples of trial components are data pre-processing jobs, training jobs, and batch transform jobs.

The goal of an experiment is to determine the trial that produces the best model. Multiple trials are performed, each one isolating and measuring the impact of a change to one or more inputs, while keeping the remaining inputs constant. By analyzing the trials, you can determine which features have the most effect on the model.

### Track Experiments

Amazon SageMaker Experiments enables tracking of experiments.

**Automated Tracking**

SageMaker Experiments automatically tracks Amazon SageMaker Autopilot jobs as experiments with their underlying training jobs tracked as trials. SageMaker Experiments also automatically tracks SageMaker independently executed training, batch transform, and processing jobs as trial components, whether assigned to a trial or left unassigned. Unassigned trial components can be associated with a trial at a later time. All experiment artifacts including datasets, algorithms, hyperparameters, and model
metrics are tracked and recorded. This data allows customers to trace the complete lineage of a model which helps with model governance, auditing, and compliance verifications.

**Manual Tracking**

SageMaker Experiments provides tracking APIs for recording and tracking machine learning workflows running locally on SageMaker Studio notebooks, including classic SageMaker notebooks. These experiments must be part of a SageMaker training, batch transform, or processing job.

**Compare and Evaluate Experiments**

Amazon SageMaker Experiments is integrated with Amazon SageMaker Studio. When you use SageMaker Studio, SageMaker Experiments automatically tracks your experiments and trials, and presents visualizations of the tracked data and an interface to search the data.

SageMaker Experiments automatically organizes, ranks, and sorts trials based on a chosen metric using the concept of a trial leaderboard. SageMaker Studio produces real-time data visualizations, such as metric charts and graphs, to quickly compare and identify the best performing models. These are updated in real-time as the experiment progresses.

**Amazon SageMaker Autopilot**

Amazon SageMaker Experiments is integrated with Amazon SageMaker Autopilot. When you perform an Autopilot job, SageMaker Experiments creates an experiment for the job, and trials for each of the different combinations of the available trial components, parameters, and artifacts. You can visually drill into all trials and components using SageMaker Studio.

**Create an Amazon SageMaker Experiment**

Create an Amazon SageMaker experiment to track your SageMaker training, processing, and transform jobs.

The following procedure shows you how to create a SageMaker experiment for a SageMaker training, processing, or transform job. Steps labeled as (Studio) describe how to view the experiment in Amazon SageMaker Studio. You don't have to run the experiment in Studio to view the experiment in Studio.

For a tutorial that shows this functionality in an existing SageMaker Studio notebook, see Track and Compare Tutorial (p. 853).

1. Import the `sys` module to install the SDKs.

   ```python
   import sys
   ```

2. (Optional) The Amazon SageMaker Python SDK, comes preinstalled in SageMaker Studio. If you plan to run your code outside Studio, install the SageMaker Python SDK.

   ```bash
   !{sys.executable} -m pip install sagemaker
   ```

3. Install the SageMaker Experiments Python SDK.

   ```bash
   !{sys.executable} -m pip install sagemaker-experiments
   ```

4. Import modules.

   ```python
   import time
   from time import strftime
   ```
import sagemaker
from smexperiments.experiment import Experiment
from smexperiments.trial import Trial
from smexperiments.trial_component import TrialComponent
from smexperiments.tracker import Tracker

5. Get the execution role and create the SageMaker session.

```python
role = sagemaker.get_execution_role()
sm_sess = sagemaker.session.Session()
```

6. Create a SageMaker experiment. The experiment name must be unique in your account.

```
create_date = strftime("%Y-%m-%d-%H-%M-%S")
demo_experiment = Experiment.create(experiment_name = "DEMO-{}").format(create_date),
                   description = "Demo experiment",
                   tags = [{'Key': 'my-experiments', 'Value': 'demo1'}])
```

Note
The tags parameter is optional. You can search for the tag using Studio, the SageMaker console, and the SDK. Tags can also be applied to trials and trial components. For information on how to search tags using Studio, see Search by Tag (p. 863).

7. (Studio) To view the experiment in SageMaker Studio, in the left sidebar, choose the SageMaker Experiment List icon ( ⌴ ) to display the experiments browser.

After the code runs, the experiment list contains the new experiment. It might take a moment for the list to refresh and display the experiment. The filter on the experiment tag is also displayed. Only experiments that have a matching tag are displayed. Your list should look similar to the following:

8. Create a trial for the experiment. The trial name must be unique in your account.

```python
demo_trial = Trial.create(trial_name = "DEMO-{}").format(create_date),
                        experiment_name = demo_experiment.experiment_name,
                        tags = [{'Key': 'my-experiments', 'Value': 'demo1'}])
```

9. (Studio) In the experiment list, double-click the experiment to display a list of the trials in the experiment (this example has one trial). Your list should look similar to the following:
10. Create a trial component as part of the trial. The trial component is the SageMaker job.

Add the `ExperimentConfig` parameter to the appropriate method. The SageMaker jobs listed in the following table are supported.

<table>
<thead>
<tr>
<th>Job</th>
<th>SageMaker Python SDK method</th>
<th>Boto3 method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Training</td>
<td><code>Estimator.fit</code></td>
<td><code>CreateTrainingJob</code></td>
</tr>
<tr>
<td>Processing</td>
<td><code>Processor.run</code></td>
<td><code>CreateProcessingJob</code></td>
</tr>
<tr>
<td>Transform</td>
<td><code>Transformer.transform</code></td>
<td><code>CreateTransformJob</code></td>
</tr>
</tbody>
</table>

The following examples are for a training job. The `Tags` parameter adds a tag to the trial component. `ExperimentName` isn’t specified because the trial was associated with the experiment when the trial was created in an earlier step.

**Using the SageMaker Python SDK**

```python
sagemaker.estimator.Estimator(
    ...,
    sagemaker_session = sm_sess,
    tags = [{'Key': 'my-experiments', 'Value': 'demo2'}])
estimator.fit(
    ...,
    experiment_config = {
        # "ExperimentName"
        "TrialName" : demo_trial.trial_name,
        "TrialComponentDisplayName" : "TrainingJob",
    })
```

**Using Boto3**

```python
create_training_job(
    ...,
    "ExperimentConfig": {
        # "ExperimentName"
        "TrialName" : demo_trial.trial_name,
        "TrialComponentDisplayName" : "TrainingJob",
    },
```

```
View and Compare Amazon SageMaker Experiments, Trials, and Trial Components

An Amazon SageMaker experiment consists of multiple trials with a related objective. A trial consists of one or more trial components, such as a data preprocessing job and a training job.

You use the experiments browser to display a list of these entities. You can filter the list by entity name, type, and tags. The entities are presented in a hierarchical view (that is, experiments > trials > trial components). Double-click an entity to see other entities that are at a lower level in the hierarchy. Use the breadcrumbs above the list to move to a higher level in the hierarchy.

For a tutorial using a SageMaker example notebook, see Track and Compare Tutorial (p. 853). For an overview of the Studio user interface, see Amazon SageMaker Studio UI Overview (p. 79).

Topics
- View Experiments, Trials, and Trial Components (p. 850)
- Compare Experiments, Trials, and Trial Components (p. 852)

View Experiments, Trials, and Trial Components

Amazon SageMaker Studio provides an experiments browser that you can use to view lists of experiments, trials, and trial components. You can choose one of these entities to view detailed information about the entity or choose multiple entities for comparison.
To view experiments, trials, and trial components

1.
In the left sidebar of Studio, choose the **SageMaker Experiment List** icon ( ). A list of experiments and their properties is displayed. The list includes all the SageMaker experiments in your account, including experiments created outside of SageMaker Studio.

**Note**
To view all the properties, you might have to expand the width of the experiments browser by dragging the right border.

2. In the experiments list, double-click an experiment to display a list of the trials in the experiment.

3. Double-click a trial to display a list of the components in the trial.
4. Double-click one of the components to open the **Describe Trial Component** tab.

5. On the **Describe Trial Component** tab, choose any of the following column headings to see available information about each trial component:

   - **Charts** – Build your own charts.
   - **Metrics** – Metrics that are logged by a Tracker during a trial run.
   - **Parameters** – Hyperparameter values and instance information.
   - **Artifacts** – Amazon S3 bucket storage locations for the input dataset and the output model.
   - **AWS Settings** – Job name, ARN, status, creation time, training time, billable time, instance information, and others.
   - **Debugger** – A list of debugger rules and any issues found.
   - **Trial Mappings**

For information about comparing Experiments entities, see View and Compare Amazon SageMaker Experiments, Trials, and Trial Components (p. 850).

**Compare Experiments, Trials, and Trial Components**

You can compare experiments, trials, and trial components by selecting the entities and opening them in the trial components list. The trial components list is referred to as the Studio Leaderboard. In the Leaderboard you can do the following:

- View detailed information about the entities
- Compare entities
- Stop a training job
• Deploy a model

To compare experiments, trials, and trial components

1. In the left sidebar of SageMaker Studio, choose the SageMaker Experiment List icon.
2. In the Experiments browser, choose either the experiment or trial list. For more information, see View Experiments, Trials, and Trial Components (p. 850).
3. Choose the experiments or trials that you want to compare, right-click the selection, and then choose Open in trial component list. The Leaderboard opens and lists the associated Experiments entities as shown in the following screenshot.

![Leaderboard screenshot](image)

The Leaderboard has a TABLE PROPERTIES pane on the right side. Use the Settings icon ( ) to open and close the pane. You can hide or display properties by category or by individual columns. When you display a chart, the pane changes to display chart properties.

For information on searching the Experiments entities, see Search Experiments Using Amazon SageMaker Studio (p. 859).

Track and Compare Tutorial

This tutorial demonstrates how to visually track and compare trials in a model training experiment using Amazon SageMaker Studio. The basis of the tutorial is the MNIST Handwritten Digits Classification Experiment (MNIST) example notebook.

It is intended that this topic be viewed alongside Studio with the MNIST notebook open. As you run through the cells, the sections in this document highlight the relevant code and show you how to observe the results in Studio. Some of the code snippets have been edited for brevity.

To clean up the resources created by the notebook, see Clean Up Amazon SageMaker Experiment Resources (p. 864).

For a tutorial that showcases additional features of Studio, see Amazon SageMaker Studio Tour (p. 50).

Prerequisites

• A local copy of the MNIST example notebook and the companion mnist.py file. Both files are available from the aws_sagemaker_studio/sagemaker_experiments/mnist-handwritten-digits-
classification-experiment folder in the `aws/amazon-sagemaker-examples` repository. To download the files, choose each link, right-click on the Raw button, and then choose Save as.

- An AWS SSO or IAM account to sign-on to SageMaker Studio. For more information, see Onboard to Amazon SageMaker Studio (p. 34).

**Topics**

- Open the Notebook in Studio (p. 854)
- Install the Experiments SDK and Import Modules (p. 854)
- Transform and Track the Input Data (p. 854)
- Create and Track an Experiment (p. 855)
- Compare and Analyze Trials (p. 857)

## Open the Notebook in Studio

**To open the notebook**

1. Sign-on to Studio.
2. In the left sidebar, choose the **File Browser** icon (📁).
3. At the top of the file browser pane, choose the **Up arrow** icon and then a **File Upload** dialog opens. Browse to and choose your local versions of the `mnist-handwritten-digits-classification-experiment.ipynb` and `mnist.py` files, and then choose Open.
4. The two files are listed in the file browser. Double-click the uploaded notebook file to open the notebook in a new tab.
5. At the top right of the notebook, make sure the kernel is **Python 3 (Data Science)**. If not, choose the current kernel name to open the **Select Kernel** dropdown. Choose **Python 3 (Data Science)** and then choose Select.

## Install the Experiments SDK and Import Modules

The SageMaker Experiments SDK is separate from the Amazon SageMaker Python SDK, which comes preinstalled in SageMaker Studio. Run the first few cells in the notebook to install the Experiments SDK and import the Experiments modules. The relevant sections of the notebook cells are displayed below. For more information on the Experiments SDK, see sagemaker-experiments.

```python
import sys
!{sys.executable} -m pip install sagemaker-experiments

import sagemaker
from sagemaker import get_execution_role
from sagemaker.session import Session
from sagemaker.analytics import ExperimentAnalytics

from smexperiments.experiment import Experiment
from smexperiments.trial import Trial
from smexperiments.trial_component import TrialComponent
from smexperiments.tracker import Tracker
```

## Transform and Track the Input Data

The next few cells create an Amazon S3 bucket and a folder in the bucket named `mnist`. In Studio, the file browser displays the `mnist` folder. The input data is downloaded to the `mnist/MNIST/raw` folder,
normalized, and then the transformed data is uploaded to the `mnist/MNIST/processed` folder. You can drill down into the `mnist` folder to display, but not open, the data files.

Your screen should look similar to the following:

![Image showing the dataset folder in SageMaker Studio]

The last cell in the **Dataset** section creates a Tracker for the transform job. The tracker logs the normalization parameters and the URI of the Amazon S3 bucket where the transformed dataset is stored. In a later section, we show how to find this information in Studio. In the next section, the tracker is used to track the experiment and trial runs.

```python
with Tracker.create(display_name="Preprocessing", sagemaker_boto_client=sm) as tracker:
    tracker.log_parameters(
        "normalization_mean": 0.1307,
        "normalization_std": 0.3081,
    )
    tracker.log_input(name="mnist-dataset", media_type="s3/uri", value=inputs)
```

### Create and Track an Experiment

The following procedure creates and tracks an experiment to determine the effect of the model's `num_hidden_channel` hyperparameter. As part of the experiment, five trials are created inside a loop, one for each value of the `num_hidden_channel` hyperparameter. Later in the notebook, you'll compare the results of these five trials.

1. In the left sidebar of Studio, choose the **SageMaker Experiment List** icon (agnost) to display the experiments browser.
2. Run the following cell.

```python
mnist_experiment = Experiment.create(
    experiment_name=f"mnist-hand-written-digits-classification-{int(time.time())}",
    description="Classification of mnist hand-written digits",
    sagemaker_boto_client=sm)
print(mnist_experiment)
```

Output:

```
Experiment(sagemaker_boto_client=<botocore.client.SageMaker object at 0x7f7152b326d8>,
    experiment_name='mnist-hand-written-digits-classification-1575947870',
    description='Classification of mnist hand-written digits',
```

After the code runs, the experiments list contains an entry for the experiment. It might take a moment to display and you might have to refresh the experiments list. Your screen should look similar to the following:
3. Run the following cell.

```python
preprocessing_trial_component = tracker.trial_component
```

After the code runs, the experiments list contains an entry labeled **Unassigned trial components**. The trial component entry is the data preprocessing step previously created. Double-click the trial component for verification. The trial component isn't associated with an experiment at this time. Your screen should look similar to the following:

4. Choose the **Home** icon in the navigation breadcrumb at the top on the experiments browser. From there, double-click on your experiment to display a list of the trials in the experiment.

5. The following code create trials for the experiment. Each trial trains a model using a different number for the `num_hidden_channel` hyperparameter. The preprocessing trial component is added to each trial for complete tracking (for example, for auditing purposes). The code also specifies definitions for the following metrics:

- Train loss
- Test loss
- Test accuracy

The definitions tell SageMaker to capture those metrics from the algorithm's log output. The metrics are used later to evaluate and compare the models.

```python
preprocessing_trial_component = tracker.trial_component

for i, num_hidden_channel in enumerate([2, 5, 10, 20, 32]):
    trial_name = f"cnn-training-job-{num_hidden_channel}-hidden-channels-{int(time.time())}"  
    cnn_trial = Trial.create(
        trial_name=trial_name,
        experiment_name=mnist_experiment.experiment_name,
        sagemaker_boto_client=sm,
    )
    hidden_channel_trial_name_map[num_hidden_channel] = trial_name
    cnn_trial.add_trial_component(preprocessing_trial_component)

estimator = PyTorch(
```
hyperparameters={
    'hidden_channels': num_hidden_channel,
    ...,
},
metric_definitions=[
    {'Name': 'train:loss', 'Regex': 'Train Loss: \(.*?\)\;'},
    {'Name': 'test:loss', 'Regex': 'Test Average loss: \(.*?\)\;'},
    {'Name': 'test:accuracy', 'Regex': 'Test Accuracy: \(.*?\)\%'},
],
enable_sagemaker_metrics=True,
)
cnn_training_job_name = "cnn-training-job-{}'.format(int(time.time()))
estimator.fit(
    inputs={'training': inputs},
    job_name=cnn_training_job_name,
    experiment_config={
        'TrialName': cnn_trial.trial_name,
        'TrialComponentDisplayName': 'Training',
    },
)

The trial list automatically updates as each training job runs. It takes a few minutes for each trial to be displayed. Your screen should look similar to the following:

![Image of trial list](image)

**Compare and Analyze Trials**

This section deviates from the notebook and shows you how to compare and analyze the trained models using the SageMaker Studio UI.

**To view the details of a trial**

1. Double-click one of the trials to display a list of the trial components associated with the trial. There's a preprocessing job and training job for each trial. Double-click one of the components to open a new tab that displays information about each component.

2. Under **Trial stages**, choose **Preprocessing**. On the **Describe Trial Component** menu, choose **Parameters** to display the normalization parameters that were previously logged. Next, choose **Artifacts** to display the URI of the Amazon S3 bucket where the transformed dataset was stored.

3. Under **Trial stages**, choose **Training**. On the **Describe Trial Component** menu, choose the following items to display information about the training job trial component.

   - **Metrics** – test:loss, test:accuracy, and train:loss
   - **Parameters** – hyperparameter values and instance information
   - **Artifacts** – Amazon S3 storage for the input dataset and the output model
To view a list of trials ordered by test: accuracy

1. Choose the experiment name on the navigation breadcrumb above TRIAL COMPONENTS to display the trial list.

2. Choose all five trials. Hold down the CTRL/CMD key and select each trial. Right-click the selection and then choose Open in trial component list. A new tab opens that displays each trial and trial component.

3. If the TABLE PROPERTIES pane isn't open, choose the Settings icon (=email) in the upper right corner to open it. Deselect everything except Trial, Metrics, and Training job. Choose the Settings icon to close the pane.

4. Choose the test:accuracy column header to sort the list by decreasing maximum test accuracy. Your screen should look similar to the following:

To view a chart of test:loss versus num_hidden_channel

1. In the TRIAL COMPONENTS pane, choose all five trials and then choose Add chart. Select inside the chart area to open the preferences pane for CHART PROPERTIES.

2. In CHART PROPERTIES, choose the following:
   - Data type - Summary statistics
   - Chart type - Line
   - X-axis - hidden-channels
   - Y-axis - test:lost_last
   - Color - None

   Your screen should look similar to the following:
Search Experiments Using Amazon SageMaker Studio

You can search your experiments, trials, and trial components using the experiments browser. After choosing the entities, you can search on properties of these entities on the Amazon SageMaker Leaderboard.

**Topics**
- Search Experiments, Trials, and Trial Components (p. 859)
- Search the SageMaker Studio Leaderboard (p. 860)
- Search by Tag (p. 863)

**Search Experiments, Trials, and Trial Components**

You can search and create multiple filters to display a subset of experiments, trials, and trial components in the experiments browser. After you search and filter your list, you can open the entities in the Leaderboard and search on additional properties.

**To search for experiments, trials, and trial components, and apply filters**

For more information about the following steps, along with screenshots, see Search the SageMaker Studio Leaderboard (p. 860).

1. In the experiments browser, navigate to the entity type you want to search for: experiment, trial, or trial component.
2. Place your cursor in the search box to display the list of column that are searchable.
3. Choose the column that you want to search on and one of the following dialog boxes opens that's specific to the column you chose:
   - **String** – Enter a text string in the search dialog box that opens. Enter a string of at least three characters that's part of the component's property value. This filter limits the list to those components with a property value that contains the text string.
• **Discrete values** – Choose one or more property values from the list. This filter limits the list to those components with a property value that matches at least one of the chosen values.

• **Date** – Enter a date in **From Date** and **To Date**, or select the dates from the calendar. This filter limits the list to those components that were created or last modified in the specified date range.

• **Tag** – In **Search tag key**, start entering the tag’s key. A list of tag keys is filtered to only those keys with a name that starts with the text string that you enter. Choose the tag key that you want to search on from the list. In **Search tag value**, enter the complete tag value, and then choose **Apply**.

4. To create the filter, choose **Apply**.

5. To apply additional filters, repeat the preceding steps. You can have a maximum of 20 filters. An entity must match all filters to be displayed.

**Search the SageMaker Studio Leaderboard**

The Amazon SageMaker Studio trial components list is referred to as the Leaderboard. You can use the Leaderboard to compare your trials and experiments. The Leaderboard lists properties of the trials, such as the trial status, debugger status, tags, metrics, hyperparameters, and input and output artifacts.

You can search the Leaderboard to find specific trial components. You apply the results of a search to create a filter that displays only those components that satisfy the filter. You can apply up to 20 filters. A component is displayed only if it matches all filters. To remove a filter, choose the **X** that displays to the right of the filter.

**To search the Leaderboard and apply filters**

1. If the **TABLE PROPERTIES** pane isn’t open in SageMaker Studio, choose the **Settings** icon (اداتי) in the upper-right corner to open it.

2. In the **Settings** pane, select the check boxes for the columns that you want to view, and then choose the **Settings** icon to close the pane.

3. Place your cursor in the **Search column name** box to display the following list of **TABLE PROPERTIES** column names that are searchable:

   **Summary section**
   
   • Trial component name
   • Trial component type
   • Created
   • Last modified
   • Created by
   • Tags

   **Detail sections (all columns)**
   
   • Metrics
   • Parameters
   • Input artifacts
   • Output artifacts
4. To limit the column names when you search the detail sections, start typing the column name. The list displays only those columns with a name that starts with the text string that you enter.

5. Choose the column that you want to search on. This opens a dialog that's specific to the type of data in that column.

6. Enter your search criteria in the dialog for the appropriate data type:

   - **String** – Enter a text string in the search dialog box that opens. Enter a string of at least three characters that's part of the component's property value. This filter limits the list to those components with a property value that contains the text string.

   - **Discrete values** – Choose one or more of the property values from the list. This filter limits the list to those components with a property value that matches at least one of the chosen values. For the Trial component type column, **Others** matches components with no value in the column.
• **Date** – Enter a date in **From Date** and **To Date** or select the dates from the calendar. This filter limits the list to those components that were created or last modified in the specified data range.

• **Created by** – Select **Me** or **Everyone else**. This filter limits the list to those components that were created by the currently signed-in user or everyone but the currently signed-in user.

• **Tag** – Choose the tag to search. For more information, see Search by Tag (p. 863).

• **Detail columns** – Specify the conditions that the component's property value must satisfy.
7. To create the filter, choose **Apply**.
8. To apply additional filters, repeat the preceding steps. A component must match all filters to be displayed. You can have a maximum of 20 filters.

**Search by Tag**

You can add searchable tags to experiments, trials, and trial components when the entities are created or afterwards using the **AddTags** API. A tag consists of a unique case-sensitive key and an optional value. Multiple tags can be added to an entity. You can add the same tag to multiple entities.

**To search by tag and apply filters**

1. If the **TABLE PROPERTIES** pane isn’t open in SageMaker Studio, choose the **Settings** icon (⚙️) in the upper-right corner to open it.
2. In the **Settings** pane, select **Tags** if it isn’t selected, and then choose the **Settings** icon to close the pane.
3. Place your cursor in the **Search column name** box and then choose **Tags**.
4. In **Search tag key**, start entering the tag’s key. A list of tag keys displays only those keys with a name that starts with the text string that you enter.
5. Choose the tag key that you want to search on from the list.
6. In **Search tag value**, enter the complete tag value, and then choose **Apply**.

7. The trial component list displays only those components that have tags that match the key-value pair you chose.

8. To add more tag filters, repeat the previous steps. A tag must match all filters for the component to be displayed.

### Clean Up Amazon SageMaker Experiment Resources

To avoid incurring unnecessary charges, delete the Amazon SageMaker Experiment resources you no longer need. You can't delete Experiment resources through the SageMaker Management Console or the Amazon SageMaker Studio UI. This topic shows you how to clean up these resources using Boto3 and the Experiments SDK. For more information about the Experiments SDK, see [sagemaker-experiments](https://docs.aws.amazon.com/sagemaker/latest/dg/experiments.html).

To delete the experiment, you must delete all trials in the experiment. To delete a trial, you must remove all trial components from the trial. To delete a trial component, you must remove the component from all trials.

**Note**

Trial components can exist independent of trials and experiments. You do not have to delete them. If you want to reuse them, comment out `tc.delete()` in the code.
Clean Up Using the Experiments SDK

To clean up using the Experiments SDK

```python
import sys
!{sys.executable} -m pip install sagemaker-experiments

import time
from smexperiments.experiment import Experiment
from smexperiments.trial import Trial
from smexperiments.trial_component import TrialComponent

Define cleanup_sme_sdk

def cleanup_sme_sdk(experiment):
    for trial_summary in experiment.list_trials():
        trial = Trial.load(trial_name=trial_summary.trial_name)
        for trial_component_summary in trial.list_trial_components():
            tc = TrialComponent.load(
                trial_component_name=trial_component_summary.trial_component_name)
            trial.remove_trial_component(tc)
            try:
                # comment out to keep trial components
                tc.delete()
            except:
                # tc is associated with another trial
                continue
            # to prevent throttling
            time.sleep(.5)
            trial.delete()
            experiment_name = experiment.experiment_name
            experiment.delete()
    print(f"\nExperiment {experiment_name} deleted")

Call cleanup_sme_sdk

experiment_to_cleanup = Experiment.load(
    # Use experiment name not display name
    experiment_name="experiment-name")
cleanup_sme_sdk(experiment_to_cleanup)
```

Clean Up Using the Python SDK (Boto3)

To clean up using Boto3

```python
import boto3
sm = boto3.Session().client('sagemaker')

Define cleanup_boto3

```
def cleanup_boto3(experiment_name):
    trials = sm.list_trials(ExperimentName=experiment_name)['TrialSummaries']
    print('TrialNames: ')
    for trial in trials:
        trial_name = trial['TrialName']
        print(f'\n{trial_name}

        components_in_trial = sm.list_trial_components(TrialName=trial_name)
        print('TrialComponentNames: ')
        for component in components_in_trial['TrialComponentSummaries']:
            component_name = component['TrialComponentName']
            print(f'\t{component_name}')
            sm.disassociate_trial_component(TrialComponentName=component_name, TrialName=trial_name)
        try:
            # comment out to keep trial components
            sm.delete_trial_component(TrialComponentName=component_name)
        except:
            # component is associated with another trial
            continue
            # to prevent throttling
            time.sleep(.5)
        sm.delete_trial(TrialName=trial_name)
    sm.delete_experiment(ExperimentName=experiment_name)
    print(f'\nExperiment {experiment_name} deleted')

Call cleanup_boto3

# Use experiment name not display name
experiment_name = "experiment-name"
cleanup_boto3(experiment_name)

Search Using the Amazon SageMaker Console and API

Developing a machine learning model typically requires extensive experimenting with different datasets, algorithms, and hyperparameter values. To manage up to thousands of machine learning model experiments, use the search capabilities in SageMaker.

You can use SageMaker search to:

- Organize, find, and evaluate training jobs using properties, hyperparameters, performance metrics, or any metadata.
- Find the best performing model by reviewing training job and model metrics, such as training loss or validation accuracy.
- Trace a model's lineage to the training job and its related resources, such as the training datasets.

This topic covers searching from the SageMaker console and the SageMaker API. For information on searching in Amazon SageMaker Studio, see Search Experiments Using Studio (p. 859).

Topics

- Sample Notebooks for Managing ML Experiments (p. 867)
- Organize, Find, and Evaluate Training Jobs (Console) (p. 867)
- Find and Evaluate Training Jobs (API) (p. 869)
- Verify the Datasets Used by Your Training Jobs (p. 870)
Sample Notebooks for Managing ML Experiments

For a sample notebook that uses Amazon SageMaker model tracking capability to manage ML experiments, see Managing ML Experimentation using Amazon SageMaker Model Tracking Capability. For instructions on how to create and access Jupyter notebook instances that you can use to run the example in SageMaker, see Use Amazon SageMaker Notebook Instances (p. 124). After you have created a notebook instance and opened it, choose the SageMaker Examples tab to see a list of all of the SageMaker samples. The notebook for managing ML experiments is located in the Advanced Functionality section. To open a notebook, choose its Use tab, and choose Create copy. If you have questions, post them on the Amazon Machine Learning Developer Forum.

Organize, Find, and Evaluate Training Jobs (Console)

To organize training jobs, assign one or more tags to them.

To find a specific training job, model, or resource, use model tracking to search on keywords assigned to any searchable items. Searchable items include training jobs, models, hyperparameters, metadata, tags, and URLs. To refine your tracking results, you can search using multiple criteria.

To choose the best model for deployment, evaluate how all models performed against one or more metrics. You can use model tracking results to list, sort, and evaluate the performance of the models in your experiments.

Topics
- Use Tags to Track Training Jobs (Console) (p. 867)
- Find Training Jobs (Console) (p. 868)
- Evaluate Models (Console) (p. 868)

Use Tags to Track Training Jobs (Console)

To group training jobs, create tags with descriptive keys and a value. For example, create tag keys for: project, owner, customer, and industry.

Add tags to training jobs (console)

1. Open the Amazon SageMaker console.
2. In the navigation pane, choose Training jobs and Create training job.
3. Scroll to the bottom of the page and enter a key and value for the tag.

4. To add another tag, choose Add tag, and add another key-value pair.
Find Training Jobs (Console)

You can search for training jobs using a variety of job attributes. Note that some search parameters appear only if you have created a training job with that attribute. For example, Tags appears only if you have added a tag for a training job.

To find training jobs (console)
1. Open the Amazon SageMaker console.
2. In the navigation pane, choose Search.
3. Add Parameters.
   a. In the search box, enter a parameter and choose a parameter type, for example TrainingJobName.
   b. Choose a conditional operation. For numeric values, use operators such as is equals to, lesser than, or or greater than. For text-based values, use operators such as equals to or contains.
   c. Enter a value for the parameter.
4. (Optional) To refine your search, add additional search criteria. Choose Add row and enter the parameter values.
5. Choose Search.

Evaluate Models (Console)

To evaluate a model's performance, review its metadata, hyperparameters, and metrics. To highlight metrics, adjust the view to show only metrics and important hyperparameters.

To evaluate a model (console)
1. Open the Amazon SageMaker console.
2. In the navigation pane, choose Search and search for training jobs by specifying relevant parameters. The results are displayed in a table.

Results: Training jobs

<table>
<thead>
<tr>
<th>HyperParameter</th>
<th>HyperParameter</th>
<th>Metric</th>
<th>Metric</th>
<th>Metric</th>
</tr>
</thead>
<tbody>
<tr>
<td>mini_batch_size</td>
<td>predictor_type</td>
<td>train:binary_f_beta</td>
<td>train:progress</td>
<td>train:objective_loss</td>
</tr>
<tr>
<td>300</td>
<td>binary_classifier</td>
<td>0.9666395180737793</td>
<td>100</td>
<td>0.02381423674523</td>
</tr>
<tr>
<td>100</td>
<td>binary_classifier</td>
<td>0.9652714133262634</td>
<td>100</td>
<td>0.02350491285324</td>
</tr>
<tr>
<td>200</td>
<td>binary_classifier</td>
<td>0.964742698478699</td>
<td>100</td>
<td>0.02325980737805</td>
</tr>
</tbody>
</table>

3. Open the preferences window by choosing the settings icon in the search results table.
4. To show or hide a hyperparameter or metric, turn it on or off by choosing **Hyperparameter** or **Metric**
   .
5. Make necessary changes, then choose **Update view**.
6. After viewing metrics and important hyperparameters, you can compare and contrast the result. Then, you can choose the best model to host or investigate the models that are performing poorly.

**Find and Evaluate Training Jobs (API)**

To find the training jobs, you can use the Search API.

**Topics**

- Find Training Jobs (API) (p. 869)
- Evaluate Models (API) (p. 869)
- Get Suggestions for a Search (API) (p. 870)

**Find Training Jobs (API)**

To find training jobs, create a search parameter using the `search_params` parameter. Then use the `search` function in the `smclient` subprocess in the AWS SDK for Python (Boto3).

The following example shows how to use the **Search API** to find training jobs.

```python
import boto3

search_params={
    "MaxResults": 10,
    "Resource": "TrainingJob",
    "SearchExpression": {
        "Filters": [{
            "Name": "Tags.Project",
            "Operator": "Equals",
            "Value": "Project_Binary_Classifier"
        }],
        "SortBy": "Metrics.train:binary_classification_accuracy",
        "SortOrder": "Descending"
    }
}

smclient = boto3.client(service_name='sagemaker')
results = smclient.search(**search_params)
```

**Evaluate Models (API)**

To evaluate models, run a search as described in **Find Training Jobs (API)** (p. 869), review model metrics, then, use the AWS SDK for Python (Boto3) to create a table and plot it.

The following example shows how to evaluate models and to display the results in a table.

```python
import pandas

headers=["Training Job Name", "Training Job Status", "Batch Size", "Binary Classification Accuracy"]
rows=[]
for result in results['Results']:
    trainingJob = result['TrainingJob']
    metrics = trainingJob['FinalMetricDataList']
```
Get Suggestions for a Search (API)

To get suggestions for a search, use the `GetSearchSuggestions` API.

The following example for AWS SDK for Python (Boto3) is a `get_search_suggestions` request for items containing linear.

```python
search_suggestion_params={
    "Resource": "TrainingJob",
    "SuggestionQuery": {
        "PropertyNameQuery": {
            "PropertyNameHint": "linear"
        }
    }
}
```

The following is an example response for a `get_search_suggestions` request.

```json
{
    'PropertyNameSuggestions': [
        {'PropertyName': 'hyperparameters.linear_init_method'},
        {'PropertyName': 'hyperparameters.linear_init_value'},
        {'PropertyName': 'hyperparameters.linear_init_sigma'},
        {'PropertyName': 'hyperparameters.linear_lr'},
        {'PropertyName': 'hyperparameters.linear_wd'}
    ]
}
```

After getting search suggestions, you can use one of the property names in a search.

Verify the Datasets Used by Your Training Jobs

You can use model tracking capability to verify which datasets were used in training, where holdout datasets were used, and other details about training jobs. For example, use model tracking capability to verify that a specific dataset was used in a training job for an audit or to verify compliance.

To check whether a specific dataset was used in a training job, you search for the URL to its location in Amazon Simple Storage Service (Amazon S3). Model tracking capability returns the training jobs that used the dataset that you specify. If your search doesn't return the dataset (the result is empty), the dataset wasn't used in a training job. An empty result confirms, for example, that a holdout dataset wasn't used.

Trace Model Lineage

You can use model tracking capability to get information about the lineage of training jobs and the model resources that were used for them, including the dataset, algorithm, hyperparameters, and metrics. For example, if you find that the performance of a hosted model has declined, you can review its training job and the resources it used to determine what's causing the problem.
Topics

- Trace Model Lineage (Console) (p. 871)
- Trace Model Lineage (API) (p. 871)

Trace Model Lineage (Console)

To trace a model's lineage (console)

1. Open the Amazon SageMaker console.
2. In the navigation pane, choose Endpoints, and choose the relevant endpoint.
3. Scroll to the Endpoint configuration settings section. This section lists all of the model versions deployed at the endpoint, with a hyperlink to the training job that created each.

Trace Model Lineage (API)

To trace a model's lineage, get the model's name, then use it to search for training jobs.

The following example shows how to trace a model's lineage using the API.

```python
# Get the name of model deployed at endpoint
endpoint_config = smclient.describe_endpoint_config(EndpointConfigName=endpointName)
model_name = endpoint_config['ProductionVariants'][0]['ModelName']

# Get the model's name
model = smclient.describe_model(ModelName=model_name)

# Search the training job by the location of model artifacts in Amazon S3
search_params = {
    "MaxResults": 1,
    "Resource": "TrainingJob",
    "SearchExpression": {
        "Filters": [
            {
                "Name": "ModelArtifacts.S3ModelArtifacts",
                "Operator": "Equals",
                "Value": model['PrimaryContainer']['ModelDataUrl']
            }
        ]
    }
}
results = smclient.search(**search_params)
```

After finding the training job, you can review the resources used to train the model.

Amazon SageMaker Debugger

Debug, monitor, and profile training jobs in real time, detect non-converging conditions, optimize resource utilization by eliminating bottlenecks, improve training time and reduce costs of your machine learning models using Amazon SageMaker Debugger.

New Debugger Features

SageMaker Debugger profiles and debugs your training jobs to improve the performance of machine learning models on compute resource utilization and model predictions. You can now achieve target accuracy faster using the following new Debugger features:
• Gain insights into your training jobs through SageMaker Studio Debugger insights dashboard and comprehensive reports autogenerated by Debugger. To learn more about the Debugger insights dashboard, see Debugger for Insights on Studio (p. 890). To learn more about the reports, see SageMaker Debugger Profiling Report (p. 903) and SageMaker Debugger XGBoost Training Report (p. 913).

• Use Debugger ProfilerRule (p. 949) for monitoring and profiling to automatically detect system bottlenecks in real time. To learn more, see Configure Debugger Built-in Rules (p. 941) and List of Debugger Built-in Rules (p. 948).

• Use Debugger’s built-in actions to receive notifications or automatically stop training jobs when Debugger Rule (p. 949) finds training issues in real time. To learn more, see Debugger Built-in Actions for Rules (p. 999).

• Profile system resource usage and training operations, correlate system bottleneck issues with the training operations, and merge timelines with various profiler options. To learn more, see Configure Debugger Using Amazon SageMaker Python SDK (p. 926) and Analyze Data Using the SMDebug Client Library (p. 1007).

Debugger Features

Debugger provides the following features:

• SageMaker Studio Debugger dashboards in Studio Experiments and trials (p. 888)
• Comprehensive training reports autogenerated by Debugger (p. 902)
• Three types of built-in rules: monitoring system bottlenecks, profiling model framework operations, and debugging model parameters (p. 948)
• Debugger APIs in the SageMaker Python SDK to help configure SageMaker training jobs with monitoring, profiling, and debugging features (p. 926)
• Automated notifications and training job termination using Debugger built-in actions or creating customized actions in conjunction with Amazon CloudWatch Events and AWS Lambda (p. 998)
• The SMDebug client library to support model performance and parameter analysis (p. 1007)

To see a list of SageMaker machine learning frameworks and algorithms that Debugger supports, see Supported Frameworks and Algorithms (p. 873).

To learn more about the architecture of Debugger and how it works, see Debugger Architecture and Best Practices (p. 874).
For Debugger tutorials and notebook examples, see Get Started with Debugger Tutorials (p. 878).

**Supported Frameworks and Algorithms**

The following table shows SageMaker machine learning frameworks and algorithms supported by Debugger.

<table>
<thead>
<tr>
<th>SageMaker frameworks and algorithms</th>
<th>Performance optimization</th>
<th>Model optimization</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Monitoring system bottlenecks</td>
<td>Profiling framework operations</td>
</tr>
<tr>
<td>TensorFlow</td>
<td>All AWS Deep learning containers</td>
<td>AWS TensorFlow deep learning containers 2.3.1 or later</td>
</tr>
<tr>
<td>PyTorch</td>
<td>AWS PyTorch deep learning containers 1.6.0 or later</td>
<td>AWS PyTorch deep learning containers 1.5.0 or later</td>
</tr>
<tr>
<td>MXNet</td>
<td>-</td>
<td>AWS MXNet deep learning containers 1.6.0 or later</td>
</tr>
<tr>
<td>XGBoost</td>
<td>1.0-1, 1.2-1</td>
<td>-</td>
</tr>
<tr>
<td>SageMaker generic estimator</td>
<td>SageMaker Debugger built-in algorithms using image URIs</td>
<td>Custom training containers (p. 995) (with TensorFlow, PyTorch, MXNet, and XGBoost training scripts with a manual hook registration)</td>
</tr>
<tr>
<td></td>
<td>Custom training containers (p. 995) (with the AWS deep learning container images, public Docker images, or your own Docker images)</td>
<td></td>
</tr>
</tbody>
</table>

If the framework or algorithm that you want to train and debug is not listed in the table, go to AWS Discussion Forum and leave feedback on SageMaker Debugger.

**Use Debugger with Custom Training Containers**

Bring your training containers to SageMaker and gain insights into your training jobs using Debugger. Maximize your work efficiency by optimizing your model on EC2 instances using the aforementioned SageMaker Debugger features.

For more information about how to push your training container to Amazon Elastic Container Registry (ECR) and debug, see Use Debugger with Custom Training Containers (p. 995).

**Debugger Open Source GitHub Repositories**

Debugger APIs are provided through the SageMaker Python SDK and designed to construct Debugger hook and rule configurations for the SageMaker CreateTrainingJob and DescribeTrainingJob API operations. The SMDebug client library provides tools to register hooks and access the training data through its trial feature, all through its flexible and powerful API operations. It supports the machine learning frameworks TensorFlow, PyTorch, MXNet, and XGBoost on Python 3.6 and later.

If you want to find direct resources of the Debugger and SMDebug API operations, see the following links:
If you use the Java SDK to conduct SageMaker training jobs and want to configure Debugger APIs, see the following references:

- Amazon SageMaker Debugger API Operations (p. 1015)
- Use the SageMaker and Debugger Configuration API Operations to Create, Update, Debug, and Profile Your Training Job (p. 1016)

This SageMaker Debugger developer guide walks you through the following topics.

Topics
- Debugger Architecture and Best Practices (p. 874)
- Get Started with Debugger Tutorials (p. 878)
- Debugger on Studio (p. 888)
- SageMaker Debugger Interactive Reports (p. 902)
- Configure Debugger Using Amazon SageMaker Python SDK (p. 926)
- List of Debugger Built-in Rules (p. 948)
- Create Debugger Custom Rules for Training Job Analysis (p. 993)
- Use Debugger with Custom Training Containers (p. 995)
- Action on Amazon SageMaker Debugger Rules (p. 998)
- Analyze Data Using the SMDebug Client Library (p. 1007)
- Amazon SageMaker Debugger Advanced Topics and Reference Documentation (p. 1015)

Debugger Architecture and Best Practices

This topic walks you through a high-level overview of the Amazon SageMaker Debugger workflow and best practices.

Amazon SageMaker Debugger Architecture

Debugger supports profiling functionality for performance optimization, identifying model performance issues and providing suggestions to optimize hardware resource utilization at scale. Debugger's debugging functionality for model optimization is about analyzing training progress, watching for issues while minimizing the loss functions using optimization algorithms, such as gradient descent and its variations. The following diagram shows the architecture of SageMaker Debugger. The blocks with bold boundary lines are what Debugger manages to analyze your training job.
Debugger stores the following data from your training jobs in your secured Amazon S3 bucket:

- **System metrics** – Hardware resource utilization data, such as CPU, GPU, CPU and GPU memory, Network, and data input and output (I/O) metrics.
- **Framework metrics** – Model training information tracking each operation per call or sampling, such as convolutional operations in forward pass, batch normalization operations in backward pass, data loader processes between steps, and gradient descent algorithm operations to calculate and update the loss function.
- **Output tensors** – Collections of model parameters that are continuously updated during the back propagation and optimization process of training machine learning and deep learning models. It includes scalar values (accuracy and loss) and matrices (weights, gradients, input layers, and output layers).
Note
By default, Debugger monitors and debugs SageMaker training jobs without any Debugger-
specific parameters configured in SageMaker estimators. Debugger collects system metrics
every 500 milliseconds and basic output tensors (scalar outputs such as loss and accuracy)
every 500 steps. It also runs the ProfilerReport rule to analyze the system metrics and
aggregate Studio Debugger insights dashboard and a profiling report. Debugger saves the
output data in your secured S3 bucket.

The Debugger built-in rules run on processing containers, which are designed to evaluate machine
learning models by processing the training data collected in your S3 bucket (see Process Data and
Evaluate Models). The built-in rules are fully managed by Debugger. You can also create your own rules
customized to your model to watch for any issues you want to monitor.

See the following topics for best practices to improve the performance of your model using SageMaker Debugger.

Topics
• Choose a Machine Learning Framework (p. 876)
• Use Studio Debugger Insights Dashboard (p. 876)
• Download Debugger Reports and Gain More Insights (p. 876)
• Capture Data from Your Training Job and Save Data to Amazon S3 (p. 877)
• Analyze the Data with a Fleet of Debugger Built-in Rules (p. 877)
• Take Actions Based on the Built-in Rule Status (p. 877)
• Dive Deep into the Data Using the SMDebug Client Library (p. 877)
• Monitoring System Utilization and Detect Bottlenecks (p. 877)
• Profiling Framework Operations (p. 878)
• Debugging Model Parameters (p. 878)

Choose a Machine Learning Framework
You can choose a machine learning framework and use SageMaker pre-built training containers or your
own containers. Use Debugger to detect training and performance issues, and analyze training progress
of your training job in SageMaker. SageMaker provides you options to use pre-built containers that are
prepared for a number of machine learning framework environments to train your model on Amazon
EC2. Any training job can be adapted to run in AWS Deep Learning Containers, SageMaker training
containers, and custom containers. To learn more, see Configure Debugger Using Amazon SageMaker
Python SDK (p. 926) and Use Debugger with Custom Training Containers (p. 995).

Use Studio Debugger Insights Dashboard
With Studio Debugger insights dashboard, you are in control of your training jobs. Use the Studio
Debugger dashboards to keep your model performance on Amazon EC2 instances in control and
optimized. For any SageMaker training jobs running on Amazon EC2 instance, Debugger monitors
resource utilization and basic model output data (loss and accuracy values). Through the Studio
Debugger dashboards, gain insights into your training jobs and improve your model training
performance. To learn more, see Debugger on Studio (p. 888).

Download Debugger Reports and Gain More Insights
You can view aggregated results and gain insights in Debugger reports. Debugger aggregates training
and profiling results collected from the built-in rule analysis into a report per training job. You can find
Capture Data from Your Training Job and Save Data to Amazon S3

You can use a Debugger hook to save output tensors. After you choose a container and a framework that fit your training script, use a Debugger hook to configure which tensors to save and to which directory to save them, such as an Amazon S3 bucket. A Debugger hook helps you to build the configuration and to keep it in your account to use in subsequent analyses, where it is secured for use with the most privacy-sensitive applications. To learn more, see Configure Debugger Hook to Save Tensors (p. 934).

Analyze the Data with a Fleet of Debugger Built-in Rules

You can use Debugger built-in rules to inspect tensors in parallel with a training job. To analyze the training performance data, Debugger provides built-in rules that watch for abnormal training process behaviors. For example, a Debugger rule detects issues when the training process suffers from system bottleneck issues or training issues, such as vanishing gradients, exploding tensors, overfitting, or overtraining. If necessary, you can also build customized rules by creating a rule definition with your own criteria to define a training issue. To learn more about the Debugger rules, see Configure Debugger Built-in Rules (p. 941) for detailed instructions of using the Amazon SageMaker Python SDK. For a full list of the Debugger built-in rules, see List of Debugger Built-in Rules (p. 948). If you want to create a custom rule, see Create Debugger Custom Rules for Training Job Analysis (p. 993).

Take Actions Based on the Built-in Rule Status

You can use Debugger with Amazon CloudWatch Events and AWS Lambda. You can automate actions based on the rule status, such as stopping training jobs early and setting up notifications through email or text. When the Debugger rules detect problems and triggers an "IssuesFound" evaluation status, CloudWatch Events detects the rule status changes and invokes the Lambda function to take actions. To configure automated actions to your training issues, see Create Actions on Rules Using Amazon CloudWatch and AWS Lambda (p. 1002).

Dive Deep into the Data Using the SMDebug Client Library

You can use the SMDebug tools to access and analyze training data collected by Debugger. The TrainingJob and create_trial classes load the metrics and tensors saved by Debugger. These classes provide extended class methods to analyze the data in real time or after the training has finished. The SMDebug library also provides visualization tools: merge timelines of framework metrics to aggregate different profiling, line charts and heatmap to track the system utilization, and histograms to find step duration outliers. To learn more about the SMDebug library tools, see Analyze Data Using the SMDebug Client Library (p. 1007).

Monitoring System Utilization and Detect Bottlenecks

With Amazon SageMaker Debugger monitoring, you can measure hardware system resource utilization of Amazon EC2 instances. Monitoring is available for any SageMaker training job constructed with the SageMaker framework estimators (TensorFlow, PyTorch, and MXNet) and the generic SageMaker estimator (SageMaker built-in algorithms and your own custom containers). Debugger built-in rules for monitoring detect system bottleneck issues and notify you when they detect the bottleneck issues.

To learn how to enable Debugger system monitoring, see Configure Debugger Using Amazon SageMaker Python SDK (p. 926) and then Configure Debugger Monitoring Hardware System Resource Utilization (p. 930).
For a full list of available built-in rules for monitoring, see Debugger Built-in Rules for Monitoring Hardware System Resource Utilization (System Metrics) (p. 949).

Profiling Framework Operations

With Amazon SageMaker Debugger profiling you can profile deep learning frameworks operations. You can profile your model training with the SageMaker TensorFlow training containers, the SageMaker PyTorch framework containers, and your own training containers. Using the profiling feature of Debugger, you can drill down into the Python operators and functions that are executed to perform the training job. Debugger supports detailed profiling, Python profiling, data loader profiling, and Horovod distributed training profiling. You can merge the profiled timelines to correlate with the system bottlenecks. Debugger built-in rules for profiling watch framework operation related issues, including excessive training initialization time due to data downloading before training starts and step duration outliers in training loops.

To learn how to configure Debugger for framework profiling, see Configure Debugger Using Amazon SageMaker Python SDK (p. 926) and then Configure Debugger Framework Profiling (p. 930).

For a complete list of available built-in rules for profiling, see Debugger Built-in Rules for Profiling Model Performance Data (Framework Metrics) (p. 949).

Debugging Model Parameters

Debugging is available for deep learning frameworks using AWS Deep Learning Containers and the SageMaker training containers. For fully supported framework versions (see the versions at Supported Frameworks and Algorithms (p. 873)), Debugger automatically registers hooks to collect output tensors, and you can directly run your training script. For the versions with one asterisk sign, you need to manually register the hooks to collect tensors. Debugger provides preconfigured tensor collections with generalized names that you can utilize across the different frameworks. If you want to customize output tensor configuration, you can also use the CollectionConfig and DebuggerHookConfig API operations and the Amazon SageMaker Python SDK to configure your own tensor collections. Debugger built-in rules for debugging analyze the output tensors and identifies model optimization problems that blocks your model from minimizing the loss function. For example, the rules identify overfitting, overtraining, loss not decreasing, exploding tensors, and vanishing gradients.

To learn how to configure Debugger for debugging output tensors, see Configure Debugger Using Amazon SageMaker Python SDK (p. 926) and then Configure Debugger Hook to Save Tensors (p. 934).

For a full list of available built-in rules for debugging, see Debugger Built-in Rules for Debugging Model Training Data (Output Tensors) (p. 950).

Get Started with Debugger Tutorials

The following topics walk you through tutorials from the basics to advanced use cases of monitoring, profiling, and debugging SageMaker training jobs using Debugger. Explore the Debugger features and learn how you can debug and improve your machine learning models efficiently by using Debugger.

Topics
- Debugger Tutorial Videos (p. 878)
- Debugger Example Notebooks (p. 880)
- Debugger Advanced Demos and Visualization (p. 882)

Debugger Tutorial Videos

The following videos provide a tour of Amazon SageMaker Debugger capabilities using SageMaker Studio and SageMaker notebook instances.
Topics

- Analyze, Detect, and Get Alerted on Problems with Training Runs Using Amazon SageMaker Debugger (p. 879)
- Debug Models with Amazon SageMaker Debugger in Studio (p. 879)
- Deep Dive on Amazon SageMaker Debugger and SageMaker Model Monitor (p. 879)

Analyze, Detect, and Get Alerted on Problems with Training Runs Using Amazon SageMaker Debugger

Emily Webber, AWS Machine Learning Specialist | Length: 13 minutes 54 seconds

This tutorial video gives you a tour of Amazon SageMaker Debugger to capture, debug, and visualize model output data from a training model with MXNet. Learn how Amazon SageMaker Debugger makes the training process transparent by automatically capturing metrics, analyzing training runs, and detecting problems.

You can find the example notebook in this video at Visualizing Debugging Tensors of MXNet training in the Amazon SageMaker Examples GitHub repository.

Debug Models with Amazon SageMaker Debugger in Studio

Julien Simon, AWS Technical Evangelist | Length: 14 minutes 17 seconds

This tutorial video demonstrates how to use Amazon SageMaker Debugger to capture and inspect debugging information from a training model. The example training model used in this video is a simple convolutional neural network (CNN) based on Keras with the TensorFlow backend. SageMaker in a TensorFlow framework and Debugger enable you to build an estimator directly using the training script and debug the training job.

You can find the example notebook in the video in this Studio Demo repository provided by the author. You need to clone the debugger.ipynb notebook file and the mnist_keras_tf.py training script to your SageMaker Studio or a SageMaker notebook instance. After you clone the two files, specify the path keras_script_path to the mnist_keras_tf.py file inside the debugger.ipynb notebook. For example, if you cloned the two files in the same directory, set it as keras_script_path = "mnist_keras_tf.py".

Deep Dive on Amazon SageMaker Debugger and SageMaker Model Monitor

Julien Simon, AWS Technical Evangelist | Length: 44 minutes 34 seconds

This video session explores advanced features of Debugger and SageMaker Model Monitor that help boost productivity and the quality of your models. First, this video shows how to detect and fix training issues, visualize tensors, and improve models with Debugger. Next, at 22:41, the video shows how to monitor models in production and identify prediction issues such as missing features or data drift using SageMaker Model Monitor. Finally, it offers cost optimization tips to help you make the most of your machine learning budget.

You can find the example notebook in the video in this AWS Dev Days 2020 repository offered by the author.
Debugger Example Notebooks

SageMaker Debugger example notebooks are provided in the aws/amazon-sagemaker-examples repository. The Debugger example notebooks walk you through basic to advanced use cases of debugging and profiling training jobs.

It is recommended to run the example notebooks on SageMaker Studio or a SageMaker Notebook instance because most of the examples are designed for training jobs in the SageMaker ecosystem, including Amazon EC2, Amazon S3, and Amazon SageMaker Python SDK.

To clone the example repository to SageMaker Studio, follow the instructions at Amazon SageMaker Studio Tour.

To find the examples in a SageMaker Notebook instance, follow the instructions at SageMaker Notebook Instance Example Notebooks.

Important
To use the new Debugger features, you need to upgrade the SageMaker Python SDK and the SMDebug client library. In your iPython kernel, Jupyter notebook, or JupyterLab environment, run the following code to install the latest versions of the libraries and restart the kernel.

```python
import sys
import IPython
!{sys.executable} -m pip install -U sagemaker smdebug
IPython.Application.instance().kernel.do_shutdown(True)
```

Debugger Example Notebooks for Profiling Training Jobs

The following list shows Debugger example notebooks introducing adaptability of Debugger to monitor and profile training jobs of various machine learning models, datasets, and frameworks.

<table>
<thead>
<tr>
<th>Notebook Title</th>
<th>Framework</th>
<th>Model</th>
<th>Dataset</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amazon SageMaker Debugger Profiling Data Analysis</td>
<td>TensorFlow</td>
<td>Keras ResNet50</td>
<td>Cifar-10</td>
<td>This notebook provides an introduction to interactive analysis of profiled data captured by SageMaker Debugger. Explore the full functionality of the SMDebug interactive analysis tools.</td>
</tr>
<tr>
<td>Profile machine learning training with Amazon SageMaker Debugger</td>
<td>TensorFlow</td>
<td>1-D Convolutional Neural Network</td>
<td>IMDB dataset</td>
<td>Profile a TensorFlow 1-D CNN for sentiment analysis of IMDB data that consists of movie reviews labeled as having positive or negative sentiment. Explore the Studio Debugger insights and Debugger profiling report.</td>
</tr>
</tbody>
</table>
### Debugger Example Notebooks for Analyzing Model Parameters

The following list shows Debugger example notebooks introducing adaptability of Debugger to debug training jobs of various machine learning models, datasets, and frameworks.

<table>
<thead>
<tr>
<th>Notebook Title</th>
<th>Framework</th>
<th>Model</th>
<th>Dataset</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amazon SageMaker Debugger - Tensorflow 2.1</td>
<td>TensorFlow</td>
<td>ResNet50</td>
<td>Cifar-10</td>
<td>Use Amazon SageMaker Debugger hook configuration and built-in rules for debugging a model with Tensorflow 2.1 framework.</td>
</tr>
<tr>
<td>Visualizing Debugging Tensors of MXNet training</td>
<td>MXNet</td>
<td>Gluon Convolutional Neural Network</td>
<td>Fashion MNIST</td>
<td>Run a training job and configure SageMaker Debugger to store all tensors from this job. Afterwards we will visualize those tensors in our notebook.</td>
</tr>
<tr>
<td>Enable Spot Training with Amazon SageMaker Debugger</td>
<td>MXNet</td>
<td>Gluon Convolutional Neural Network</td>
<td>Fashion MNIST</td>
<td>Learn how Debugger collects tensor data from a training job on spot instance, and how to use Debugger built-in rules with managed spot training.</td>
</tr>
<tr>
<td>Debugging XGBoost Training Jobs with Amazon SageMaker Debugger Using Rules</td>
<td>XGBoost</td>
<td>XGBoost Regression</td>
<td>Abalone</td>
<td>Learn how to use Debugger hook and built-in rules for collecting and visualizing tensor data from an XGBoost regression model, such as loss values, features, and SHAP values.</td>
</tr>
</tbody>
</table>
To find advanced visualizations of model parameters and use cases, see the next topic at Debugger Advanced Demos and Visualization (p. 882).

**Debugger Advanced Demos and Visualization**

The following demos walk you through advanced use cases and visualization scripts using Debugger.

**Topics**
- Train and Tune Your Models with Amazon SageMaker Experiments and Debugger (p. 882)
- Using SageMaker Debugger to Monitor a Convolutional Autoencoder Model Training (p. 885)
- Using SageMaker Debugger to Monitor Attentions in BERT Model Training (p. 885)
- Using SageMaker Debugger to Visualize Class Activation Maps in Convolutional Neural Networks (CNNs) (p. 888)

**Train and Tune Your Models with Amazon SageMaker Experiments and Debugger**

*Dr. Nathalie Rauschmayr, AWS Applied Scientist | Length: 49 minutes 26 seconds*

Train and Prune Models with SageMaker Experiments and Debugger

Find out how Amazon SageMaker Experiments and Debugger can simplify the management of your training jobs. Amazon SageMaker Debugger provides transparent visibility into training jobs and saves training metrics into your Amazon S3 bucket. SageMaker Experiments enables you to call the training information as **trials** through SageMaker Studio and supports visualization of the training job. This helps you keep model quality high while reducing less important parameters based on importance rank.

This video demonstrates a *model pruning* technique that makes pre-trained ResNet50 and AlexNet models lighter and affordable while keeping high standards for model accuracy.

SageMaker Estimator trains those algorithms supplied from PyTorch model zoo in an AWS Deep Learning Containers with PyTorch framework, and Debugger extracts training metrics from the training process.

The video also demonstrates how to set up a Debugger custom rule to watch the accuracy of a pruned model, to trigger an Amazon CloudWatch event and an AWS Lambda function when the accuracy hits a threshold, and to automatically stop the pruning process to avoid redundant iterations.

Learning objectives are as follows:

- Learn how to use SageMaker to accelerate ML model training and improve model quality.
- Understand how to manage training iterations with SageMaker Experiments by automatically capturing input parameters, configurations, and results.
- Discover how Debugger makes the training process transparent by automatically capturing real-time tensor data from metrics such as weights, gradients, and activation outputs of convolutional neural networks.
- Use CloudWatch to trigger Lambda when Debugger catches issues.
- Master the SageMaker training process using SageMaker Experiments and Debugger.

You can find the notebooks and training scripts used in this video from SageMaker Debugger PyTorch Iterative Model Pruning.

The following image shows how the iterative model pruning process reduces the size of AlexNet by cutting out the 100 least significant filters based on importance rank evaluated by activation outputs and gradients.
The pruning process reduced the initial 50 million parameters to 18 million. It also reduced the estimated model size from 201 MB to 73 MB.

<table>
<thead>
<tr>
<th>Layer (type)</th>
<th>Output Shape</th>
<th>Param #</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conv2d-1</td>
<td>[-1, 58, 55, 55]</td>
<td>21,112</td>
</tr>
<tr>
<td>ReLU-2</td>
<td>[-1, 58, 55, 55]</td>
<td>0</td>
</tr>
<tr>
<td>MaxPool2d-3</td>
<td>[-1, 58, 27, 27]</td>
<td>0</td>
</tr>
<tr>
<td>Conv2d-4</td>
<td>[-1, 166, 27, 27]</td>
<td>240,866</td>
</tr>
<tr>
<td>ReLU-5</td>
<td>[-1, 166, 27, 27]</td>
<td>0</td>
</tr>
<tr>
<td>MaxPool2d-6</td>
<td>[-1, 166, 13, 13]</td>
<td>0</td>
</tr>
<tr>
<td>Conv2d-7</td>
<td>[-1, 305, 13, 13]</td>
<td>455,975</td>
</tr>
<tr>
<td>ReLU-8</td>
<td>[-1, 305, 13, 13]</td>
<td>0</td>
</tr>
<tr>
<td>Conv2d-9</td>
<td>[-1, 206, 13, 13]</td>
<td>565,676</td>
</tr>
<tr>
<td>ReLU-10</td>
<td>[-1, 206, 13, 13]</td>
<td>0</td>
</tr>
<tr>
<td>Conv2d-11</td>
<td>[-1, 217, 13, 13]</td>
<td>402,535</td>
</tr>
<tr>
<td>ReLU-12</td>
<td>[-1, 217, 13, 13]</td>
<td>0</td>
</tr>
<tr>
<td>MaxPool2d-13</td>
<td>[-1, 217, 6, 6]</td>
<td>0</td>
</tr>
<tr>
<td>AdaptiveAvgPool2d-14</td>
<td>[-1, 217, 6, 6]</td>
<td>0</td>
</tr>
<tr>
<td>Dropout-15</td>
<td>[-1, 7812]</td>
<td>0</td>
</tr>
<tr>
<td>Linear-16</td>
<td>[-1, 4096]</td>
<td>32,002,048</td>
</tr>
<tr>
<td>ReLU-17</td>
<td>[-1, 4096]</td>
<td>0</td>
</tr>
<tr>
<td>Dropout-18</td>
<td>[-1, 4096]</td>
<td>0</td>
</tr>
<tr>
<td>Linear-19</td>
<td>[-1, 4096]</td>
<td>16,781,312</td>
</tr>
<tr>
<td>ReLU-20</td>
<td>[-1, 4096]</td>
<td>0</td>
</tr>
<tr>
<td>Linear-21</td>
<td>[-1, 101]</td>
<td>413,797</td>
</tr>
</tbody>
</table>

Total params: 50,883,321
Trainable params: 50,883,321
Non-trainable params: 0

Input size (MB): 0.57
Forward/backward pass size (MB): 7.27
Params size (MB): 194.10
Estimated Total Size (MB): 201.95

You also need to track model accuracy, and the following image shows how you can plot the model pruning process to visualize changes in model accuracy based on the number of parameters in SageMaker Studio.
In SageMaker Studio, choose the **Experiments** tab, select a list of tensors saved by Debugger from the pruning process, and then compose a **Trial Component List** panel. Select all ten iterations and then choose **Add chart** to create a **Trial Component Chart**. After you decide on a model to deploy, choose the trial component and choose a menu to perform an action or choose **Deploy model**.

**Note**
To deploy a model through SageMaker Studio using the following notebook example, add a line at the end of the train function in the `train.py` script.

```python
# In the train.py script, look for the train function in line 58.
def train(epochs, batch_size, learning_rate):
    ...
    print('acc:{:.4f}'.format(correct/total))
    hook.save_scalar("accuracy", correct/total, sm_metric=True)

    # Add the following code to line 128 of the train.py script to save the pruned models
    # under the current SageMaker Studio model directory
    torch.save(model.state_dict(), os.environ['SM_MODEL_DIR'] + '/model.pt')
```

884
Using SageMaker Debugger to Monitor a Convolutional Autoencoder Model Training

This notebook demonstrates how SageMaker Debugger visualizes tensors from an unsupervised (or self-supervised) learning process on a MNIST image dataset of handwritten numbers.

The training model in this notebook is a convolutional autoencoder with the MXNet framework. The convolutional autoencoder has a bottleneck-shaped convolutional neural network that consists of an encoder part and a decoder part.

The encoder in this example has two convolution layers to produce compressed representation (latent variables) of the input images. In this case, the encoder produces a latent variable of size (1, 20) from an original input image of size (28, 28) and significantly reduces the size of data for training by 40 times.

The decoder has two deconvolutional layers and ensures that the latent variables preserve key information by reconstructing output images.

The convolutional encoder powers clustering algorithms with smaller input data size and the performance of clustering algorithms such as k-means, k-NN, and t-Distributed Stochastic Neighbor Embedding (t-SNE).

This notebook example demonstrates how to visualize the latent variables using Debugger, as shown in the following animation. It also demonstrates how the t-SNE algorithm classifies the latent variables into ten clusters and projects them into a two-dimensional space. The scatter plot color scheme on the right side of the image reflects the true values to show how well the BERT model and t-SNE algorithm organize the latent variables into the clusters.

Using SageMaker Debugger to Monitor Attentions in BERT Model Training

Bidirectional Encode Representations from Transformers (BERT) is a language representation model. As the name of model reflects, the BERT model builds on transfer learning and the Transformer model for natural language processing (NLP).

The BERT model is pre-trained on unsupervised tasks such as predicting missing words in a sentence or predicting the next sentence that naturally follows a previous sentence. The training data contains 3.3 billion words (tokens) of English text, such as Wikipedia and electronic books. For a simple example, the BERT model can give a high attention to appropriate verb tokens or pronoun tokens from a subject token.
The pre-trained BERT model can be fine-tuned with an additional output layer to achieve state-of-the-art model training in NLP tasks, such as automated response to questions, text classification, and many others.

Debugger collects tensors from the fine-tuning process. In the context of NLP, the weight of neurons is called **attention**.

This notebook demonstrates how to use the pre-trained BERT model from the GluonNLP model zoo on the Stanford Question and Answering dataset and how to set up SageMaker Debugger to monitor the training job.

Plotting **attention scores** and individual neurons in the query and key vectors can help to identify causes of incorrect model predictions. With SageMaker Debugger, you can retrieve the tensors and plot the **attention-head view** in real time as training progresses and understand what the model is learning.

The following animation shows the attention scores of the first 20 input tokens for ten iterations in the training job provided in the notebook example.
Using SageMaker Debugger to Visualize Class Activation Maps in Convolutional Neural Networks (CNNs)

This notebook demonstrates how to use SageMaker Debugger to plot class activation maps for image detection and classification in convolutional neural networks (CNNs). In deep learning, a convolutional neural network (CNN or ConvNet) is a class of deep neural networks, most commonly applied to analyzing visual imagery. One of the applications that adopts the class activation maps is self-driving cars, which require instantaneous detection and classification of images such as traffic signs, roads, and obstacles.

In this notebook, the PyTorch ResNet model is trained on the German Traffic Sign Dataset, which contains more than 40 classes of traffic-related objects and more than 50,000 images in total.

![Input image](image1.jpg) ![Predicted class 42 with probability 5%](image2.jpg)

During the training process, SageMaker Debugger collects tensors to plot the class activation maps in real time. As shown in the animated image, the class activation map (also called as a saliency map) highlights regions with high activation in red color.

Using tensors captured by Debugger, you can visualize how the activation map evolves during the model training. The model starts by detecting the edge on the lower-left corner at the beginning of the training job. As the training progresses, the focus shifts to the center and detects the speed limit sign, and the model successfully predicts the input image as Class 3, which is a class of speed limit 60km/h signs, with a 97% confidence level.

Debugger on Studio

Use the Debugger dashboards on Studio to analyze your model performance on Amazon EC2 instances. For any SageMaker training jobs running on Amazon EC2 instance, by default, Debugger monitors system metrics (CPU, GPU, CPU and GPU memory, Network, and data I/O) every 500 milliseconds and basic output tensors (loss and accuracy) every 500 iterations. You can also further customize Debugger configuration parameter values and adjust the saving intervals through the Studio UI or using the Amazon SageMaker Python SDK. Through the Studio Debugger dashboards, gain insights into your training jobs and improve your model training performance and accuracy.

**Important**

If using existing SageMaker Studio apps, you need to restart them to use the new features. To restart and update your Studio environment, follow the instructions at Update Amazon SageMaker Studio.
To open the Debugger dashboards

1. Choose SageMaker Components and registries, open the drop down menu, and choose Experiments and trials.

2. Look up your training job name to open the Debugger dashboards. If you have not specified a SageMaker Experiments object, your training job trial is collected under the Unassigned trial components list.

3. Open the drop down menu of the training job. There are two menu items to browse the Debugger dashboards: Open Debugger for insights and Open in trial details.
• **Open Debugger for insights**

  Opens a **Debug [your-training-job-name]** tab. In this tab, Debugger provides an overview of your model performance on EC2 instances and identifies system bottleneck problems. While **monitoring** your model performance, you can also enable **profiling** to capture framework metrics, which consist of data from neural network operations executed during forward and backward pass and data loading. Debugger correlates the system resource utilization metrics with the framework metrics, and helps identify resource-intensive operators that might be the root cause of the system bottlenecks. You can also download an aggregated Debugger profiling report. For more information, see [Debugger for Insights on Studio](#) (p. 890).

• **Open in trial details**

  Opens a **Describe Trial Component** tab. You can check **Charts** and **Metrics** of the model output data captured by Debugger and logged by a SageMaker Experiments Tracker. Under the **Debugger** tab, you can check the status of Debugger rules in real time. For more information, see [Debugger in Studio Experiments](#) (p. 900).

See the following topics to learn more about **Debugger Insights on Studio** and **Debugger in Studio Experiment List Trial**.

**Topics**

- Debugger for Insights on Studio (p. 890)
- Debugger in Studio Experiments (p. 900)

**Debugger for Insights on Studio**

When you initiate a SageMaker training job, Debugger starts monitoring hardware system resource utilization of EC2 instances on which your training job is running. You can track the system utilization status, statistics overview, and bottleneck issues through the Studio

**Note**

Studio Debugger insights dashboard runs a Studio kernel on an `ml.m5.4xlarge` instance to process and render the visualizations. When you close the Debugger insights tab, the corresponding kernel session will also be closed. Each Debugger insights tab runs one Studio kernel session. For information about pricing, see the Amazon SageMaker Pricing page.

**Open Debugger for Insights on Studio**

This guide walks you through every component of the Studio Debugger insights dashboard. The following animated screenshot shows a quick overview of the Studio Debugger insights dashboard that you can see after choosing a training job trial in **Experiment List** and **Open Debugger for insights**. On the **Debug [your-training-job-name]** page, Debugger autogenerates model training performance summaries and insights on the **Overview** tab, and charts on the **Nodes** tab in real time.
**Note**
It might take a few minutes to load the Debugger insights dashboard if you are using it for the first time.

**Tips**
- If you want to manually refresh the **Debug [your-training-job-name]** page, choose the refresh button (the round arrow at the upper-left corner) as shown in the following screenshot.
- To download a comprehensive Debugger profiling report with more details and analysis, choose **Download report**. For more information about the Debugger profiling report, see SageMaker Debugger Profiling Report (p. 903).

**Enable Debugger Profiling for Detailed Insights**

When you enable **Profiling**, Debugger starts collecting framework metrics. Framework metrics are model data collected from ML framework operations of your model, such as forward pass, backward pass, batch normalization, and data loader processes. Debugger correlates system performance bottlenecks with the framework operations, and helps improve the performance of your model with a fleet of Debugger built-in rules.

**Note**
After you’ve enabled profiling, Debugger collects every framework operation call that’s executed in each step: operations for convolving down input layers during forward pass, updating weights of millions of neurons during backward pass, and data loader processes. While profiling enables you to understand model performance at a deeper level, collecting framework metrics might
impact your training time and performance. We recommend that you enable profiling to inspect your model up to two steps at a time. For more information about how to configure Debugger for framework profiling using Amazon SageMaker Python SDK, see Configure Debugger Framework Profiling (p. 930) and Updating Debugger System Monitoring and Framework Profiling Configuration while a Training Job is Running (p. 934).

If you want to enable profiling while your training job is running, use the following steps to start profiling.

1. In SageMaker Studio, turn on Profiling to enable Debugger framework profiling.

2. In Configure Debugger monitoring and profiling, S3 bucket URI and Collect monitoring data every are already set to the default values.

You can choose to change the monitoring interval using the drop down menu and selecting among the following available options: 100 milliseconds, 200 milliseconds, 500 milliseconds (default), 1 second, 5 seconds, and 1 minute.

- S3 bucket URI – Specify the base S3 bucket URI.
- Collect monitoring data every – Select a time interval to collect system metrics.

**Note**
If you choose one of the lower time intervals, you increase the granularity of monitoring system metrics. It allows you to capture spikes and anomalies with a higher time
resolution. However, as the size of system metrics to process proportionally increases, it might impact the overall training and processing time.

3. In **Advanced settings for profiling**, configure framework metrics profiling options. Specify **Start step** (or **Start time**) and **Number of steps to profile** (or **Time duration to profile**) to profile. You can also leave the input fields blank. The default values will be automatically configured to use the current step and for 1 step duration.

- **Detailed profiling** – Specify a target step or time range to profile framework operations using the native framework profilers (TensorFlow profiler and PyTorch profiler). For example, if using TensorFlow, the Debugger hooks enable the TensorFlow profiler to collect TensorFlow-specific framework metrics. Detailed profiling enables you to profile all framework operators at a pre-step (before the first step), within steps, and between steps of a training job.

  **Note**
  
  The detailed profiling might significantly increase GPU memory consumption. It is not recommended to enable the detailed profiling for more than a couple of steps.

- **Python profiling** – Specify a target step or time range to profile Python functions. You can also choose between two Python profilers: cProfile and Pyinstrument.

  - **cProfile** – The standard Python profiler. cProfile collects for every Python operator called during training. With cProfile, Debugger saves cumulative time and annotation of each function call, providing a complete detail of Python functions. In deep learning, for example, the most frequently called functions might be the convolutional filters and backward pass operators, and cProfile profiles every single of them. For the cProfile option, you can further select a timer option: total time, CPU time, and off-CPU time. While you can profile every function calls executing on processors (both CPU and GPU) in CPU time, you can also identify I/O or network bottlenecks with the off-CPU time option. The default is total time, and Debugger profiles both CPU and off-CPU time. With cProfile, you are able to drill down to every single functions when analyzing the profile data.

  - **Pyinstrument** – Pyinstrument is a low overhead Python profiler that works based on sampling. With the Pyinstrument option, Debugger samples profiling events every millisecond. Because Pyinstrument measures elapsed wallclock time instead of CPU time, the Pyinstrument option can be a better choice over the cProfile option for reducing profiling noises (filtering out irrelevant function calls that are cumulatively fast) and capturing operators that are actually compute intensive (cumulatively slow) for training your model. With Pyinstrument, you are able to see a tree of function calls and better understand the structure and root cause of the slowness.

  **Note**
  
  Enabling Python profiling might result in slowing down the overall training time. In case of cProfile, Python operators that are the most frequently called are profiled at every
call, so the processing time on profiling increases with respect to the number of calls. For Pyinstrument, the cumulative profiling time increases with respect to time because of its sampling mechanism.

- **Dataloader profiling** – Specify a target step or time range to profile deep learning framework data loader processes. Debugger collects every data loader event of the frameworks.

**Note**
The data loader profiling can lower the training performance while collecting information from data loaders. We don't recommend that you enable the data loader profiling for more than a couple of steps. Debugger is pre-configured to annotate data loader processes only for the AWS deep learning containers. Debugger cannot profile data loader processes on any other custom or external containers.

4. Choose **Confirm** to finish your profiling configuration. When the configuration is successfully updated, you should be able to see the following confirmation message.

![Debugger configuration updated successfully](image)

---

**Debugger Insights – Overview**

On the **Overview** tab, you can find a training job summary, resource utilization summary, resource intensive operations, and insights.

**Training job summary**

The **Training job summary** section shows the overall training time spent on different phases of training: initialization, training loop, and finalization. The pie chart shows the time usage percentage and absolute time amount spent on the different training phases. For example, you can have a high-level overview of how long it takes for initializing a training job, check if the initialization is taking too long due to data downloading, leaving the GPUs idle.
This section has the following features:

- The **Training progress over time** chart shows the timeline of the different training phase over time. If using spot training, you can also find the spot interruptions in the timeline chart.

- The **Training job details** panel shows the exact time stamps and utilization rate percentage numbers.
  - **Start time** – The exact time when the training job started.
  - **End time** – The exact time when the training job finished.
  - **Job duration** – The total training time from the **Start time** to the **End time**.
  - **Training loop start** – The exact time when the first step of the first epoch has started.
  - **Training loop end** – The exact time when the last step of the last epoch has finished.
  - **Training loop duration** – The total time between the Training loop start time and the Training loop end time.

- **Initialization** – Time spent on initializing the training job, such as compiling the training script, initiating EC2 instances, and downloading training data.

- **Finalization** – Time spent on finalizing the training job, such as finishing the model training, updating the model artifacts, and closing the EC2 instances.

- **Initialization (%)** – The percentage of time spent on **Initialization** over the total **Job duration**.
- **Training loop (%)** – The percentage of time spent on **Training loop** over the total **Job duration**.
- **Finalization (%)** – The percentage of time spent on **Finalization** over the total **Job duration**.

### Resource utilization summary

This summary table shows hardware system resource utilization statistics of all workers (algo-n). System metrics include total CPU utilization, total GPU utilization, total CPU memory utilization, total GPU memory utilization, total I/O wait time, and total Network in bytes. The table shows the minimum and the maximum values, and p99, p90, and p50 percentiles.

<table>
<thead>
<tr>
<th>Node</th>
<th>Metric</th>
<th>Unit</th>
<th>Min</th>
<th>p99</th>
<th>p90</th>
<th>p50</th>
<th>p50</th>
<th>Min</th>
</tr>
</thead>
<tbody>
<tr>
<td>algo-1</td>
<td>Network</td>
<td>243994671.5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>algo-2</td>
<td>Network</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>algo-1</td>
<td>GPU</td>
<td>75</td>
<td>74.25</td>
<td>68.75</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>algo-2</td>
<td>GPU</td>
<td>75</td>
<td>74.25</td>
<td>68</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>algo-1</td>
<td>CPU</td>
<td>34.82</td>
<td>22.78</td>
<td>15.11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>algo-2</td>
<td>CPU</td>
<td>51.6</td>
<td>22.97</td>
<td>15.45</td>
<td>0.6</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>algo-1</td>
<td>CPU memory</td>
<td>5.04</td>
<td>5.05</td>
<td>4.92</td>
<td>0.87</td>
<td>0.97</td>
<td>0.55</td>
<td>0</td>
</tr>
<tr>
<td>algo-2</td>
<td>CPU memory</td>
<td>5</td>
<td>5</td>
<td>4.95</td>
<td>0.87</td>
<td>0.97</td>
<td>0.55</td>
<td>0</td>
</tr>
<tr>
<td>algo-1</td>
<td>GPU memory</td>
<td>36.75</td>
<td>24.25</td>
<td>13.25</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>algo-2</td>
<td>GPU memory</td>
<td>31.75</td>
<td>22.75</td>
<td>14.5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>algo-1</td>
<td>I/O</td>
<td>22.16</td>
<td>6.25</td>
<td>0.57</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>algo-2</td>
<td>I/O</td>
<td>21.88</td>
<td>6.25</td>
<td>0.62</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

### Resource intensive operations

The **Resource intensive operations** section provides more detailed profiling results that show what operations of the training job were compute intensive. In the following example, it shows that the convolutional neural network backward pass operators were the most resource intensive on the GPUs.
Insights

In the **Insights** pane, you can find training issues detected by Debugger built-in rules. You can expand each of the list to find useful insights, suggestions, a description of the rule, and criteria of triggering the rule.

For more information about the Debugger built-in rules, see [List of Debugger Built-in Rules](#).
The first two graphs show CPU utilization and network utilization over time. By default, the graphs show the mean values: the average of CPU and network utilization over the total number of CPU cores. You can select one or more CPU cores, by selecting on the labels, to graph them on single chart and compare utilization across cores. The timeline graphs are interactive, and the two graphs are sync up. You can drag and zoom in and out to see specific time windows that you want to have a closer look.

GPU and GPU memory utilization

The following graphs show GPU utilization and GPU memory utilization over time. By default, the graphs show the mean utilization rate over time. You can select the GPU core labels to see the utilization rate of each core. Taking the mean of utilization rate over the total number of GPU cores shows the mean utilization of the entire hardware system resource. By looking at the mean utilization rate, you can check the overall system resource usage of EC2 instance. The following figure shows an example training job on an ml.p3.16xlarge instance with 8 GPU cores. You can monitor if the training job is well distributed, fully utilizing all GPUs.
Overall system utilization over time

The following heatmap shows the entire system utilization over time projected onto the two-dimensional plot. Every CPU and GPU cores are listed in the vertical axis, and the utilization is recorded over time with colors. See the labeled colorbar on the right side of the plot to find out what color level corresponds to what utilization rate. For example, in the following heatmap, after initialization phase has ended around Sun 23:18, you can find that the training job fully utilizes an ml.p3.16xlarge instance: the GPU cores are fully utilized and the CPUs are moderately used for processing Python operations. There were several CPU bottleneck problems scattered across the CPUs at different times.
System resource utilization over time and framework event phase

The System metrics over time graph shows the overall CPU, GPU, and data I/O utilization. The Framework metrics over time graph shows the framework metrics, which are the framework event phases that you can correlate with the System metrics over time graph.

You can select a time interval of interest in the system resource usage timeline, and the framework event phase spots the interval to show what events happened during the selected time interval. In each event phase block, you can find what time interval was actually spent for training loop, and break the training loop into backward pass and forward pass events. Overall, you can see that actual training time intervals are occupying only a small percentage over the entire training time.
Time spent in training

In the following graph, framework metrics from the last 30 steps of the last training loop are captured. This graph shows cumulative time spent by different events in each step.

Debugger in Studio Experiments

In this section, you learn how to use the Debugger in Studio Experiments. You can select any training jobs from the Experiment trial list to see the model output data graphs, such as accuracy and loss curves, debugging built-in rule status, and Debugger configuration information for debugging.
Visualize Tensors Using SageMaker Debugger and Studio

SageMaker Studio provides visualizations to interpret tensor outputs that are captured by Debugger.

Loss Curves While Training Is in Progress

The following screenshot shows visualizations of loss curves for training. The training is in progress.

Analyzing Training Jobs: Comparing Loss Curves Across Multiple Jobs

SageMaker Studio enables you to compare across multiple jobs (in this case, the loss). This helps you identify the best-performing training jobs.
Rules Triggering and Logs from Jobs

When rules are triggered for anomalous conditions, SageMaker Studio presents logs for the failing rule, which enables you to analyze the causes of the condition.

SageMaker Debugger Interactive Reports

Receive training and profiling reports autogenerated by Debugger. The Debugger reports provide insights into your training jobs and suggest recommendations to improve your model performance. The following screenshot shows a collage of the Debugger profiling report. To learn more, see SageMaker Debugger Profiling Report (p. 903).

**Note**
You can download a Debugger report while your training job is running or after the job has finished. During training, Debugger concurrently updates the report reflecting the current rules' evaluation status. You can download a complete Debugger report only after the training job has completed.

**Important**
To use the new Debugger features, you need to upgrade the SageMaker Python SDK and the SMDebug client library. In your iPython kernel, Jupyter notebook, or JupyterLab environment, run the following code to install the latest versions of the libraries and restart the kernel.

```python
import sys
import IPython
!{sys.executable} -m pip install -U sagemaker smdebug
IPython.Application.instance().kernel.do_shutdown(True)
```

**Topics**

- SageMaker Debugger Profiling Report (p. 903)
- SageMaker Debugger XGBoost Training Report (p. 913)
For any SageMaker training jobs, the Debugger ProfilerReport (p. 951) rule invokes all of the monitoring and profiling rules (p. 949) and aggregates the rule analysis into a comprehensive report. Following this guide, download the report using the Amazon SageMaker Python SDK or the S3 console, and learn what you can interpret from the profiling results.

**Important**
To use the new Debugger features, you need to upgrade the SageMaker Python SDK and the SMDebug client library. In your iPython kernel, Jupyter notebook, or JupyterLab environment, run the following code to install the latest versions of the libraries and restart the kernel.

```python
import sys
import IPython
!{sys.executable} -m pip install -U sagemaker smdebug
IPython.Application.instance().kernel.do_shutdown(True)
```

**Topics**
- Download a Debugger Profiling Report (p. 903)
- Debugger Profiling Report Walkthrough (p. 906)

**Download a Debugger Profiling Report**

Download the Debugger profiling report while your training job is running or after the job has finished using the Amazon SageMaker Python SDK and AWS Command Line Interface (CLI).

**Tip**
You can also download the report with one click and no additional scripting through the SageMaker Studio Debugger insights dashboard. To find out how to download the report from Studio, see Debugger for Insights on Studio (p. 890).
Download using SageMaker Python SDK and AWS CLI

1. Check the current job's default S3 output base URI.
   ```python
   estimator.output_path
   ```

2. Check the current job name.
   ```python
   estimator.latest_training_job.job_name
   ```

3. The Debugger profiling report is stored under `<default-s3-output-base-uri>/
   <training-job-name>/rule-output`. Configure the rule output path as follows:
   ```python
   rule_output_path = estimator.output_path + estimator.latest_training_job.job_name + 
   "rule-output"
   ```

4. To check if the report is generated, list directories and files recursively under the
   `rule_output_path` using `aws s3 ls` with the `--recursive` option.
   ```
   ! aws s3 ls {rule_output_path} --recursive
   ```

   This should return a complete list of files under an autogenerated folder that's named
   as `ProfilerReport-1234567890`. The folder name is a combination of strings:
   `ProfilerReport` and a unique 10-digit tag based on the Unix timestamp when the
   `ProfilerReport` rule is initiated.

   The `profiler-report.html` is an autogenerated profiling report by Debugger. The remaining
   files are the built-in rule analysis components stored in JSON and a Jupyter notebook that are
   used to aggregate them into the report.

5. Download the files recursively using `aws s3 cp`. The following command saves all of the rule
   output files to the `ProfilerReport-1234567890` folder under the current working directory.
   ```
   ! aws s3 cp {rule_output_path} ./ --recursive
   ```

   **Tip**
   If using a Jupyter notebook server, run `!pwd` to double check the current working
   directory.

6. Under the `/ProfilerReport-1234567890/profiler-output` directory, open `profiler-report.html`. If using JupyterLab, choose `Trust HTML` to see the autogenerated Debugger
   profiling report.

7. Open the `profiler-report.ipynb` file to explore how the report is generated. You can also
   customize and extend the profiling report using the Jupyter notebook file.
Download using Amazon S3 Console

1. Sign in to the AWS Management Console and open the Amazon S3 console at https://console.aws.amazon.com/s3/.

2. Search for the base S3 bucket. For example, if you haven't specified any base job name, the base S3 bucket name should be in the following format: sagemaker-<region>-111122223333. Look up the base S3 bucket through the Find bucket by name field.

3. In the base S3 bucket, look up the training job name by specifying your job name prefix into the Find objects by prefix input field. Choose the training job name.

4. In the training job's S3 bucket, there must be three subfolders for training data collected by Debugger: debug-output/, profiler-output/, and rule-output/. Choose rule-output/.

5. In the rule-output/ folder, choose ProfilerReport-1234567890, and choose profiler-output/ folder. The profiler-output/ folder contains profiler-report.html (the autogenerated profiling report in html), profiler-report.ipynb (a Jupyter notebook with scripts that are used for generating the report), and a profiler-report/ folder (contains rule analysis JSON files that are used as components of the report).

6. Select the profiler-report.html file, choose Actions, and Download.
Open the downloaded `profiler-report.html` file in a web browser.

**Note**
If you started your training job without configuring the Debugger-specific parameters, Debugger generates the report based only on the system monitoring rules because the Debugger parameters are not configured to save framework metrics. To enable framework metrics profiling and receive an extended Debugger profiling report, configure the `profiler_config` parameter when constructing or updating SageMaker estimators.
To learn how to configure the `profiler_config` parameter before starting a training job, see Configure Debugger Framework Profiling (p. 930).
To update the current training job and enable framework metrics profiling, see Update Debugger Framework Profiling Configuration (p. 934).

**Debugger Profiling Report Walkthrough**

This section walks you through the Debugger profiling report section by section. The profiling report is generated based on the built-in rules for monitoring and profiling. The report shows result plots only for the rules that found issues.

**Topics**
- Training Job Summary (p. 907)
- System Usage Statistics (p. 908)
- Framework metrics summary (p. 908)
- Rules Summary (p. 910)
- Analyzing the Training Loop – Step Durations (p. 911)
- GPU Utilization Analysis (p. 911)
Training Job Summary

At the beginning of the report, Debugger provides a summary of your training job. In this section, you can overview the time durations and timestamps at different training phases.

**Training job summary**

The following table gives a summary about the training job. The table includes information about when the training job started and ended, how much time initialization, training loop and finalization took. Your training job started on 11/06/2020 at 23:12:42 and ran for 707 seconds.

<table>
<thead>
<tr>
<th>#</th>
<th>Job Statistics</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Start time 23:12:42 11/06/2020</td>
</tr>
<tr>
<td>1</td>
<td>End time 23:24:59 11/06/2020</td>
</tr>
<tr>
<td>2</td>
<td>Job duration 737 seconds</td>
</tr>
<tr>
<td>3</td>
<td>Training/loop start 23:17:31 11/06/2020</td>
</tr>
<tr>
<td>4</td>
<td>Training/loop end 23:44:59 11/06/2020</td>
</tr>
<tr>
<td>5</td>
<td>Training/loop duration 448 seconds</td>
</tr>
<tr>
<td>6</td>
<td>Initialization time 288 seconds</td>
</tr>
<tr>
<td>7</td>
<td>Finalization time 9 seconds</td>
</tr>
<tr>
<td>8</td>
<td>Initialization 31 %</td>
</tr>
<tr>
<td>9</td>
<td>Training/loop 80 %</td>
</tr>
<tr>
<td>10</td>
<td>Finalization 9 %</td>
</tr>
</tbody>
</table>

The summary table contains the following information:

- **start_time** – The exact time when the training job started.
- **end_time** – The exact time when the training job finished.
- **job_duration_in_seconds** – The total training time from the start_time to the end_time.
- **training_loop_start** – The exact time when the first step of the first epoch has started.
- **training_loop_end** – The exact time when the last step of the last epoch has finished.
- **training_loop_duration_in_seconds** – The total time between the training loop start time and the training loop end time.
- **initialization_in_seconds** – Time spent on initializing the training job. The initialization phase covers the period from the start_time to the training_loop_start time. The initialization time is spent on compiling the training script, starting the training script, creating and initializing the model, initiating EC2 instances, and downloading training data.
- **finalization_in_seconds** – Time spent on finalizing the training job, such as finishing the model training, updating the model artifacts, and closing the EC2 instances. The finalization phase covers the period from the training_loop_end time to the end_time.
- **initialization (%)** – The percentage of time spent on initialization over the total job_duration_in_seconds.
- **training loop (%)** – The percentage of time spent on training_loop over the total job_duration_in_seconds.
- **finalization (%)** – The percentage of time spent on finalization over the total job_duration_in_seconds.
System Usage Statistics

In this section, you can see an overview of system utilization statistics.

System usage statistics

The 95th quantile of the total GPU utilization on node algo-2 is 74%. GPUs on node algo-2 are well utilized.

The following table shows usage statistics per worker node such as total CPU and GPU utilization, total CPU and memory footprint. The table also include total IO wait time and total sent/received bytes. The table shows min and max values as well as p99, p95 and p50 percentiles.

<table>
<thead>
<tr>
<th>#</th>
<th>node</th>
<th>metric</th>
<th>unit</th>
<th>max</th>
<th>p99</th>
<th>p95</th>
<th>p50</th>
<th>min</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>algo-1</td>
<td>Network</td>
<td>bytes</td>
<td>218817581.57</td>
<td>168.02</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>algo-1</td>
<td>I/O</td>
<td>percentage</td>
<td>13.2953125</td>
<td>5.59283125</td>
<td>0.0000000000</td>
<td>0.1955937499999</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>algo-1</td>
<td>GPU memory</td>
<td>percentage</td>
<td>32.25</td>
<td>29.25</td>
<td>21</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>algo-1</td>
<td>GPU</td>
<td>percentage</td>
<td>75</td>
<td>74.5</td>
<td>74.25</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>algo-1</td>
<td>CPU memory</td>
<td>percentage</td>
<td>5.05</td>
<td>5.01</td>
<td>4.98</td>
<td>2.17</td>
<td>0.55</td>
</tr>
<tr>
<td>4</td>
<td>algo-1</td>
<td>CPU</td>
<td>percentage</td>
<td>32.959825</td>
<td>22.62913125</td>
<td>17.034</td>
<td>3.7624999999999</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>algo-2</td>
<td>Network</td>
<td>bytes</td>
<td>4135.24</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>algo-2</td>
<td>I/O</td>
<td>percentage</td>
<td>20.1975</td>
<td>8.155250000000</td>
<td>1.7478124999999</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>algo-2</td>
<td>GPU memory</td>
<td>percentage</td>
<td>38</td>
<td>31.75</td>
<td>21.75</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>algo-2</td>
<td>GPU</td>
<td>percentage</td>
<td>75</td>
<td>74.5</td>
<td>74.25</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>algo-2</td>
<td>CPU memory</td>
<td>percentage</td>
<td>5.05</td>
<td>5.02</td>
<td>4.99</td>
<td>2.17</td>
<td>0.55</td>
</tr>
<tr>
<td>5</td>
<td>algo-2</td>
<td>CPU</td>
<td>percentage</td>
<td>35.004374999999999</td>
<td>25.699988750000</td>
<td>18.33406875</td>
<td>3.77828125</td>
<td>0</td>
</tr>
</tbody>
</table>

The Debugger profiling report includes the following information:

- **node** – Lists the name of nodes. If using distributed training on multi nodes (multiple EC2 instances), the node names are in format of algo-n.
- **metric** – The system metrics collected by Debugger: CPU, GPU, CPU memory, GPU memory, I/O, and Network metrics.
- **unit** – The unit of the system metrics.
- **max** – The maximum value of each system metric.
- **p99** – The 99th percentile of each system utilization.
- **p95** – The 95th percentile of each system utilization.
- **p50** – The 50th percentile (median) of each system utilization.
- **min** – The maximum value of each system metric.

Framework metrics summary

In this section, the following pie charts show the breakdown of framework operations on CPUs and GPUs.
Each of the pie charts analyzes the collected framework metrics in various aspects as follows:

- **Ratio between TRAIN/EVAL phase and others** – Shows the ratio between time durations spent on different training phases.

- **Ratio between forward and backward pass** – Shows the ratio between time durations spent on forward and backward pass in the training loop.

- **Ratio between CPU/GPU operators** – Shows the ratio between time spent on operators running on CPU or GPU, such as convolutional operators.

- **General metrics recorded in framework** – Shows the ratio between time spent on major framework metrics, such as data loading, forward and backward pass.

**Overview: CPU Operators**

This section provides information of the CPU operators in detail. The table shows the percentage of the time and the absolute cumulative time spent on the most frequently called CPU operators.
Overview: GPU Operators

This section provides information of the GPU operators in detail. The table shows the percentage of the time and the absolute cumulative time spent on the most frequently called GPU operators.

Rules Summary

In this section, Debugger aggregates all of the rule evaluation results, analysis, rule descriptions, and suggestions.

Rules summary

<table>
<thead>
<tr>
<th>Description</th>
<th>Recommendation</th>
<th>Number of times rule triggered</th>
<th>Number of data points</th>
<th>Rule parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load Balancer</td>
<td>Request 0.2 or less in the load balancing between multiple GPUs. Workloads related to can be improved by machine learning to handle training when gradients are accumulated on primary GPU or secondary GPU.</td>
<td>263</td>
<td>5407</td>
<td>Increase 0.2</td>
</tr>
<tr>
<td>Cost/Benefit Analysis</td>
<td>Check the GPU Utilization is low or suffers from fluctuations. If this happens, the CPU or GPU workload should be optimized.</td>
<td>264</td>
<td>5408</td>
<td>Increase 0.2</td>
</tr>
<tr>
<td>Batch Size</td>
<td>Check if GPU is under-utilized because of the batch size being too low. If so, adjust the batch size to increase the overall performance.</td>
<td>265</td>
<td>5409</td>
<td>Increase 0.2</td>
</tr>
<tr>
<td>GPU Memory Usage</td>
<td>If used, the GPU usage should also be increased to larger than the training run-out of memory and storage.</td>
<td>266</td>
<td>5410</td>
<td>Increase 0.2</td>
</tr>
<tr>
<td>CPU Bottleneck</td>
<td>Check if CPU usage is high but the GPU usage is low at the same time. If so, it indicates a CPU bottleneck where GPU is waiting for data to be sent from CPU.</td>
<td>267</td>
<td>5411</td>
<td>Increase 0.2</td>
</tr>
<tr>
<td>I/O Bottleneck</td>
<td>Check if I/O wait time is high but at the same time CPU usage is low, it may indicate an I/O bottleneck where CPU is waiting for data to arrive from disk.</td>
<td>268</td>
<td>5412</td>
<td>Increase 0.2</td>
</tr>
<tr>
<td>Slow Output</td>
<td>Check if the slow output is stop on the training.</td>
<td>269</td>
<td>5413</td>
<td>Increase 0.2</td>
</tr>
<tr>
<td>Warmup/Initiation Time</td>
<td>Check if the training initiation is taking too much time.</td>
<td>270</td>
<td>5414</td>
<td>Increase 0.2</td>
</tr>
</tbody>
</table>
Analyzing the Training Loop – Step Durations

In this section, you can find a detailed statistics of step durations on each GPU core of each node. Debugger evaluates mean, maximum, p99, p95, p50, and minimum values of step durations, and evaluate step outliers. The following histogram shows the step durations captured on different worker nodes and GPUs. You can enable or disable the histogram of each worker by choosing the legends on the right side. You can check if there is a particular GPU that's causing step duration outliers.

GPU Utilization Analysis

This section shows the detailed statistics about GPU core utilization based on LowGPUtilization rule. It also summarizes the GPU utilization statistics, mean, p95, and p5 to determine if the training job is underutilizing GPUs.

Batch Size

This section shows the detailed statistics of total CPU utilization, individual GPU utilizations, and GPU memory footprints. The BatchSize rule determines if you need to change the batch size to better utilize the GPUs. You can check whether the batch size is too small resulting in underutilization or too large causing overutilization and out of memory issues. In the plot, the boxes show the p25 and p75 percentile ranges (filled with dark purple and bright yellow respectively) from the median (p50), and the error bars show the 5th percentile for the lower bound and 95th percentile for the upper bound.
CPU Bottlenecks

In this section, you can drill down into the CPU bottlenecks that the CPUBottleneck rule detected from your training job. The rule checks if the CPU utilization is above `cpu_threshold` (90% by default) and also if the GPU utilization is below `gpu_threshold` (10% by default).

The pie charts show the following information:

- **Low GPU usage caused by CPU bottlenecks** – Shows the ratio of data points between the ones with GPU utilization above and below the threshold and the ones that matches the CPU bottleneck criteria.
- **Ratio between TRAIN/EVAL phase and others** – Shows the ratio between time durations spent on different training phases.
- **Ratio between forward and backward pass** – Shows the ratio between time durations spent on forward and backward pass in the training loop.
• **Ratio between CPU/GPU operators** – Shows the ratio between time durations spent on GPUs and CPUs by Python operators, such as data loader processes and forward and backward pass operators.

• **General metrics recorded in framework** – Shows major framework metrics and the ratio between time durations spent on the metrics.

**I/O Bottlenecks**

In this section, you can find a summary of IO bottlenecks.

**LoadBalancing in Multi-GPU Training**

In this section, you can identify workload balancing issue across GPUs.

**GPU Memory Analysis**

In this section, you can analyze the GPU memory utilization collected by the GPUMemoryIncrease rule. In the plot, the boxes show the p25 and p75 percentile ranges (filled with dark purple and bright yellow respectively) from the median (p50), and the error bars show the 5th percentile for the lower bound and 95th percentile for the upper bound.

**SageMaker Debugger XGBoost Training Report**

For SageMaker XGBoost training jobs, use the Debugger CreateXgboostReport (p. 961) rule to receive a comprehensive training report of the training progress and results. Following this guide, specify the CreateXgboostReport (p. 961) rule while constructing an XGBoost estimator, download the report using
the Amazon SageMaker Python SDK or the Amazon S3 console, and then you can interpret the profiling results.

**Important**
To use the new Debugger features, you need to upgrade the SageMaker Python SDK and the SMDebug client library. In your iPython kernel, Jupyter notebook, or JupyterLab environment, run the following code to install the latest versions of the libraries and restart the kernel.

```python
import sys
import IPython
!{sys.executable} -m pip install -U sagemaker smdebug
IPython.Application.instance().kernel.do_shutdown(True)
```

**Topics**
- Construct a SageMaker XGBoost Estimator with the Debugger XGBoost Report Rule (p. 914)
- Download the Debugger XGBoost Training Report (p. 915)
- Debugger XGBoost Training Report Walkthrough (p. 917)

**Construct a SageMaker XGBoost Estimator with the Debugger XGBoost Report Rule**
When you construct a SageMaker estimator for an XGBoost training job, specify the rule as shown in the following example code.

The `CreateXgboostReport (p. 961)` rule collects the following output tensors from your training job:

- **hyperparameters** – Saves at the first step.
- **metrics** – Saves loss and accuracy every 5 steps.
- **feature_importance** – Saves every 5 steps.
- **predictions** – Saves every 5 steps.
- **labels** – Saves every 5 steps.

The output tensors are saved at a default S3 bucket. For example, `s3://sagemaker-<region>-<12digit_account_id>/<base-job-name>/debug-output/`

Using the SageMaker generic estimator

```python
import boto3
import sagemaker
from sagemaker.estimator import Estimator
from sagemaker import image_uris
from sagemaker.debugger import Rule, rule_configs

rules=[
    Rule.sagemaker(rule_configs.create_xgboost_report())
]

ger = boto3.Session().region_name
xgboost_container=sagemaker.image_uris.retrieve("xgboost", region, "1.2-1")

estimator=Estimator(
    role=sagemaker.get_execution_role(),
    image_uri=xgboost_container,
    base_job_name="debugger-xgboost-report-demo",
    instance_count=1,
    instance_type="ml.m5.2xlarge",
    s3_output_path="s3://sagemaker-<region>-<12digit_account_id>/<base-job-name>/debug-output/",
    output_config=rule_configs.create_xgboost_report(),
    rules=rules)
```

914
# Add the Debugger XGBoost report rule

```python
rules=rules
)
estimator.fit(wait=False)
```

## Download the Debugger XGBoost Training Report

Download the Debugger XGBoost training report while your training job is running or after the job has finished using the Amazon SageMaker Python SDK and AWS Command Line Interface (CLI).

### Download using the SageMaker Python SDK and AWS CLI

1. Check the current job’s default S3 output base URI.

   ```python
   estimator.output_path
   ```

2. Check the current job name.

   ```python
   estimator.latest_training_job.job_name
   ```

3. The Debugger XGBoost report is stored under `<default-s3-output-base-uri>/ <training-job-name>/rule-output`. Configure the rule output path as follows:

   ```python
   rule_output_path = estimator.output_path + estimator.latest_training_job.job_name + 
   "/*rule-output"
   ```

4. To check if the report is generated, list directories and files recursively under the `rule_output_path` using `aws s3 ls` with the `--recursive` option.

   ```sh
   ! aws s3 ls {rule_output_path} --recursive
   ```

This should return a complete list of files under autogenerated folders that are named `CreateXgboostReport` and `ProfilerReport-1234567890`. The XGBoost training report is stored in the `CreateXgboostReport`, and the profiling report is stored in the `ProfilerReport-1234567890` folder. To learn more about the profiling report generated by default with the XGBoost training job, see SageMaker Debugger Profiling Report (p. 903).

5. Download the files recursively using `aws s3 cp`. The following command saves all of the rule output files to the `ProfilerReport-1234567890` folder under the current working directory.

   ```sh
   ! aws s3 cp {rule_output_path} ./ --recursive
   ```

The `xgboost_report.html` is an autogenerated XGBoost training report by Debugger. The `xgboost_report.ipynb` is a Jupyter notebook that’s used to aggregate training results into the report. You can download all of the files, browse the HTML report file, and modify the report using the notebook.
Tip
If you are using a Jupyter notebook server, run `!pwd` to verify the current working directory.

6. Under the `/CreateXgboostReport` directory, open `xgboost_report.html`. If you are using JupyterLab, choose **Trust HTML** to see the autogenerated Debugger training report.

7. Open the `xgboost_report.ipynb` file to explore how the report is generated. You can customize and extend the training report using the Jupyter notebook file.

Download using the Amazon S3 console

1. Sign in to the AWS Management Console and open the Amazon S3 console at [https://console.aws.amazon.com/s3/](https://console.aws.amazon.com/s3/).

2. Search for the base S3 bucket. For example, if you haven’t specified any base job name, the base S3 bucket name should be in the following format: `sagemaker-<region>-111122223333`. Look up the base S3 bucket through the **Find bucket by name** field.

3. In the base S3 bucket, look up the training job name by entering your job name prefix in **Find objects by prefix** and then choosing the training job name.
4. In the training job's S3 bucket, choose `rule-output/` subfolder. There must be three subfolders for training data collected by Debugger: `debug-output/`, `profiler-output/`, and `rule-output/`.

5. In the `rule-output/` folder, choose the `CreateXgboostReport/` folder. The folder contains `xgboost_report.html` (the autogenerated report in html) and `xgboost_report.ipynb` (a Jupyter notebook with scripts that are used for generating the report).

6. Choose the `xgboost_report.html` file, choose **Download actions**, and then choose **Download**.

7. Open the downloaded `xgboost_report.html` file in a web browser.

**Debugger XGBoost Training Report Walkthrough**

This section walks you through the Debugger XGBoost training report. The report is automatically aggregated depending on the output tensor regex, recognizing what type of your training job is among binary classification, multiclass classification, and regression.

**Topics**

- Distribution of True Labels of the Dataset (p. 918)
• Loss versus Step Graph (p. 918)
• Feature Importance (p. 919)
• Confusion Matrix (p. 920)
• Evaluation of the Confusion Matrix (p. 921)
• Accuracy Rate of Each Diagonal Element Over Iteration (p. 922)
• Receiver Operating Characteristic Curve (p. 923)
• Distribution of Residuals at the Last Saved Step (p. 924)
• Absolute Validation Error per Label Bin Over Iteration (p. 925)

Distribution of True Labels of the Dataset

This histogram shows the distribution of labeled classes (for classification) or values (for regression) in your original dataset. Skewness in your dataset could contribute to inaccuracies. This visualization is available for the following model types: binary classification, multiclassification, and regression.

Loss versus Step Graph

This is a line chart that shows the progression of loss on training data and validation data throughout training steps. The loss is what you defined in your objective function, such as mean squared error. You can gauge whether the model is overfit or underfit from this plot. This section also provides insights that you can use to determine how to resolve the overfit and underfit problems. This visualization is available for the following model types: binary classification, multiclassification, and regression.
Feature Importance

There are three different types of feature importance visualizations provided: Weight, Gain and Coverage. We provide detailed definitions for each of the three in the report. Feature importance visualizations help you learn what features in your training dataset contributed to the predictions. Feature importance visualizations are available for the following model types: binary classification, multiclassification, and regression.
This visualization is only applicable to binary and multiclass classification models. Accuracy alone might not be sufficient for evaluating the model performance. For some use cases, such as healthcare and fraud detection, it’s also important to know the false positive rate and false negative rate. A confusion matrix gives you the additional dimensions for evaluating your model performance.
Evaluation of the Confusion Matrix

This section provides you with more insights on the micro, macro, and weighted metrics on precision, recall, and F1-score for your model.
### Accuracy Rate of Each Diagonal Element Over Iteration

This visualization is only applicable to binary classification and multiclass classification models. This is a line chart that plots the diagonal values in the confusion matrix throughout the training steps for each class. This plot shows you how the accuracy of each class progresses throughout the training steps. You can identify the under-performing classes from this plot.

<table>
<thead>
<tr>
<th>precision</th>
<th>recall</th>
<th>f1-score</th>
<th>support</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>0.95</td>
<td>0.97</td>
<td>0.96</td>
</tr>
<tr>
<td>1.0</td>
<td>0.94</td>
<td>0.98</td>
<td>0.96</td>
</tr>
<tr>
<td>2.0</td>
<td>0.93</td>
<td>0.90</td>
<td>0.92</td>
</tr>
<tr>
<td>3.0</td>
<td>0.91</td>
<td>0.89</td>
<td>0.90</td>
</tr>
<tr>
<td>4.0</td>
<td>0.92</td>
<td>0.89</td>
<td>0.90</td>
</tr>
<tr>
<td>5.0</td>
<td>0.94</td>
<td>0.89</td>
<td>0.91</td>
</tr>
<tr>
<td>6.0</td>
<td>0.95</td>
<td>0.94</td>
<td>0.95</td>
</tr>
<tr>
<td>7.0</td>
<td>0.92</td>
<td>0.94</td>
<td>0.93</td>
</tr>
<tr>
<td>8.0</td>
<td>0.87</td>
<td>0.87</td>
<td>0.87</td>
</tr>
<tr>
<td>9.0</td>
<td>0.85</td>
<td>0.90</td>
<td>0.88</td>
</tr>
</tbody>
</table>

**Accuracy**

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>overall</td>
<td>0.92</td>
<td></td>
<td>12000</td>
</tr>
<tr>
<td>macro avg</td>
<td>0.92</td>
<td>0.92</td>
<td>12000</td>
</tr>
<tr>
<td>weighted avg</td>
<td>0.92</td>
<td>0.92</td>
<td>12000</td>
</tr>
</tbody>
</table>
Receiver Operating Characteristic Curve

This visualization is only applicable to binary classification models. The Receiver Operating Characteristic curve is commonly used to evaluate binary classification model performance. The y-axis of the curve is True Positive Rate (TPF) and x-axis is false positive rate (FPR). The plot also displays the value for the area under the curve (AUC). The higher the AUC value, the more predictive your classifier. You can also use the ROC curve to understand the trade-off between TPR and FPR and identify the optimum classification threshold for your use case. The classification threshold can be adjusted to tune the behavior of the model to reduce more of one or another type of error (FP/FN).
Distribution of Residuals at the Last Saved Step

This visualization is a column chart that shows the residual distributions in the last step Debugger captures. In this visualization, you can check whether the residual distribution is close to normal distribution that’s centered at zero. If the residuals are skewed, your features may not be sufficient for predicting the labels.
Absolute Validation Error per Label Bin Over Iteration

This visualization is only applicable to regression models. The actual target values are split into 10 intervals. This visualization shows how validation errors progress for each interval throughout the training steps in line plots. Absolute validation error is the absolute value of difference between prediction and actual during validation. You can identify the underperforming intervals from this visualization.
Configure Debugger Using Amazon SageMaker Python SDK

To configure Debugger, use Amazon SageMaker Python SDK and specify Debugger-specific parameters while constructing SageMaker estimators. There are three parameters you need to configure: profiler_config, debugger_hook_config, and rules.

**Note**
By default, Debugger monitors and debugs SageMaker training jobs without any Debugger-specific parameters configured in SageMaker estimators. Debugger collects system metrics every 500 milliseconds and basic output tensors (scalar outputs such as loss and accuracy) every 500 steps. It also runs the ProfilerReport rule to analyze the system metrics and aggregate Studio Debugger insights dashboard and a profiling report. Debugger saves the output data in your secured S3 bucket.

**Important**
To use the new Debugger features, you need to upgrade the SageMaker Python SDK and the SMDebug client library. In your iPython kernel, Jupyter notebook, or JupyterLab environment, run the following code to install the latest versions of the libraries and restart the kernel.

```python
import sys
import IPython
!{sys.executable} -m pip install -U sagemaker smdebug
IPython.Application.instance().kernel.do_shutdown(True)
```

Construct a SageMaker Estimator with Debugger

The following example codes show how to construct a SageMaker estimator with the Debugger-specific parameters depending on a framework of your choice. Throughout the documentation in the following topics, you can find more information about how to configure the Debugger-specific parameters that you can mix and match as you want.

**Note**
The following example codes are not directly executable. You need to proceed to the next sections and configure the Debugger-specific parameters.

**TensorFlow**

To access the deep profiling feature for TensorFlow, currently you need to specify the latest AWS deep learning container images with CUDA 11. For example, you must specify the specific image URI as shown in the following example code:

```python
# An example of constructing a SageMaker TensorFlow estimator
import boto3
import sagemaker
from sagemaker.tensorflow import TensorFlow
from sagemaker.debugger import ProfilerConfig, DebuggerHookConfig, Rule, ProfilerRule, rule_configs

session=boto3.session.Session()
region=session.region_name

profiler_config=ProfilerConfig(...)  
debugger_hook_config=DebuggerHookConfig(...)  
rules=[
    Rule.sagemaker(rule_configs.built_in_rule()),
    ProfilerRule.sagemaker(rule_configs.BuiltInRule())
]

estimator=TensorFlow(
```
entry_point="directory/to/your_training_script.py",
role=sagemaker.get_execution_role(),
base_job_name="debugger-demo",
instance_count=1,
instance_type="ml.p3.2xlarge",
image_uri=f"763104351884.dkr.ecr.{region}.amazonaws.com/tensorflow-training:2.3.1-gpu-py37-cu110-ubuntu18.04"

# Debugger-specific parameters
profiler_config=profiler_config,
debugger_hook_config=debugger_hook_config,
rules=rules

estimator.fit(wait=False)

PyTorch

To access the deep profiling feature for PyTorch, currently you need to specify the latest AWS deep learning container images with CUDA 11. For example, you must specify the specific image URI as shown in the following example code:

# An example of constructing a SageMaker PyTorch estimator
import boto3
import sagemaker
from sagemaker.pytorch import PyTorch
from sagemaker.debugger import ProfilerConfig, DebuggerHookConfig, Rule, ProfilerRule, rule_configs

session=boto3.session.Session()
region=session.region_name

profiler_config=ProfilerConfig(...)
debugger_hook_config=DebuggerHookConfig(...)
rules=[
    Rule.sagemaker(rule_configs.built_in_rule()),
    ProfilerRule.sagemaker(rule_configs.BuiltInRule())
]

estimator=PyTorch(
    entry_point="directory/to/your_training_script.py",
    role=sagemaker.get_execution_role(),
    base_job_name="debugger-demo",
    instance_count=1,
    instance_type="ml.p3.2xlarge",
    image_uri=f"763104351884.dkr.ecr.{region}.amazonaws.com/pytorch-training:1.6.0-gpu-py36-cu110-ubuntu18.04",

    # Debugger-specific parameters
    profiler_config=profiler_config,
    debugger_hook_config=debugger_hook_config,
    rules=rules
)

estimator.fit(wait=False)

MXNet

# An example of constructing a SageMaker MXNet estimator
import boto3
import sagemaker
from sagemaker.mxnet import MXNet
from sagemaker.debugger import ProfilerConfig, DebuggerHookConfig, Rule, ProfilerRule, rule_configs
profiler_config=ProfilerConfig(...)  
debugger_hook_config=DebuggerHookConfig(...)  
rules=[  
    Rule.sagemaker(rule_configs.built_in_rule()),  
    ProfilerRule.sagemaker(rule_configs.BuiltInRule())  
]  

estimator=MXNet(  
    entry_point="directory/to/your_training_script.py",  
    role=sagemaker.get_execution_role(),  
    base_job_name="debugger-demo",  
    instance_count=1,  
    instance_type="ml.p3.2xlarge",  
    framework_version="1.7.0",  
    py_version="py37",  
    # Debugger-specific parameters  
    profiler_config=profiler_config,  
    debugger_hook_config=debugger_hook_config,  
    rules=rules  
)  

estimator.fit(wait=False)

**Note**  
For MXNet, when configuring the `profiler_config` parameter, you can only configure for system monitoring. Profiling framework metrics is not supported for MXNet.

**XGBoost**

# An example of constructing a SageMaker XGBoost estimator  
import sagemaker  
from sagemaker.xgboost.estimator import XGBoost  
from sagemaker.debugger import ProfilerConfig, DebuggerHookConfig, Rule, ProfilerRule, rule_configs  

profiler_config=ProfilerConfig(...)  
debugger_hook_config=DebuggerHookConfig(...)  
rules=[  
    Rule.sagemaker(rule_configs.built_in_rule()),  
    ProfilerRule.sagemaker(rule_configs.BuiltInRule())  
]  

estimator=XGBoost(  
    entry_point="directory/to/your_training_script.py",  
    role=sagemaker.get_execution_role(),  
    base_job_name="debugger-demo",  
    instance_count=1,  
    instance_type="ml.p3.2xlarge",  
    framework_version="1.2-1",  
    # Debugger-specific parameters  
    profiler_config=profiler_config,  
    debugger_hook_config=debugger_hook_config,  
    rules=rules  
)  

estimator.fit(wait=False)

**Note**  
For XGBoost, when configuring the `profiler_config` parameter, you can only configure for system monitoring. Profiling framework metrics is not supported for XGBoost.
Generic estimator

```python
# An example of constructing a SageMaker generic estimator using the XGBoost algorithm
import boto3
import sagemaker
from sagemaker.estimator import Estimator
from sagemaker import image_uris
from sagemaker.debugger import ProfilerConfig, DebuggerHookConfig, Rule, ProfilerRule, rule_configs

profiler_config = ProfilerConfig(...)
database_hook_config = DebuggerHookConfig(...)

rules = [
    Rule.sagemaker(rule_configs.built_in_rule()),
    ProfilerRule.sagemaker(rule_configs.BuiltInRule())
]

region = boto3.Session().region_name
xgboost_container = sagemaker.image_uris.retrieve("xgboost", region, "1.2-1")

estimator = Estimator(
    role=sagemaker.get_execution_role(),
    image_uri=xgboost_container,
    base_job_name="debugger-demo",
    instance_count=1,
    instance_type="ml.m5.2xlarge",

    # Debugger-specific parameters
    profiler_config=profiler_config,
    debugger_hook_config=debugger_hook_config,
    rules=rules
)

estimator.fit(wait=False)
```

Where you configure the following parameters:

- **profiler_config** parameter – Configure Debugger to collect system metrics and framework metrics from your training job and save into your secured S3 bucket URI or local machine. To learn how to configure the profiler_config parameter, see Configure Debugger Monitoring Hardware System Resource Utilization (p. 930) and Configure Debugger Framework Profiling (p. 930).

- **debugger_hook_config** parameter – Configure Debugger to collect output tensors from your training job and save into your secured S3 bucket URI or local machine. To learn how to configure the debugger_hook_config parameter, see Configure Debugger Hook to Save Tensors (p. 934).

- **rules** parameter – Configure this parameter to enable Debugger built-in rules that you want to run in parallel. The rules automatically analyze your training job and find training issues. The ProfilerReport rule saves the Debugger profiling reports in your secured S3 bucket URI. To learn how to configure the rules parameter, see Configure Debugger Built-in Rules (p. 941).

**Note**

Debugger securely saves output data in subfolders of your default S3 bucket. For example, the format of the default S3 bucket URI is s3://sagemaker-<region>-<12digit_account_id>/<base-job-name>/<debugger-subfolders/>. There are three subfolders created by Debugger: debug-output, profiler-output, and rule-output. You can also retrieve the default S3 bucket URIs using the SageMaker estimator classmethods (p. 947).

See the following topics to find out how to configure the Debugger-specific parameters in detail.
Configure Debugger Monitoring Hardware System Resource Utilization

To adjust Debugger system monitoring time intervals, use the `ProfilerConfig` API operation to create a parameter object while constructing a SageMaker framework or generic estimator depending on your preference.

**Note**
By default, for all SageMaker training jobs, Debugger collects hardware system utilization data from Amazon EC2 instances every 500 milliseconds for system monitoring, without any Debugger-specific parameters specified in SageMaker estimators. Debugger saves the system metrics in a default S3 bucket. The format of the default S3 bucket URI is `s3://sagemaker-<region>-<12digit_account_id>/<training-job-name>/profiler-output/`.

The following example code shows how to set up the `profiler_config` parameter with a system monitoring time interval of 1000 milliseconds.

```python
from sagemaker.debugger import ProfilerConfig
profiler_config=ProfilerConfig(
    system_monitor_interval_millis=1000
)
```

- `system_monitor_interval_millis` (int) – Specify the monitoring intervals in milliseconds to record system metrics. Available values are 100, 200, 500, 1000 (1 second), 5000 (5 seconds), and 60000 (1 minute) milliseconds. The default value is 500 milliseconds.

To see the progress of system monitoring, see Debugger for Insights on Studio (p. 890).

Configure Debugger Framework Profiling

To enable Debugger framework profiling, configure the `framework_profile_params` parameter when you construct an estimator. Debugger framework profiling collects framework metrics, such as data from initialization stage, data loader processes, Python operators of deep learning frameworks and training scripts, detailed profiling within and between steps, with cProfile or Pyinstrument options. Using the `FrameworkProfile` class, you can configure custom framework profiling options.

**Note**
Before getting started with Debugger framework profiling, verify that the framework used to build your model is supported by Debugger for framework profiling. For more information, see Supported Frameworks and Algorithms (p. 873).
Debugger saves the framework metrics in a default S3 bucket. The format of the default S3 bucket URI is

```
s3://sagemaker-<region>-<12digit_account_id>/<training-job-name>/profiler-output/.
```

### Start a Training Job with the Default System Monitoring and Framework Profiling

The following example code is the simplest `profiler_config` parameter setting to start the default system monitoring and the default framework profiling. The `FrameworkProfile` class in the following example code initiates the default framework profiling when a training job starts. Debugger framework profiling includes the following options: detailed profiling, data loader profiling, and Python profiling.

```python
from sagemaker.debugger import ProfilerConfig, FrameworkProfile

profiler_config=ProfilerConfig(
    framework_profile_params=FrameworkProfile()
)
```

With this `profiler_config` parameter configuration, Debugger calls the default settings of monitoring and profiling. Debugger monitors system metrics every 500 milliseconds, profiles the fifth step with the detailed profiling option, the seventh step with the data loader profiling option, and the ninth, tenth, and eleventh steps with the Python profiling option.

To find available profiling configuration options, the default parameter settings, and examples of how to configure them, see Start a Training Job with the Default System Monitoring and Customized Framework Profiling with Different Profiling Options (p. 932) and SageMaker Debugger APIs – FrameworkProfile in the Amazon SageMaker Python SDK.

If you want to change the system monitoring interval and enable the default framework profiling, you can specify the `system_monitor_interval_millis` parameter explicitly with the `framework_profile_params` parameter. For example, to monitor every 1000 milliseconds and enable the default framework profiling, use the following example code.

```python
from sagemaker.debugger import ProfilerConfig, FrameworkProfile

profiler_config=ProfilerConfig(
    system_monitor_interval_millis=1000,
    framework_profile_params=FrameworkProfile()
)
```

For more information about the `FrameworkProfile` class, see SageMaker Debugger APIs – FrameworkProfile in the Amazon SageMaker Python SDK.

### Start a Training Job with the Default System Monitoring and Customized Framework Profiling for Target Steps or a Target Time Range

If you want to specify target steps or target time intervals to profile your training job, you need to specify parameters for the `FrameworkProfile` class. The following code examples show how to specify the target ranges for profiling along with system monitoring.

- **For a target step range**

  With the following example configuration, Debugger monitors the entire training job every 500 milliseconds (the default monitoring) and profiles a target step range from step 5 to step 15 (for 10 steps).

```python
from sagemaker.debugger import ProfilerConfig, FrameworkProfile

profiler_config=ProfilerConfig(
    system_monitor_interval_millis=1000,
    framework_profile_params=FrameworkProfile()
)
```
With the following example configuration, Debugger monitors the entire training job every 1000 milliseconds and profiles a target step range from step 5 to step 15 (for 10 steps).

```python
from sagemaker.debugger import ProfilerConfig, FrameworkProfile

profiler_config=ProfilerConfig(
    framework_profile_params=FrameworkProfile(start_step=5, num_steps=10)
)
```

For a target time range

With the following example configuration, Debugger monitors the entire training job every 500 milliseconds (the default monitoring) and profiles a target time range from the current Unix time for 600 seconds.

```python
import time
from sagemaker.debugger import ProfilerConfig, FrameworkProfile

profiler_config=ProfilerConfig(
    framework_profile_params=FrameworkProfile(start_unix_time=int(time.time()), duration=600)
)
```

With the following example configuration, Debugger monitors the entire training job every 1000 milliseconds and profiles a target time range from the current Unix time for 600 seconds.

```python
import time
from sagemaker.debugger import ProfilerConfig, FrameworkProfile

profiler_config=ProfilerConfig(
    system_monitor_interval_millis=1000,
    framework_profile_params=FrameworkProfile(start_unix_time=int(time.time()), duration=600)
)
```

The framework profiling is performed for all of the profiling options at the target step or time range.

To find more information about available profiling options, see SageMaker Debugger APIs – FrameworkProfile in the Amazon SageMaker Python SDK.

The next section show you how to script the available profiling options.

Start a Training Job with the Default System Monitoring and Customized Framework Profiling with Different Profiling Options

You can use the following profiling configuration classes to micromanage the framework profiling options:

- **DetailedProfilingConfig** – Specify a target step or time range to profile framework operations using the native framework profilers (TensorFlow profiler and PyTorch profiler). For example, if using TensorFlow, the Debugger hooks enable the TensorFlow profiler to collect TensorFlow-specific framework metrics. Detailed profiling enables you to profile all framework operators at a pre-step (before the first step), within steps, and between steps of a training job.
Note
The detailed profiling might significantly increase GPU memory consumption. It is not recommended to enable the detailed profiling for more than a couple of steps.

- DataloaderProfilingConfig – Specify a target step or time range to profile deep learning framework data loader processes. Debugger collects every data loader event of the frameworks.

  Note
  The data loader profiling might lower the training performance while collecting information from data loaders. We don't recommend that you enable the data loader profiling for more than a couple of steps.
  Debugger is preconfigured to annotate data loader processes only for the AWS deep learning containers. Debugger cannot profile data loader processes from any other custom or external training containers.

- PythonProfilingConfig – Specify a target step or time range to profile Python functions. You can also choose between two Python profilers: cProfile and Pyinstrument.

  cProfile – The standard Python profiler. cProfile collects for every Python operator called during training. With cProfile, Debugger saves cumulative time and annotation of each function call, providing a complete detail of Python functions. In deep learning, for example, the most frequently called functions might be the convolutional filters and backward pass operators, and cProfile profiles every single of them. For the cProfile option, you can further select a timer option: total time, CPU time, and off-CPU time. While you can profile every function calls executing on processors (both CPU and GPU) in CPU time, you can also identify I/O or network bottlenecks with the off-CPU time option. The default is total time, and Debugger profiles both CPU and off-CPU time. With cProfile, you are able to drill down to every single functions when analyzing the profile data.

  Pyinstrument – Pyinstrument is a low overhead Python profiler that works based on sampling. With the Pyinstrument option, Debugger samples profiling events every millisecond. Because Pyinstrument measures elapsed wall clock time instead of CPU time, the Pyinstrument option can be a better choice over the cProfile option for reducing profiling noises (filtering out irrelevant function calls that are cumulatively fast) and capturing operators that are actually compute intensive (cumulatively slow) for training your model. With Pyinstrument, you are able to see a tree of function calls and better understand the structure and root cause of the slowness.

  Note
  Enabling Python profiling might result in slowing down the overall training time. In case of cProfile, Python operators that are the most frequently called are profiled at every call, so the processing time on profiling increases with respect to the number of calls. For Pyinstrument, the cumulative profiling time increases with respect to time because of its sampling mechanism.

The following example configuration shows the full structure when you use the different profiling options with specified values.

```python
import time
from sagemaker.debugger import (ProfilerConfig,
                                 FrameworkProfile,
                                 DetailedProfilingConfig,
                                 DataloaderProfilingConfig,
                                 PythonProfilingConfig)

profiler_config=ProfilerConfig(
    system_monitor_interval_millis=500,
    framework_profile_params=FrameworkProfile(
        detailed_profiling_config=DetailedProfilingConfig(
            start_step=5,
            num_steps=1
        ),
        dataloader_profiling_config=DataloaderProfilingConfig(
            start_step=7,
        ),
    ),
)  ```
Configure Debugger Using SageMaker Python SDK

```python
class CustomSession(SagemakerSession):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.debugger_config = DebugConfig()

    def configure_debugger(self, project_name, code_root, s3_output_path, num_steps=1,
                           python_profiling_config=PythonProfilingConfig(
                       start_step=9,
                       num_steps=1,
                       python_profiler="cProfile",
                       cprofile_timer="total_time"
                   )
               )

    def start_training_job(self, input_data, output_data):  # This is the function to train a model
        self.debugger_config = self.configure_debugger(  # Configure the debugger
                project_name, code_root, output_data,
                num_steps=num_steps,
                python_profiling_config=python_profiling_config
            )

        # Train the model
```

To find more information about available profiling options, see `DetailedProfilingConfig`, `DataloaderProfilingConfig`, and `PythonProfilingConfig` in the Amazon SageMaker Python SDK.

Updating Debugger System Monitoring and Framework Profiling Configuration while a Training Job is Running

If you want to enable or update Debugger monitoring and profiling configuration to a training job that is currently running, use the following SageMaker estimator extension methods:

- To enable Debugger system monitoring for a running training job and receive a Debugger profiling report, use the following:

  ```python
  estimator.enable_default_profiling()
  ```

  When you use the `enable_default_profiling` method, Debugger initiates the default system monitoring and the ProfileReport built-in rule, which generates a comprehensive profiling report at the end of the training job. This method can be called only if the current training job is running without both Debugger monitoring and profiling.

  For more information, see `estimator.enable_default_profiling` in the Amazon SageMaker Python SDK.

- To enable Debugger built-in rules, system monitoring, and framework profiling with customizable configuration parameters, use the following:

  ```python
  estimator.update_profiler(
      rules=[ProfilerRule.sagemaker(rule_configs.BuiltinRule())],
      system_monitor_interval_millis=500,
      framework_profile_params=FrameworkProfile()
  )
  ```

  For more information, see `estimator.update_profiler` in the Amazon SageMaker Python SDK.

Configure Debugger Hook to Save Tensors

*Tensors* are data collections of updated parameters from backward and forward pass of each training iteration, and Debugger collects the output tensors to analyze the state of a training job. The Amazon SageMaker Debugger `CollectionConfig` and `DebuggerHookConfig` API operations provide methods to group tensors into *collections*, and save them to a target S3 bucket.

**Note**

By default, for all SageMaker training jobs, Debugger collects loss and accuracy output scalars from the training jobs every 500 steps, without any Debugger-specific parameters specified in SageMaker estimators. Debugger saves the output data in a default S3 bucket. The format of the default S3 bucket URI is `s3://sagemaker-<region>-<12digit_account_id>/<training-job-name>/debug-output/`. 

934
While constructing a SageMaker estimator, enable Debugger by specifying the hook configuration parameter, debugger_hook_config. The following steps include examples of how to set up the debugger_hook_config using the CollectionConfig and DebuggerHookConfig API operations to pull tensors out of your training jobs and save them. If you use Debugger-supported AWS containers for zero script change, you can simply run the training job without changing your training script. You can also use Debugger for training jobs running in any other Debugger-supported AWS containers with script mode, making a minimal change of your training script.

**Configure Debugger Tensor Collections Using the Collectivisation API Operation**

Use the CollectionConfig API operation to configure tensor collections. Debugger provides pre-built tensor collections that covers variety of regular expressions (regex) of parameters if using Debugger-supported deep learning frameworks and machine learning algorithms. As shown in the following example code, add the built-in tensor collections you want to debug.

```python
from sagemaker.debugger import CollectionConfig

collection_configs=[
    CollectionConfig(name="weights"),
    CollectionConfig(name="gradients")
]
```

The preceding collections set up the Debugger hook to save the tensors every 500 steps based on the default "save_interval" value.

For a full list of available Debugger built-in collections, see Debugger Built-in Collections.

If you want to customize the built-in collections, such as changing the save intervals and tensor regex, use the following CollectionConfig template to adjust parameters.

```python
from sagemaker.debugger import CollectionConfig

collection_configs=[
    CollectionConfig(
        name="tensor_collection",
        parameters={
            "key_1": "value_1",
            "key_2": "value_2",
            ...
            "key_n": "value_n"
        }
    )
]
```

For more information about available parameter keys, see CollectionConfig in the Amazon SageMaker Python SDK. For example, the following code example shows how you can adjust the save intervals of the "losses" tensor collection at different phases of training: save loss every 100 steps in training phase and validation loss every 10 steps in validation phase.

```python
from sagemaker.debugger import CollectionConfig

collection_configs=[
    CollectionConfig(
        name="losses",
        parameters={
            "train.save_interval": "100",
            "eval.save_interval": "10"
        }
    )
]
```
Configure Debugger Hook to Save Tensors

Use the `DebuggerHookConfig` class to create a `debugger_hook_config` object using the `collection_configs` object you created in the previous step.

```python
from sagemaker.debugger import DebuggerHookConfig

ddebugger_hook_config = DebuggerHookConfig(
    collection_configs=collection_configs
)
```

Debugger saves the model training output tensors into the default S3 bucket. The format of the default S3 bucket URI is `s3://sagemaker-<region>-<12digit_account_id>/<training-job-name>/debug-output/`.

If you want to specify an exact S3 bucket URI, use the following code example:

```python
from sagemaker.debugger import DebuggerHookConfig

ddebugger_hook_config = DebuggerHookConfig(
    s3_output_path="specify-your-s3-bucket-uri",
    collection_configs=collection_configs
)
```

For more information, see `DebuggerHookConfig` in the Amazon SageMaker Python SDK.

Example Notebooks and Code Samples to Configure Debugger Hook

In the following sections, notebooks and code examples of how to use Debugger hook to save, access, and visualize output tensors are provided.

Topics

- Tensor Visualization Example Notebooks (p. 936)
- Save Tensors Using Debugger Built-in Collections (p. 938)
- Save Tensors Using Debugger Modified Built-in Collections (p. 939)
- Save Tensors Using Debugger Custom Collections (p. 939)
- Use Amazon SageMaker Debugger to Save TensorBoard Summaries and Histograms (p. 940)

Tensor Visualization Example Notebooks

The following two notebook examples show advanced use of Amazon SageMaker Debugger for visualizing tensors. Debugger provides a transparent view into training deep learning models.

- Interactive Tensor Analysis in SageMaker Studio Notebook with MXNet

  This notebook example shows how to visualize saved tensors using Amazon SageMaker Debugger. By visualizing the tensors, you can easily see how the tensor values change while training deep learning algorithms. This notebook includes a training job with a poorly configured neural network and uses Amazon SageMaker Debugger to aggregate and analyze tensors, including gradients,
activation outputs, and weights. For example, the following plot shows the distribution of gradients of a convolutional layer that is suffering from a vanishing gradient problem.

![Distribution of Gradients](image)

This notebook also illustrates how a good initial hyperparameter setting improves the training process by generating the same tensor distribution plots.

**Visualizing and Debugging Tensors from MXNet Model Training**

This notebook example shows how to save and visualize tensors from an MXNet Gluon model training job using Amazon SageMaker Debugger. It illustrates that Debugger is set to save all tensors to an Amazon S3 bucket and retrieves ReLu activation outputs for the visualization. The following figure shows a three-dimensional visualization of the ReLu activation outputs. The color scheme is set for blue to indicate values close to 0 and yellow to indicate values close to 1.

![Three-dimensional Visualization](image)

In this notebook, the `TensorPlot` class imported from `tensor_plot.py` is designed to plot convolutional neural networks (CNNs) that take two-dimensional images for inputs. The
Configure Debugger Using SageMaker Python SDK

The tensor_plot.py script provided with the notebook retrieves tensors using Debugger and visualizes the CNN. You can run this notebook on SageMaker Studio to reproduce the tensor visualization and implement your own convolutional neural network model to it.

- **Real-time Tensor Analysis in a SageMaker Notebook with MXNet**

This example guides you through installing required components for emitting tensors in an Amazon SageMaker training job and using the Debugger API operations to access those tensors while training is running. A gluon CNN model is trained on the Fashion MNIST dataset. While the job is running, you will see how Debugger retrieves activation outputs of the first convolutional layer from each of 100 batches and visualizes them. Also, this will show you how to visualize weights after the job is done.

**Save Tensors Using Debugger Built-in Collections**

You can use built-in collections of tensors using the `CollectionConfig` API and save them using the `DebuggerHookConfig` API. The following example shows how to use the default settings of Debugger hook configurations to construct a SageMaker TensorFlow estimator. You can also utilize this for MXNet, PyTorch, and XGBoost estimators.

**Note**

In the following example code, the `s3_output_path` parameter for `DebuggerHookConfig` is optional. If you do not specify it, Debugger saves the tensors at `s3://<output_path>/debug-output/`, where the `<output_path>` is the default output path of SageMaker training jobs. For example:

```
s3://sagemaker-us-east-1-111122223333/sagemaker-debugger-training-YYYY-MM-DD-HH-MM-SS-123/debug-output
```

```python
import sagemaker
from sagemaker.tensorflow import TensorFlow
from sagemaker.debugger import DebuggerHookConfig, CollectionConfig

# use Debugger CollectionConfig to call built-in collections
collection_configs=[
    CollectionConfig(name="weights"),
    CollectionConfig(name="gradients"),
    CollectionConfig(name="losses"),
    CollectionConfig(name="biases")
]

# configure Debugger hook
# set a target S3 bucket as you want
sagemaker_session=sagemaker.Session()
BUCKET_NAME=sagemaker_session.default_bucket()
LOCATION_IN_BUCKET='debugger-built-in-collections-hook'

hook_config=DebuggerHookConfig(
    s3_output_path='s3://{BUCKET_NAME}/{LOCATION_IN_BUCKET}'.format(BUCKET_NAME=BUCKET_NAME,
                                                                    LOCATION_IN_BUCKET=LOCATION_IN_BUCKET),
    collection_configs=collection_configs
)

# construct a SageMaker TensorFlow estimator
sagemaker_estimator=TensorFlow(
    entry_point='directory/to/your_training_script.py',
    role=sm.get_execution_role(),
    base_job_name='debugger-demo-job',
    train_instance_count=1,
    train_instance_type="ml.m4.xlarge",
    framework_version="2.3.0",
    py_version="py37",
)```
To see a list of Debugger built-in collections, see Debugger Built-in Collections.

Save Tensors Using Debugger Modified Built-in Collections

You can modify the Debugger built-in collections using the CollectionConfig API operation. The following example shows how to tweak the built-in losses collection and construct a SageMaker TensorFlow estimator. You can also use this for MXNet, PyTorch, and XGBoost estimators.

```python
import sagemaker
from sagemaker.tensorflow import TensorFlow
from sagemaker.debugger import DebuggerHookConfig, CollectionConfig

# use Debugger CollectionConfig to call and modify built-in collections
collection_configs = [
    CollectionConfig(
        name="losses",
        parameters={"save_interval": "50"})
]

# configure Debugger hook
# set a target S3 bucket as you want
sagemaker_session = sagemaker.Session()
BUCKET_NAME = sagemaker_session.default_bucket()
LOCATION_IN_BUCKET = 'debugger-modified-collections-hook'

hook_config = DebuggerHookConfig(
    s3_output_path='s3://{BUCKET_NAME}/{LOCATION_IN_BUCKET}'.format(BUCKET_NAME=BUCKET_NAME, LOCATION_IN_BUCKET=LOCATION_IN_BUCKET),
    collection_configs=collection_configs
)

# construct a SageMaker TensorFlow estimator
sagemaker_estimator = TensorFlow(
    entry_point='directory/to/your_training_script.py',
    role=sm.get_execution_role(),
    base_job_name='debugger-demo-job',
    train_instance_count=1,
    train_instance_type="ml.m4.xlarge",
    framework_version="2.3.0",
    py_version="py37",
    # debugger-specific hook argument below
    debugger_hook_config=hook_config
)

sagemaker_estimator.fit()
```

For a full list of CollectionConfig parameters, see Debugger CollectionConfig API.

Save Tensors Using Debugger Custom Collections

You can also save a reduced number of tensors instead of the full set of tensors (for example, if you want to reduce the amount of data saved in your Amazon S3 bucket). The following example shows how to customize the Debugger hook configuration to specify target tensors that you want to save. You can use this for TensorFlow, MXNet, PyTorch, and XGBoost estimators.

```python
import sagemaker
from sagemaker.tensorflow import TensorFlow
from sagemaker.debugger import DebuggerHookConfig, CollectionConfig

# configure Debugger collection
# set a target S3 bucket as you want
sagemaker_session = sagemaker.Session()
BUCKET_NAME = sagemaker_session.default_bucket()
LOCATION_IN_BUCKET = 'debugger-modified-collections-hook'

hook_config = DebuggerHookConfig(
    s3_output_path='s3://{BUCKET_NAME}/{LOCATION_IN_BUCKET}'.format(BUCKET_NAME=BUCKET_NAME, LOCATION_IN_BUCKET=LOCATION_IN_BUCKET),
    collection_configs=[
        CollectionConfig(
            name="losses",
            parameters={"save_interval": "50"})
    ]
)

# construct a SageMaker TensorFlow estimator
sagemaker_estimator = TensorFlow(
    entry_point='directory/to/your_training_script.py',
    role=sm.get_execution_role(),
    base_job_name='debugger-demo-job',
    train_instance_count=1,
    train_instance_type="ml.m4.xlarge",
    framework_version="2.3.0",
    py_version="py37",
    # debugger-specific hook argument below
    debugger_hook_config=hook_config
)

sagemaker_estimator.fit()
```
import sagemaker
from sagemaker.tensorflow import TensorFlow
from sagemaker.debugger import DebuggerHookConfig, CollectionConfig

# use Debugger CollectionConfig to create a custom collection
collection_configs = [CollectionConfig(
    name="custom_activations_collection",
    parameters={
        "include_regex": "relu|tanh", # Required
        "reductions": "mean, variance, max, abs_mean, abs_variance, abs_max"
    }
) ]

# configure Debugger hook
# set a target S3 bucket as you want
sagemaker_session = sagemaker.Session()
BUCKET_NAME = sagemaker_session.default_bucket()
LOCATION_IN_BUCKET = 'debugger-custom-collections-hook'
hook_config = DebuggerHookConfig(
    s3_output_path='s3://{BUCKET_NAME}/{LOCATION_IN_BUCKET}'.format(BUCKET_NAME=BUCKET_NAME, LOCATION_IN_BUCKET=LOCATION_IN_BUCKET),
    collection_configs=collection_configs
)

# construct a SageMaker TensorFlow estimator
sagemaker_estimator = TensorFlow(
    entry_point='directory/to/your_training_script.py',
    role=sm.get_execution_role(),
    base_job_name='debugger-demo-job',
    train_instance_count=1,
    train_instance_type="ml.m4.xlarge",
    framework_version="2.3.0",
    py_version="py37",
    # debugger-specific hook argument below
    debugger_hook_config=hook_config
)

sagemaker_estimator.fit()

For a full list of CollectionConfig parameters, see Debugger CollectionConfig.

Use Amazon SageMaker Debugger to Save TensorBoard Summaries and Histograms

Amazon SageMaker Debugger can automatically generate TensorBoard scalar summaries, distributions, and histograms for saved tensors. You enable this by passing a TensorBoardOutputConfig object when you create an estimator, as shown in the following example. You can also choose to disable or enable histograms for individual collections. By default, the save_histogram flag for a collection is set to True. Debugger adds scalar summaries to TensorBoard for all ScalarCollections and scalars saved through hook.save_scalar. For more information about scalar collections and the save_scalar method, see the Debugger Hook API.

The following example saves weights and gradients as full tensors, and also saves the gradients as histograms and distributions that can be visualized with TensorBoard. Amazon SageMaker Debugger saves them in the Amazon S3 location passed in the TensorBoardOutputConfig object.

import sagemaker
from sagemaker.debugger import DebuggerHookConfig, CollectionConfig, TensorBoardOutputConfig
sagemaker_session=sagemaker.Session()
BUCKET_NAME=sagemaker_session.default_bucket()
LOCATION_IN_BUCKET='smdebug-hook-tensorboard'

hook_config=DebuggerHookConfig(
    s3_output_path='s3://{BUCKET_NAME}/{LOCATION_IN_BUCKET}'.format(BUCKET_NAME=BUCKET_NAME, LOCATION_IN_BUCKET=LOCATION_IN_BUCKET),
    collection_configs=[
        CollectionConfig(
            name="weights",
            parameters={"save_histogram": "False"}),
        CollectionConfig(name="gradients"),
    ]
)

tb_config=TensorBoardOutputConfig('s3://{BUCKET_NAME}/{LOCATION_IN_BUCKET}'.format(BUCKET_NAME=BUCKET_NAME, LOCATION_IN_BUCKET=LOCATION_IN_BUCKET))

from sagemaker.tensorflow import TensorFlow
sagemaker_estimator=TensorFlow(
    entry_point='directory/to/your_training_script.py',
    role=sm.get_execution_role(),
    base_job_name='smdebug-demo-job',
    train_instance_count=1,
    train_instance_type="ml.m4.xlarge",
    framework_version="2.3.0",
    py_version="py37",
    # debugger-specific arguments below
debugger_hook_config=hook_config,
tensorboard_output_config=tb_config
)
sagemaker_estimator.fit(wait=True)


Configure Debugger Built-in Rules

Amazon SageMaker Debugger rules analyze tensors emitted during the training of a model. Debugger offers the Rule API operation that monitors training job progress and errors for the success of training your model. For example, the rules can detect whether gradients are getting too large or too small, whether a model is overfitting or overtraining, and whether a training job does not decrease loss function and improve. To see a full list of available built-in rules, see List of Debugger Built-in Rules (p. 948).

Note
The built-in rules are prepared in Amazon SageMaker processing containers and fully managed by SageMaker Debugger. By default, Debugger initiates the ProfilerReport (p. 951) rule for all SageMaker training jobs, without any Debugger-specific rule parameter specified to the SageMaker estimators. The ProfilerReport rule invokes all of the following built-in rules for monitoring system bottlenecks and profiling framework metrics:

- BatchSize (p. 952)
- CPUBottleneck (p. 953)
- GPUMemoryIncrease (p. 955)
- IOBottleneck (p. 956)
- LoadBalancing (p. 957)
- LowGPUUtilization (p. 957)
- OverallSystemUsage (p. 959)
• MaxInitializationTime (p. 959)
• OverallFrameworkMetrics (p. 960)
• StepOutlier (p. 960)

Debugger saves the profiling report in a default S3 bucket. The format of the default S3 bucket URI is s3://sagemaker-<region>-<12digit_account_id>/<training-job-name>/rule-output/. For more information about how to download the profiling report, see SageMaker Debugger Profiling Report (p. 903). SageMaker Debugger fully manages the built-in rules and analyzes your training job in parallel. For more information about billing, see the Amazon SageMaker Studio is available at no additional charge section of the Amazon SageMaker Pricing page.

In the following topics, learn how to use the Debugger built-in rules.

Topics
• Use Debugger Built-in Rules with the Default Parameter Settings (p. 942)
• Use Debugger Built-in Rules with Custom Parameter Values (p. 943)
• Example Notebooks and Code Samples to Configure Debugger Rules (p. 944)

Use Debugger Built-in Rules with the Default Parameter Settings

To specify Debugger built-in rules in an estimator, you need to configure a rules list object. The following example code shows the basic structure of listing the Debugger built-in rules:

```python
from sagemaker.debugger import Rule, ProfilerRule, rule_configs

rules=[
    ProfilerRule.sagemaker(rule_configs.BuiltInProfilerRuleName_1()),
    ProfilerRule.sagemaker(rule_configs.BuiltInProfilerRuleName_2()),
    ...
    ProfilerRule.sagemaker(rule_configs.BuiltInProfilerRuleName_n()),
    Rule.sagemaker(rule_configs.built_in_rule_name_1()),
    Rule.sagemaker(rule_configs.built_in_rule_name_2()),
    ...
    Rule.sagemaker(rule_configs.built_in_rule_name_n())
]
```

For more information about default parameter values and descriptions of the built-in rule, see List of Debugger Built-in Rules (p. 948).

For example, to inspect the overall training performance and progress of your model, construct a SageMaker estimator with the following built-in rule configuration:

```python
from sagemaker.debugger import Rule, rule_configs

rules=[
    ProfilerRule.sagemaker(rule_configs.ProfilerReport()),
    Rule.sagemaker(rule_configs.loss_not_decreasing()),
    Rule.sagemaker(rule_configs.overfit()),
    Rule.sagemaker(rule_configs.overtraining()),
    Rule.sagemaker(rule_configs.stalled_training_rule())
]
```

When you start the training job, Debugger collects system resource utilization data every 500 milliseconds and the loss and accuracy values every 500 steps by default. Debugger
analyzes the resource utilization to identify if your model is having bottleneck problems. The
loss_not_decreasing, overfit, overtraining, and stalled_training_rule monitors if your
model is optimizing the loss function without those training issues. If the rules detect training anomalies,
the rule evaluation status changes to IssueFound. You can set up automated actions, such as notifying
training issues and stopping training jobs using Amazon CloudWatch Events and AWS Lambda. For more
information, see Action on Amazon SageMaker Debugger Rules (p. 998).

Use Debugger Built-in Rules with Custom Parameter Values

If you want to adjust the built-in rule parameter values and customize tensor collection regex, configure
the base_config and rule_parameters parameters for the ProfilerRule.sagemaker and
Rule.sagemaker class methods. In case of the Rule.sagemaker class methods, you can also customize
tensor collections through the collections_to_save parameter. The instruction of how to use
the CollectionConfig class is provided at Configure Debugger Tensor Collections Using the
Collectivisation API Operation (p. 935).

Use the following configuration template for built-in rules to customize parameter values. By changing
the rule parameters as you want, you can adjust the sensitivity of the rules to be triggered.

- The base_config argument is where you call the built-in rule methods.
- The rule_parameters argument is to adjust the default key values of the built-in rules listed in List
  of Debugger Built-in Rules (p. 948).
- The collections_to_save argument takes in a tensor configuration through the
  CollectionConfig API, which requires name and parameters arguments.
- To find available tensor collections for name, see Debugger Built-in Tensor Collections.
- For a full list of adjustable parameters, see Debugger CollectionConfig API.

For more information about the Debugger rule class, methods, and parameters, see SageMaker
Debugger Rule class in the Amazon SageMaker Python SDK.

```python
from sagemaker.debugger import Rule, ProfilerRule, rule_configs, CollectionConfig
rules=[
    ProfilerRule.sagemaker(
        base_config=rule_configs.BuiltInProfilerRuleName(),
        rule_parameters={
            "key": "value"
        }
    ),
    Rule.sagemaker(
        base_config=rule_configs.built_in_rule_name(),
        rule_parameters={
            "key": "value"
        }
    ),
    collections_to_save=[
        CollectionConfig(
            name="tensor_collection_name",
            parameters={
                "key": "value"
            }
        )
    ]
]
```

The parameter descriptions and value customization examples are provided for each rule at List of
Debugger Built-in Rules (p. 948).
Example Notebooks and Code Samples to Configure Debugger Rules

In the following sections, notebooks and code samples of how to use Debugger rules to monitor SageMaker training jobs are provided.

Topics
- Debugger Built-in Rules Example Notebooks (p. 944)
- Debugger Built-in Rules Example Code (p. 945)
- Use Debugger Built-in Rules with Parameter Modifications (p. 946)

Debugger Built-in Rules Example Notebooks

The following example notebooks show how to use Debugger built-in rules when running training jobs with Amazon SageMaker:

- Using a SageMaker Debugger built-in rule with TensorFlow
- Using a SageMaker Debugger built-in rule with Managed Spot Training and MXNet
- Using a SageMaker Debugger built-in rule with XGBoost
- Using a SageMaker Debugger built-in rule with parameter modifications for a real-time training job analysis with XGBoost

While running the example notebooks in SageMaker Studio, you can find the training job trial created on the Studio Experiment List tab. For example, as shown in the following screenshot, you can find and open a Describe Trial Component window of your current training job. On the Debugger tab, you can check if the Debugger rules, vanishing_gradient() and loss_not_decreasing(), are monitoring the training session in parallel. For a full instruction of how to find your training job trial components in the Studio UI, see SageMaker Studio - View Experiments, Trials, and Trial Components.
There are two ways of using the Debugger built-in rules in the SageMaker environment: deploy the built-in rules as it is prepared or adjust their parameters as you want. The following topics show you how to use the built-in rules with example codes.

**Debugger Built-in Rules Example Code**

The following code sample shows how to set the Debugger built-in rules using the `Rule.sagemaker` method. To specify built-in rules that you want to run, use the `rules_configs` API operation to call the built-in rules. To find a full list of Debugger built-in rules and default parameter values, see List of Debugger Built-in Rules (p. 948).

```python
import sagemaker
from sagemaker.tensorflow import TensorFlow
from sagemaker.debugger import Rule, CollectionConfig, rule_configs

# call built-in rules that you want to use.
built_in_rules = [
    Rule.sagemaker(rule_configs.vanishing_gradient()),
    Rule.sagemaker(rule_configs.loss_not_decreasing())
]

# construct a SageMaker estimator with the Debugger built-in rules
sagemaker_estimator = TensorFlow(
    entry_point='directory/to/your_training_script.py',
    role=sm.get_execution_role(),
    base_job_name='debugger-built-in-rules-demo',
    inference_instance_count=1,
    rules=built_in_rules
)
```
train_instance_count=1,
train_instance_type="ml.m4.xlarge",
framework_version="2.1.0",
py_version="py3",

# debugger-specific arguments below
rules=rules
}
sagemaker_estimator.fit()

**Note**
The Debugger built-in rules run in parallel with your training job. The maximum number of built-in rule containers for a training job is 20.

For more information about the Debugger rule class, methods, and parameters, see the [SageMaker Debugger Rule class in the Amazon SageMaker Python SDK](#).

To find an example of how to adjust the Debugger rule parameters, see the following [Use Debugger Built-in Rules with Parameter Modifications](#) section.

**Use Debugger Built-in Rules with Parameter Modifications**

The following code example shows the structure of built-in rules to adjust parameters. In this example, the `stalled_training_rule` collects the `losses` tensor collection from a training job at every 50 steps and an evaluation stage at every 10 steps. If the training process starts stalling and not collecting tensor outputs for 120 seconds, the `stalled_training_rule` stops the training job.

```python
import sagemaker
from sagemaker.tensorflow import TensorFlow
from sagemaker.debugger import Rule, CollectionConfig, rule_configs

# call the built-in rules and modify the CollectionConfig parameters

base_job_name_prefix= 'smdebug-stalled-demo-' + str(int(time.time()))
built_in_rules_modified = [
    Rule.sagemaker(
        base_config=rule_configs.stalled_training_rule(),
        rule_parameters={
            'threshold': '120',
            'training_job_name_prefix': base_job_name_prefix,
            'stop_training_on_fire' : 'True'
        }
    )
    collections_to_save=[
        CollectionConfig(
            name="losses",
            parameters={
                "train.save_interval": "50",
                "eval.save_interval": "10"
            }
        )
    ]
]

# construct a SageMaker estimator with the modified Debugger built-in rule
sagemaker_estimator=TensorFlow(
    entry_point='directory/to/your_training_script.py',
    role=sm.get_execution_role(),
    base_job_name=base_job_name_prefix,
    train_instance_count=1,
    train_instance_type="ml.m4.xlarge",
    framework_version="2.1.0",
    py_version="py3",
)```
# debugger-specific arguments below
rules=built_in_rules_modified
)
sagemaker_estimator.fit()

For an advanced configuration of the Debugger built-in rules using the CreateTrainingJob API, see Add Debugger Built-in Rule Configuration to the CreateTrainingJob API Operation (p. 1016).

## Turn Off Debugger

If you want to completely turn off Debugger, do one of the following:

- Before starting a training job, do the following:

  To disable both monitoring and profiling, include the `disable_profiler` parameter to your estimator and set it to `True`.

    **Warning**
    
    If you disable it, you won't be able to view the comprehensive Studio Debugger insights dashboard and the autogenerated profiling report.

  To disable debugging, set the `debugger_hook_config` parameter to `False`.

    **Warning**
    
    If you disable it, you won't be able to collect output tensors and cannot debug your model parameters.

    ```python
    estimator=Estimator(
        ...
        disable_profiler=True
        debugger_hook_config=False
    )
    ```

    For more information about the Debugger-specific parameters, see SageMaker Estimator in the Amazon SageMaker Python SDK.

- While a training job is running, do the following:

  To disable both monitoring and profiling while your training job is running, use the following estimator classmethod:

    ```python
    estimator.disable_profiling()
    ```

  To disable framework profiling only and keep system monitoring, use the `update_profilier` method:

    ```python
    estimator.update_profilier(disable_framework_metrics=true)
    ```

    For more information about the estimator extension methods, see the `estimator.disable_profiling` and `estimator.update_profilier` classmethods in the Amazon SageMaker Python SDK documentation.

## Useful SageMaker Estimator Classmethods for Debugger

The following estimator class methods are useful for accessing your SageMaker training job information and retrieving output paths of training data collected by Debugger. The following methods are executable after you initiate a training job with the `estimator.fit()` method.
• To check the base S3 bucket URI of a SageMaker training job:

```python
estimator.output_path
```

• To check the base job name of a SageMaker training job:

```python
estimator.latest_training_job.job_name
```

• To see a full CreateTrainingJob API operation configuration of a SageMaker training job:

```python
estimator.latest_training_job.describe()
```

• To check a full list of the Debugger rules while a SageMaker training job is running:

```python
estimator.latest_training_job.rule_job_summary()
```

• To check the S3 bucket URI at where the model parameter data (output tensors) are saved:

```python
estimator.latest_job_debugger_artifacts_path()
```

• To check the S3 bucket URI at where the model performance data (system and framework metrics) are saved:

```python
estimator.latest_job_profiler_artifacts_path()
```

• To check the Debugger rule configuration for debugging output tensors:

```python
estimator.debugger_rule_configs
```

• To check the list of the Debugger rules for debugging while a SageMaker training job is running:

```python
estimator.debugger_rules
```

• To check the Debugger rule configuration for monitoring and profiling system and framework metrics:

```python
estimator.profiler_rule_configs
```

• To check the list of the Debugger rules for monitoring and profiling while a SageMaker training job is running:

```python
estimator.profiler_rules
```

For more information about the SageMaker estimator class and its methods, see Estimator API in the Amazon SageMaker Python SDK.

## List of Debugger Built-in Rules

Use the Debugger built-in rules provided by Amazon SageMaker Debugger and analyze tensors emitted while training your models. The Debugger built-in rules monitor various common conditions that are critical for the success of a training job. You can call the built-in rules using Amazon SageMaker Python SDK or the low-level SageMaker API operations. Depending on deep learning frameworks of your choice, there are four scopes of validity for the built-in rules as shown in the following table.

**Note**

The maximum numbers of built-in rules for a training job are 20 for `ProfilerRule` and 20 for `Rule`. SageMaker Debugger fully manages the built-in rules and analyzes your training job in
parallel. For more information about billing, see the Amazon SageMaker Studio is available at no additional charge section of the Amazon SageMaker Pricing page.

Important
To use the new Debugger features, you need to upgrade the SageMaker Python SDK and the SMDebug client library. In your iPython kernel, Jupyter notebook, or JupyterLab environment, run the following code to install the latest versions of the libraries and restart the kernel.

```python
import sys
import IPython
!{sys.executable} -m pip install -U sagemaker smdebug
IPython.Application.instance().kernel.do_shutdown(True)
```

### Debugger ProfilerRule

The following rules are the Debugger built-in rules that are callable using the ProfilerRule.sagemaker classmethod.

#### Debugger Built-in Rules for Generating Profiling Reports

<table>
<thead>
<tr>
<th>Scope of Validity</th>
<th>Built-in Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>Profiling Report for any SageMaker training job</td>
<td>• ProfilerReport (p. 951)</td>
</tr>
</tbody>
</table>

#### Debugger Built-in Rules for Monitoring Hardware System Resource Utilization (System Metrics)

<table>
<thead>
<tr>
<th>Scope of Validity</th>
<th>Built-in Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generic system monitoring rules for any SageMaker training job</td>
<td>• BatchSize (p. 952)</td>
</tr>
<tr>
<td></td>
<td>• CPUBottleneck (p. 953)</td>
</tr>
<tr>
<td></td>
<td>• GPUMemoryIncrease (p. 955)</td>
</tr>
<tr>
<td></td>
<td>• IOBottleneck (p. 956)</td>
</tr>
<tr>
<td></td>
<td>• LoadBalancing (p. 957)</td>
</tr>
<tr>
<td></td>
<td>• LowGPUUtilization (p. 957)</td>
</tr>
<tr>
<td></td>
<td>• OverallSystemUsage (p. 959)</td>
</tr>
</tbody>
</table>

#### Debugger Built-in Rules for Profiling Model Performance Data (Framework Metrics)

<table>
<thead>
<tr>
<th>Scope of Validity</th>
<th>Built-in Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>Profiling rules for deep learning frameworks (TensorFlow and PyTorch)</td>
<td>• MaxInitializationTime (p. 959)</td>
</tr>
<tr>
<td></td>
<td>• OverallFrameworkMetrics (p. 960)</td>
</tr>
<tr>
<td></td>
<td>• StepOutlier (p. 960)</td>
</tr>
</tbody>
</table>

### Debugger Rule

The following rules are the Debugger built-in rules that are callable using the Rule.sagemaker classmethod.
Debugger Built-in Rules for Generating Training Reports

<table>
<thead>
<tr>
<th>Scope of Validity</th>
<th>Built-in Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>Training Report for SageMaker XGboost training job</td>
<td>• create_xgboost_report (p. 961)</td>
</tr>
</tbody>
</table>

Debugger Built-in Rules for Debugging Model Training Data (Output Tensors)

<table>
<thead>
<tr>
<th>Scope of Validity</th>
<th>Built-in Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deep learning frameworks (TensorFlow, MXNet, and PyTorch)</td>
<td>• dead_relu (p. 962)</td>
</tr>
<tr>
<td></td>
<td>• exploding_tensor (p. 963)</td>
</tr>
<tr>
<td></td>
<td>• poor_weight_initialization (p. 964)</td>
</tr>
<tr>
<td></td>
<td>• saturated_activation (p. 966)</td>
</tr>
<tr>
<td></td>
<td>• vanishing_gradient (p. 969)</td>
</tr>
<tr>
<td></td>
<td>• weight_update_ratio (p. 970)</td>
</tr>
<tr>
<td>Deep learning frameworks (TensorFlow, MXNet, and PyTorch) and the XGBoost algorithm</td>
<td>• all_zero (p. 971)</td>
</tr>
<tr>
<td></td>
<td>• class_imbalance (p. 973)</td>
</tr>
<tr>
<td></td>
<td>• loss_not_decreasing (p. 975)</td>
</tr>
<tr>
<td></td>
<td>• overfit (p. 977)</td>
</tr>
<tr>
<td></td>
<td>• overtraining (p. 978)</td>
</tr>
<tr>
<td></td>
<td>• similar_across_runs (p. 980)</td>
</tr>
<tr>
<td></td>
<td>• stalled_training_rule (p. 981)</td>
</tr>
<tr>
<td></td>
<td>• tensor_variance (p. 982)</td>
</tr>
<tr>
<td></td>
<td>• unchanged_tensor (p. 984)</td>
</tr>
<tr>
<td>Deep learning applications</td>
<td>• check_input_images (p. 986)</td>
</tr>
<tr>
<td></td>
<td>• nlp_sequence_ratio (p. 987)</td>
</tr>
<tr>
<td>XGBoost algorithm</td>
<td>• confusion (p. 988)</td>
</tr>
<tr>
<td></td>
<td>• feature_importance_overweight (p. 991)</td>
</tr>
<tr>
<td></td>
<td>• tree_depth (p. 992)</td>
</tr>
</tbody>
</table>

To use the built-in rules with default parameter values – use the following configuration format:

```python
from sagemaker.debugger import Rule, ProfilerRule, rule_configs

rules = [  
    ProfilerRule.sagemaker(rule_configs.BuiltInRuleName_1()),  
    ProfilerRule.sagemaker(rule_configs.BuiltInRuleName_2()),  
    ...  
    ProfilerRule.sagemaker(rule_configs.BuiltInRuleName_n()),  
    Rule.sagemaker(rule_configs.built_in_rule_name_1()),  
    Rule.sagemaker(rule_configs.built_in_rule_name_2()),  
    ...  
    Rule.sagemaker(rule_configs.built_in_rule_name_n())
]
```

To use the built-in rules with customizing the parameter values – use the following configuration format:
from sagemaker.debugger import Rule, ProfilerRule, rule_configs

rules = [
    ProfilerRule.sagemaker(
        base_config=rule_configs.BuiltinRuleName(),
        rule_parameters={
            "key": "value"
        }
    ),
    Rule.sagemaker(
        base_config=rule_configs.built_in_rule_name(),
        rule_parameters={
            "key": "value"
        },
        collections_to_save=[
            CollectionConfig(
                name="tensor_collection_name",
                parameters={
                    "key": "value"
                }
            )
        ]
    ),
]

To find available keys for the rule_parameters parameter, see the parameter description tables.

Sample rule configuration codes are provided for each built-in rule below the parameter description tables.

- For a full instruction and examples of using the Debugger built-in rules, see Debugger Built-in Rules Example Code (p. 945).
- For a full instruction of using the built-in rules with the low-level SageMaker API operations, see Add Debugger Built-in Rule Configuration to the CreateTrainingJob API Operation (p. 1016).

**ProfilerReport**

The ProfilerReport rule invokes all of the built-in rules for monitoring and profiling. It creates a profiling report and updates when the individual rules are triggered. You can download a comprehensive profiling report while a training job is running or after the training job is complete. You can adjust the rule parameter values to customize sensitivity of the built-in monitoring and profiling rules. The following example code shows the basic format to adjust the built-in rule parameters through the ProfilerReport rule.

```python
rules=[
    ProfilerRule.sagemaker(
        rule_configs.ProfilerReport(<BuiltInRuleName>_<parameter_name> = value)
    )
]
```

If you trigger this ProfilerReport rule without any customized parameter as shown in the following example code, then the ProfilerReport rule triggers all of the built-in rules for monitoring and profiling with their default parameter values.

```python
rules=[ProfilerRule.sagemaker(rule_configs.ProfilerReport())]
```
The following example code shows how to specify and adjust the CPUBottleneck rule's `cpu_threshold` parameter and the IOBottleneck rule's `threshold` parameter.

```python
rules=[
    ProfilerRule.sagemaker(
        rule_configs.ProfilerReport(
            CPUBottleneck_cpu_threshold = 90,
            IOBottleneck_threshold = 90
        )
    )
]
```

### Parameter Descriptions for the OverallSystemUsage Rule

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>base_trial</td>
<td>The base trial training job name. This parameter is automatically set to the current training job by Amazon SageMaker Debugger.</td>
</tr>
<tr>
<td></td>
<td><strong>Required</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: String</td>
</tr>
<tr>
<td><code>&lt;BuiltInRuleName&gt;_&lt;parameter_name&gt;</code></td>
<td>Customizable parameter to adjust thresholds of other built-in monitoring and profiling rules.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Default value: None</td>
</tr>
</tbody>
</table>

### BatchSize

The BatchSize rule helps detect if GPU is underutilized due to a small batch size. To detect this issue, this rule monitors the average CPU utilization, GPU utilization, and GPU memory utilization. If utilization on CPU, GPU, and GPU memory is low on average, it may indicate that the training job can either run on a smaller instance type or can run with a bigger batch size. This analysis does not work for frameworks that heavily overallocate memory. However, increasing the batch size can lead to processing or data loading bottlenecks because more data preprocessing time is required in each iteration.

### Parameter Descriptions for the BatchSize Rule

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>base_trial</td>
<td>The base trial training job name. This parameter is automatically set to the current training job by Amazon SageMaker Debugger.</td>
</tr>
<tr>
<td></td>
<td><strong>Required</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: String</td>
</tr>
<tr>
<td>cpu_threshold_p95</td>
<td>Defines the threshold for 95th quantile of CPU utilization in percentage.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: Integer</td>
</tr>
<tr>
<td>Parameter Name</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td></td>
<td>Default value: 70 (in percentage)</td>
</tr>
<tr>
<td>gpu_threshold_p95</td>
<td>Defines the threshold for 95th quantile of GPU utilization in percentage.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: Integer</td>
</tr>
<tr>
<td></td>
<td>Default value: 70 (in percentage)</td>
</tr>
<tr>
<td>gpu_memory_threshold_p95</td>
<td>Defines the threshold for 95th quantile of GPU memory utilization in percentage.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: Integer</td>
</tr>
<tr>
<td></td>
<td>Default values: 70 (in percentage)</td>
</tr>
<tr>
<td>patience</td>
<td>Defines the number of data points to skip until the rule starts evaluation.</td>
</tr>
<tr>
<td></td>
<td>The first several steps of training jobs usually show high volume of data</td>
</tr>
<tr>
<td></td>
<td>processes, so keep the rule patient and prevent it from being invoked too</td>
</tr>
<tr>
<td></td>
<td>soon with a given number of profiling data that you specify with this</td>
</tr>
<tr>
<td></td>
<td>parameter.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: Integer</td>
</tr>
<tr>
<td></td>
<td>Default values: 100</td>
</tr>
<tr>
<td>window</td>
<td>Window size for computing quantiles.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: Integer</td>
</tr>
<tr>
<td></td>
<td>Default values: 500</td>
</tr>
<tr>
<td>scan_interval_us</td>
<td>Time interval that timeline files are scanned.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: Integer</td>
</tr>
<tr>
<td></td>
<td>Default values: 60000000 (in microseconds)</td>
</tr>
</tbody>
</table>

**CPUBottleneck**

The CPUBottleneck rule helps detect if GPU is underutilized due to CPU bottlenecks. Rule returns True if number of CPU bottlenecks exceeds a predefined threshold.
## Parameter Descriptions for the CPUBottleneck Rule

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>base_trial</td>
<td>The base trial training job name. This parameter is automatically set to the current training job by Amazon SageMaker Debugger.</td>
</tr>
<tr>
<td></td>
<td><strong>Required</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: String</td>
</tr>
<tr>
<td>threshold</td>
<td>Defines the threshold for proportion of bottlenecked time to the total training time. If the proportion exceeds the percentage specified to the threshold parameter, the rule switches the rule status to True.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: Integer</td>
</tr>
<tr>
<td></td>
<td>Default value: 50 (in percentage)</td>
</tr>
<tr>
<td>gpu_threshold</td>
<td>A threshold that defines low GPU utilization.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: Integer</td>
</tr>
<tr>
<td></td>
<td>Default value: 10 (in percentage)</td>
</tr>
<tr>
<td>cpu_threshold</td>
<td>A threshold that defines high CPU utilization.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: Integer</td>
</tr>
<tr>
<td></td>
<td>Default values: 90 (in percentage)</td>
</tr>
<tr>
<td>patience</td>
<td>Defines the number of data points to skip until the rule starts evaluation. The first several steps of training jobs usually show high volume of data processes, so keep the rule patient and prevent it from being invoked too soon with a given number of profiling data that you specify with this parameter.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: Integer</td>
</tr>
<tr>
<td></td>
<td>Default values: 100</td>
</tr>
<tr>
<td>scan_interval_us</td>
<td>Time interval with which timeline files are scanned.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: Integer</td>
</tr>
</tbody>
</table>
### Parameter Name  
### Description

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Default values:</strong></td>
<td>60000000 (in microseconds)</td>
</tr>
</tbody>
</table>

**GPUMemoryIncrease**

The GPUMemoryIncrease rule helps detect a large increase in memory usage on GPUs.

**Parameter Descriptions for the GPUMemoryIncrease Rule**

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>base_trial</strong></td>
<td>The base trial training job name. This parameter is automatically set to the current training job by Amazon SageMaker Debugger.</td>
</tr>
<tr>
<td><strong>Required</strong></td>
<td>Valid values: String</td>
</tr>
<tr>
<td><strong>increase</strong></td>
<td>Defines the threshold for absolute memory increase.</td>
</tr>
<tr>
<td><strong>Optional</strong></td>
<td>Valid values: Integer</td>
</tr>
<tr>
<td></td>
<td>Default value: 10 (in percentage)</td>
</tr>
<tr>
<td><strong>patience</strong></td>
<td>Defines the number of data points to skip until the rule starts evaluation. The first several steps of training jobs usually show high volume of data processes, so keep the rule patient and prevent it from being invoked too soon with a given number of profiling data that you specify with this parameter.</td>
</tr>
<tr>
<td><strong>Optional</strong></td>
<td>Valid values: Integer</td>
</tr>
<tr>
<td></td>
<td>Default values: 100</td>
</tr>
<tr>
<td><strong>window</strong></td>
<td>Window size for computing quantiles.</td>
</tr>
<tr>
<td><strong>Optional</strong></td>
<td>Valid values: Integer</td>
</tr>
<tr>
<td></td>
<td>Default values: 500</td>
</tr>
<tr>
<td><strong>scan_interval_us</strong></td>
<td>Time interval that timeline files are scanned.</td>
</tr>
<tr>
<td><strong>Optional</strong></td>
<td>Valid values: Integer</td>
</tr>
<tr>
<td></td>
<td>Default values: 60000000 (in microseconds)</td>
</tr>
</tbody>
</table>
**IOBottleneck**

This rule helps to detect if GPU is underutilized due to data IO bottlenecks. Rule returns True if number of IO bottlenecks exceeds a predefined threshold.

### Parameter Descriptions for the IOBottleneck Rule

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>base_trial</td>
<td>The base trial training job name. This parameter is automatically set to the current training job by Amazon SageMaker Debugger.</td>
</tr>
<tr>
<td></td>
<td><strong>Required</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: String</td>
</tr>
<tr>
<td>threshold</td>
<td>Defines the threshold when Rule to return True.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: Integer</td>
</tr>
<tr>
<td></td>
<td>Default value: 50 (in percentage)</td>
</tr>
<tr>
<td>gpu_threshold</td>
<td>A threshold that defines when GPU is considered underutilized.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: Integer</td>
</tr>
<tr>
<td></td>
<td>Default value: 70 (in percentage)</td>
</tr>
<tr>
<td>io_threshold</td>
<td>A threshold that defines high IO wait time.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: Integer</td>
</tr>
<tr>
<td></td>
<td>Default values: 50 (in percentage)</td>
</tr>
<tr>
<td>patience</td>
<td>Defines the number of data points to skip until the rule starts evaluation. The first several steps of training jobs usually show high volume of data processes, so keep the rule patient and prevent it from being invoked too soon with a given number of profiling data that you specify with this parameter.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: Integer</td>
</tr>
<tr>
<td></td>
<td>Default values: 1000</td>
</tr>
<tr>
<td>scan_interval_us</td>
<td>Time interval that timeline files are scanned.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: Integer</td>
</tr>
</tbody>
</table>
### LoadBalancing

The LoadBalancing rule helps detect issues in workload balancing among multiple GPUs.

#### Parameter Descriptions for the LoadBalancing Rule

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>base_trial</td>
<td>The base trial training job name. This parameter is automatically set to the current training job by Amazon SageMaker Debugger.</td>
</tr>
<tr>
<td>threshold</td>
<td>Defines the workload percentage.</td>
</tr>
<tr>
<td>patience</td>
<td>Defines the number of data points to skip until the rule starts evaluation. The first several steps of training jobs usually show high volume of data processes, so keep the rule patient and prevent it from being invoked too soon with a given number of profiling data that you specify with this parameter.</td>
</tr>
<tr>
<td>scan_interval_us</td>
<td>Time interval that timeline files are scanned.</td>
</tr>
</tbody>
</table>

- **base_trial**: Required
- **threshold**: Optional
- **patience**: Optional
- **scan_interval_us**: Optional

### LowGPUUtilization

The LowGPUUtilization rule helps detect if GPU utilization is low or suffers from fluctuations. This is checked for each GPU on each worker. Rule returns True if 95th quantile is below threshold_p95 which indicates underutilization. Rule returns true if 95th quantile is above threshold_p95 and 5th quantile is below threshold_p5 which indicates fluctuations.
### Parameter Descriptions for the LowGPUUtilization Rule

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
</table>
| base_trial     | The base trial training job name. This parameter is automatically set to the current training job by Amazon SageMaker Debugger.  
**Required**  
Valid values: String |
| threshold_p95  | A threshold for 95th quantile below which GPU is considered to be underutilized.  
**Optional**  
Valid values: Integer  
Default value: 70 (in percentage) |
| threshold_p5   | A threshold for 5th quantile. Default is 10 percent.  
**Optional**  
Valid values: Integer  
Default values: 10 (in percentage) |
| patience       | Defines the number of data points to skip until the rule starts evaluation. The first several steps of training jobs usually show high volume of data processes, so keep the rule patient and prevent it from being invoked too soon with a given number of profiling data that you specify with this parameter.  
**Optional**  
Valid values: Integer  
Default values: 1000 |
| window         | Window size for computing quantiles.  
**Optional**  
Valid values: Integer  
Default values: 500 |
| scan_interval_us | Time interval that timeline files are scanned.  
**Optional**  
Valid values: Integer  
Default values: 60000000 (in microseconds) |
OverallSystemUsage

The OverallSystemUsage rule measures overall system usage per worker node. The rule currently only aggregates values per node and computes their percentiles.

Parameter Descriptions for the OverallSystemUsage Rule

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>base_trial</td>
<td>The base trial training job name. This parameter is automatically set to the current training job by Amazon SageMaker Debugger.</td>
</tr>
<tr>
<td></td>
<td><strong>Required</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: String</td>
</tr>
<tr>
<td>scan_interval_us</td>
<td>Time interval to scan timeline files.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: Integer</td>
</tr>
<tr>
<td></td>
<td>Default values: 60000000 (in microseconds)</td>
</tr>
</tbody>
</table>

MaxInitializationTime

The MaxInitializationTime rule helps detect if the training initialization is taking too much time. The rule waits until the first step is available.

Parameter Descriptions for the MaxInitializationTime Rule

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>base_trial</td>
<td>The base trial training job name. This parameter is automatically set to the current training job by Amazon SageMaker Debugger.</td>
</tr>
<tr>
<td></td>
<td><strong>Required</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: String</td>
</tr>
<tr>
<td>threshold</td>
<td>Defines the threshold in minutes to wait for the first step to become available.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: Integer</td>
</tr>
<tr>
<td></td>
<td>Default value: 20 (in minutes)</td>
</tr>
<tr>
<td>scan_interval_us</td>
<td>Time interval with which timeline files are scanned.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: Integer</td>
</tr>
<tr>
<td></td>
<td>Default values: 60000000 (in microseconds)</td>
</tr>
</tbody>
</table>
OverallFrameworkMetrics

The OverallFrameworkMetrics rule summarizes the time spent on framework metrics, such as forward and backward pass, and data loading.

Parameter Descriptions for the OverallFrameworkMetrics Rule

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>base_trial</td>
<td>The base trial training job name. This parameter is automatically set to the current training job by Amazon SageMaker Debugger.</td>
</tr>
<tr>
<td></td>
<td><strong>Required</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: String</td>
</tr>
<tr>
<td>scan_interval_us</td>
<td>Time interval to scan timeline files.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: Integer</td>
</tr>
<tr>
<td></td>
<td>Default values: 60000000 (in microseconds)</td>
</tr>
</tbody>
</table>

StepOutlier

The StepOutlier rule helps detect outliers in step durations. This rule returns True if there are outliers with step durations larger than stddev sigmas of the entire step durations in a time range.

Parameter Descriptions for the StepOutlier Rule

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>base_trial</td>
<td>The base trial training job name. This parameter is automatically set to the current training job by Amazon SageMaker Debugger.</td>
</tr>
<tr>
<td></td>
<td><strong>Required</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: String</td>
</tr>
<tr>
<td>stddev</td>
<td>Defines a factor by which to multiply the standard deviation. For example, the rule is invoked by default when a step duration is larger or smaller than 5 times the standard deviation.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: Integer</td>
</tr>
<tr>
<td></td>
<td>Default value: 5 (in minutes)</td>
</tr>
<tr>
<td>mode</td>
<td>Mode under which steps have been saved and on which Rule should run on. Per default rule will run on steps from EVAL and TRAIN phase</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
</tbody>
</table>
### Parameter Name | Description
--- | ---
**n_outliers** | How many outliers to ignore before rule returns True  
  [Optional]  
  Valid values: Integer  
  Default value: 10

**scan_interval_us** | Time interval with which timeline files are scanned.  
  [Optional]  
  Valid values: Integer  
  Default values: 60000000 (in microseconds)

---

**CreateXgboostReport**

The CreateXgboostReport rule collects output tensors from an XGBoost training job and autogenerates a comprehensive training report. You can download a comprehensive profiling report while a training job is running or after the training job is complete, and check progress of training or the final result of the training job. The CreateXgboostReport rule collects the following output tensors by default:

- hyperparameters – Saves at the first step
- metrics – Saves loss and accuracy every 5 steps
- feature_importance – Saves every 5 steps
- predictions – Saves every 5 steps
- labels – Saves every 5 steps

---

**Parameter Descriptions for the StepOutlier Rule**

| Parameter Name | Description |
--- | ---
**base_trial** | The base trial training job name. This parameter is automatically set to the current training job by Amazon SageMaker Debugger.  
  [Required]  
  Valid values: String

---

```
import sagemaker

# Create the training rules
rules=
    Rule.sagemaker(
        rule_configs.create_xgboost_report()
    )
```
DeadRelu

This rule detects when the percentage of rectified linear unit (ReLU) activation functions in a trial are considered dead because their activation activity has dropped below a threshold. If the percent of inactive ReLUs in a layer is greater than the threshold_layer value of inactive ReLUs, the rule returns True.

Parameter Descriptions for the DeadRelu Rule

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>base_trial</td>
<td>The base trial training job name. This parameter is automatically set to the current training job by Amazon SageMaker Debugger.</td>
</tr>
<tr>
<td></td>
<td><strong>Required</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: String</td>
</tr>
<tr>
<td>tensor_regex</td>
<td>A list of regex patterns used to restrict this comparison to specific scalar-valued tensors. The rule inspects only the tensors that match the</td>
</tr>
<tr>
<td></td>
<td>regex patterns specified in the list. If no patterns are passed, the rule compares all tensors gathered in the trials by default. Only scalar-valued tensors can be matched.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: List of strings or a comma-separated string</td>
</tr>
<tr>
<td></td>
<td>Default value: &quot;.*relu_output&quot;</td>
</tr>
<tr>
<td>threshold_inactivity</td>
<td>Defines a level of activity below which a ReLU is considered to be dead. A ReLU might be active in the beginning of a trial and then slowly die during the training process. If the ReLU is active less than the threshold_inactivity, it is considered to be dead.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: Float</td>
</tr>
<tr>
<td></td>
<td>Default values: 1.0 (in percentage)</td>
</tr>
<tr>
<td>threshold_layer</td>
<td>Returns True if the percentage of inactive ReLUs in a layer is greater than threshold_layer. Returns False if the percentage of inactive ReLUs in a layer is less than threshold_layer.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: Float</td>
</tr>
<tr>
<td></td>
<td>Default values: 50.0 (in percentage)</td>
</tr>
</tbody>
</table>
built_in_rules = [
    Rule.sagemaker(
        base_config=rule_configs.dead_relu(),
        rule_parameters={
            "tensor_regex": ".*relu_output|.ReLU_output",
            "threshold_inactivity": "1.0",
            "threshold_layer": "50.0"
        },
        collections_to_save=[
            CollectionConfig(
                name="custom_relu_collection",
                parameters={
                    "include_regex": ".*relu_output|.ReLU_output",
                    "save_interval": "500"
                }
            )
        ]
    )
]

For an example of how to configure and deploy a built-in rule, see Configure Debugger Built-in Rules (p. 941).

**Note**
This rule is not available for the XGBoost algorithm.

### ExplodingTensor

This rule detects whether the tensors emitted during training have non-finite values, either infinite or NaN (not a number). If a non-finite value is detected, the rule returns True.

#### Parameter Descriptions for the ExplodingTensor Rule

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>base_trial</td>
<td>The base trial training job name. This parameter is automatically set to the current training job by Amazon SageMaker Debugger. Required</td>
</tr>
<tr>
<td></td>
<td>Valid values: String</td>
</tr>
<tr>
<td>collection_names</td>
<td>The list of collection names whose tensors the rule inspects. Optional</td>
</tr>
<tr>
<td></td>
<td>Valid values: String</td>
</tr>
<tr>
<td></td>
<td>Default value: None</td>
</tr>
<tr>
<td>tensor_regex</td>
<td>A list of regex patterns used to restrict this comparison to specific scalar-valued tensors. The rule inspects only the tensors that match the regex patterns specified in the list. If no patterns are passed, the rule compares all tensors gathered in the trials by default. Only scalar-valued tensors can be matched. Optional</td>
</tr>
</tbody>
</table>

963
### List of Built-in Rules

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>only_nan</td>
<td>True to monitor the base_trial tensors only for NaN values and not for infinity. False to treat both NaN and infinity as exploding values and to monitor for both. Optional Default value: False</td>
</tr>
</tbody>
</table>

```python
built_in_rules = [
    Rule.sagemaker(
        base_config=rule_configs.exploding_tensor(),
        rule_parameters={
            "tensor_regex": ".*gradient",
            "only_nan": "False"
        },
        collections_to_save=[
            CollectionConfig(
                name="gradients",
                parameters={
                    "save_interval": "500"
                }
            )
        ]
    )
]
```

For an example of how to configure and deploy a built-in rule, see [Configure Debugger Built-in Rules](p. 941).

**Note**

This rule is not available for the XGBoost algorithm.

### PoorWeightInitialization

This rule detects if your model parameters have been poorly initialized.

Good initialization breaks the symmetry of the weights and gradients in a neural network and maintains commensurate activation variances across layers. Otherwise, the neural network doesn't learn effectively. Initializers like Xavier aim to keep variance constant across activations, which is especially relevant for training very deep neural nets. Too small an initialization can lead to vanishing gradients. Too large an initialization can lead to exploding gradients. This rule checks the variance of activation inputs across layers, the distribution of gradients, and the loss convergence for the initial steps to determine if a neural network has been poorly initialized.

**Parameter Descriptions for the PoorWeightInitialization Rule**

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>base_trial</td>
<td>The base trial training job name. This parameter is automatically set to the current training job by Amazon SageMaker Debugger.</td>
</tr>
<tr>
<td>Parameter Name</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>activation_inputs_regex</td>
<td>A list of regex patterns used to restrict this comparison to specific scalar-valued tensors. The rule inspects only the tensors that match the regex patterns specified in the list. If no patterns are passed, the rule compares all tensors gathered in the trials by default. Only scalar-valued tensors can be matched.</td>
</tr>
<tr>
<td>threshold</td>
<td>If the ratio between minimum and maximum variance of weights per layer exceeds the threshold at a step, the rule returns True.</td>
</tr>
<tr>
<td>distribution_range</td>
<td>If the minimum difference between 5th and 95th percentiles of the gradient distribution is less than the distribution_range, the rule returns True.</td>
</tr>
<tr>
<td>patience</td>
<td>The number of steps to wait until the loss is considered to be no longer decreasing.</td>
</tr>
<tr>
<td>steps</td>
<td>The number of steps this rule analyzes. You typically need to check only the first few iterations.</td>
</tr>
</tbody>
</table>
SaturatedActivation

This rule detects if the tanh and sigmoid activation layers are becoming saturated. An activation layer is saturated when the input of the layer is close to the maximum or minimum of the activation function. The minimum and maximum of the tanh and sigmoid activation functions are defined by their respective min_threshold and max_thresholds values. If the activity of a node drops below the threshold_inactivity percentage, it is considered saturated. If more than a threshold_layer percent of the nodes are saturated, the rule returns True.

Parameter Descriptions for the SaturatedActivation Rule

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>base_trial</td>
<td>The base trial training job name. This parameter is automatically set to the current training job by Amazon SageMaker Debugger.</td>
</tr>
<tr>
<td></td>
<td><strong>Required</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: String</td>
</tr>
<tr>
<td>collection_names</td>
<td>The list of collection names whose tensors the rule inspects.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: List of strings or a comma-separated string</td>
</tr>
<tr>
<td></td>
<td>Default value: None</td>
</tr>
<tr>
<td>tensor_regex</td>
<td>A list of regex patterns used to restrict this comparison to specific scalar-valued tensors.</td>
</tr>
<tr>
<td>Parameter Name</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------</td>
<td>------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td></td>
<td>rule inspects only the tensors that match the regex patterns specified in the list. If no patterns are passed, the rule compares all tensors gathered in the trials by default. Only scalar-valued tensors can be matched.</td>
</tr>
</tbody>
</table>

**Optional**

Valid values: String

Default value: ".*tanh_input|.*sigmoid_input".

<table>
<thead>
<tr>
<th>threshold_tanh_min</th>
<th>The minimum and maximum thresholds that define the extremes of the input for a tanh activation function, defined as: (min_threshold, max_threshold). The default values are determined based on a vanishing gradient threshold of 0.0000001.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optional</td>
<td></td>
</tr>
<tr>
<td>Valid values: Float</td>
<td></td>
</tr>
<tr>
<td>Default values: -9.4999</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>threshold_tanh_max</th>
<th>The minimum and maximum thresholds that define the extremes of the input for a tanh activation function, defined as: (min_threshold, max_threshold). The default values are determined based on a vanishing gradient threshold of 0.0000001.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optional</td>
<td></td>
</tr>
<tr>
<td>Valid values: Float</td>
<td></td>
</tr>
<tr>
<td>Default values: 9.4999</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>threshold_sigmoid_min</th>
<th>The minimum and maximum thresholds that define the extremes of the input for a sigmoid activation function, defined as: (min_threshold, max_threshold). The default values are determined based on a vanishing gradient threshold of 0.0000001.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optional</td>
<td></td>
</tr>
<tr>
<td>Valid values: Float</td>
<td></td>
</tr>
<tr>
<td>Default values: -23</td>
<td></td>
</tr>
<tr>
<td>Parameter Name</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>threshold_sigmoid_max</td>
<td>The minimum and maximum thresholds that define the extremes of the input for a sigmoid activation function, defined as: $(\text{min_threshold}, \text{max_threshold})$. The default values are determined based on a vanishing gradient threshold of 0.0000001.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: Float</td>
</tr>
<tr>
<td></td>
<td>Default values: 16.99999</td>
</tr>
<tr>
<td>threshold_inactivity</td>
<td>The percentage of inactivity below which the activation layer is considered to be saturated. The activation might be active in the beginning of a trial and then slowly become less active during the training process.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: Float</td>
</tr>
<tr>
<td></td>
<td>Default values: 1.0</td>
</tr>
<tr>
<td>threshold_layer</td>
<td>Returns True if the number of saturated activations in a layer is greater than the threshold_layer percentage.</td>
</tr>
<tr>
<td></td>
<td>Returns False if the number of saturated activations in a layer is less than the threshold_layer percentage.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: Float</td>
</tr>
<tr>
<td></td>
<td>Default values: 50.0</td>
</tr>
</tbody>
</table>

```python
built_in_rules = [
    Rule.sagemaker(
        base_config=rule_configs.saturated_activation(),
        rule_parameters={
            "tensor_regex": ".*tanh_input|.*sigmoid_input",
            "threshold_tanh_min": "-9.4999",
            "threshold_tanh_max": "9.4999",
            "threshold_sigmoid_min": "-23",
            "threshold_sigmoid_max": "16.99999",
            "threshold_inactivity": "1.0",
            "threshold_layer": "50.0"
        },
        collections_to_save=[
            CollectionConfig(
                name="custom_activations_collection",
                parameters={
                    "include_regex": ".*tanh_input|.*sigmoid_input",
                    "save_interval": "500"
                }
            )
        ]
    ),
]```

968
For an example of how to configure and deploy a built-in rule, see Configure Debugger Built-in Rules (p. 941).

**Note**
This rule is not available for the XGBoost algorithm.

**VanishingGradient**

This rule detects if the gradients in a trial become extremely small or drop to a zero magnitude. If the mean of the absolute values of the gradients drops below a specified threshold, the rule returns True.

**Parameters Descriptions for the VanishingGradient Rule**

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>base_trial</td>
<td>The base trial training job name. This parameter is automatically set to the current training job by Amazon SageMaker Debugger.</td>
</tr>
<tr>
<td></td>
<td><strong>Required</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: String</td>
</tr>
<tr>
<td>threshold</td>
<td>The value at which the gradient is determined to be vanishing.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: Float</td>
</tr>
<tr>
<td></td>
<td>Default value: 0.0000001</td>
</tr>
</tbody>
</table>

```python
built_in_rules = [
    Rule.sagemaker(
        base_config=rule_configs.vanishing_gradient(),
        rule_parameters={
            "threshold": "0.0000001"
        },
        collections_to_save=[
            CollectionConfig(
                name="gradients",
                parameters={
                    "save_interval": "500"
                }
            )
        ]
    )
]```

For an example of how to configure and deploy a built-in rule, see Configure Debugger Built-in Rules (p. 941).

**Note**
This rule is not available for the XGBoost algorithm.
WeightUpdateRatio

This rule keeps track of the ratio of updates to weights during training and detects if that ratio gets too large or too small. If the ratio of updates to weights is larger than the `large_threshold` value or if this ratio is smaller than `small_threshold`, the rule returns True.

Conditions for training are best when the updates are commensurate to gradients. Excessively large updates can push the weights away from optimal values, and very small updates result in very slow convergence. This rule requires weights to be available for two training steps, and `train.save_interval` needs to be set equal to `num_steps`.

Parameter Descriptions for the WeightUpdateRatio Rule

<table>
<thead>
<tr>
<th>Parameter Name,</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>base_trial</td>
<td>The base trial training job name. This parameter is automatically set to the current training job by Amazon SageMaker Debugger.</td>
</tr>
<tr>
<td>Required</td>
<td>Valid values: String</td>
</tr>
<tr>
<td>num_steps</td>
<td>The number of steps across which the rule checks to determine if the tensor has changed. The number of steps across which you want to compare the weight ratios. If you pass no value, the rule runs by default against the current step and the immediately previous saved step. If you override the default by passing a value for this parameter, the comparison is done between weights at step $s$ and at a step $&gt;= s - num_steps$.</td>
</tr>
<tr>
<td>Optional</td>
<td>Valid values: Integer</td>
</tr>
<tr>
<td>large_threshold</td>
<td>The maximum value that the ratio of updates to weight can take before the rule returns True.</td>
</tr>
<tr>
<td>Optional</td>
<td>Valid values: Float</td>
</tr>
<tr>
<td>small_threshold</td>
<td>The minimum value that the ratio of updates to weight can take, below which the rule returns True.</td>
</tr>
<tr>
<td>Optional</td>
<td>Valid values: Float</td>
</tr>
</tbody>
</table>

Default value: None

Default value: 10.0

Default value: 0.00000001
### Parameter Name, Description

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>epsilon</td>
<td>A small constant used to ensure that Debugger does not divide by zero when computing the ratio updates to weigh.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: Float</td>
</tr>
<tr>
<td></td>
<td>Default value: 0.000000001</td>
</tr>
</tbody>
</table>

```python
built_in_rules = [
    Rule.sagemaker(
        base_config=rule_configs.weight_update_ratio(),
        rule_parameters={
            "num_steps": "100",
            "large_threshold": "10.0",
            "small_threshold": "0.00000001",
            "epsilon": "0.000000001"
        },
        collections_to_save=[
            CollectionConfig(
                name="weights",
                parameters={
                    "train.save_interval": "100"
                }
            )
        ]
    )
]
```

For an example of how to configure and deploy a built-in rule, see Configure Debugger Built-in Rules (p. 941).

**Note**

This rule is not available for the XGBoost algorithm.

### AllZero

This rule detects if all or a specified percentage of the tensor values are zero.

This rule can be applied either to one of the supported deep learning frameworks (TensorFlow, MXNet, and PyTorch) or to the XGBoost algorithm. You must specify either the `collection_names` or `tensor_regex` parameter. If both the parameters are specified, the rule inspects the union of tensors from both sets.

For an example of how to configure and deploy a built-in rule, see Configure Debugger Built-in Rules (p. 941).

### Parameters Descriptions for the AllZero Rule

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>base_trial</td>
<td>The base trial training job name. This parameter is automatically set to the current training job by Amazon SageMaker Debugger.</td>
</tr>
</tbody>
</table>

**Required**
<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Valid values: String</td>
</tr>
<tr>
<td>collection_names</td>
<td>The list of collection names whose tensors the rule inspects.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: List of strings or a comma-separated string</td>
</tr>
<tr>
<td></td>
<td>Default value: None</td>
</tr>
<tr>
<td>tensor_regex</td>
<td>A list of regex patterns used to restrict this comparison to specific scalar-valued tensors. The rule inspects only the tensors that match the regex patterns specified in the list. If no patterns are passed, the rule compares all tensors gathered in the trials by default. Only scalar-valued tensors can be matched.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: List of strings or a comma-separated string</td>
</tr>
<tr>
<td></td>
<td>Default value: None</td>
</tr>
<tr>
<td>threshold</td>
<td>Specifies the percentage of values in the tensor that needs to be zero for this rule to be invoked.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: Float</td>
</tr>
<tr>
<td></td>
<td>Default value: 100 (in percentage)</td>
</tr>
</tbody>
</table>

```python
built_in_rules = [
    Rule.sagemaker(
        base_config=rule_configs.all_zero(),
        rule_parameters={
            "tensor_regex": ".\*",
            "threshold": "100"
        },
        collections_to_save=[
            CollectionConfig(
                name="all",
                parameters={
                    "save_interval": "500"
                }
            )
        ]
    )
]
```
ClassImbalance

This rule measures sampling imbalances between classes and throws errors if the imbalance exceeds a threshold or if too many mispredictions for underrepresented classes occur as a result of the imbalance.

Classification models require well-balanced classes in the training dataset or a proper weighting/sampling of classes during training. The rule performs the following checks:

- It counts the occurrences per class. If the ratio of number of samples between smallest and largest class is larger than the \( \text{threshold}_\text{imbalance} \), an error is thrown.
- It checks the prediction accuracy per class. If resampling or weighting has not been correctly applied, then the model can reach high accuracy for the class with many training samples, but low accuracy for the classes with few training samples. If a fraction of mispredictions for a certain class is above \( \text{threshold}_\text{misprediction} \), an error is thrown.

This rule can be applied either to one of the supported deep learning frameworks (TensorFlow, MXNet, and PyTorch) or to the XGBoost algorithm.

For an example of how to configure and deploy a built-in rule, see Configure Debugger Built-in Rules (p. 941).

Parameter Descriptions for the ClassImbalance Rule

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>base_trial</td>
<td>The base trial training job name. This parameter is automatically set to the current training job by Amazon SageMaker Debugger.</td>
</tr>
<tr>
<td></td>
<td><strong>Required</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: String</td>
</tr>
<tr>
<td>threshold_imbalance</td>
<td>The acceptable imbalance between the number of samples in the smallest class and in the largest class. Exceeding this threshold value throws an error.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: Float</td>
</tr>
<tr>
<td></td>
<td>Default value: 10</td>
</tr>
<tr>
<td>threshold_misprediction</td>
<td>A limit on the fraction of mispredictions allowed for each class. Exceeding this threshold throws an error. The underrepresented classes are most at risk of crossing this threshold.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: Float</td>
</tr>
<tr>
<td></td>
<td>Default value: 0.7</td>
</tr>
<tr>
<td>samples</td>
<td>The number of labels that have to be processed before an imbalance is evaluated. The rule might not be triggered until it has seen sufficient samples across several steps. The more classes</td>
</tr>
<tr>
<td>Parameter Name</td>
<td>Description</td>
</tr>
<tr>
<td>---------------</td>
<td>-------------</td>
</tr>
</tbody>
</table>
| samples       | that your dataset contains, the larger this sample number should be.  
Optional  
Valid values: Integer  
Default value: 500 (assuming a dataset like MNIST with 10 classes) |
| argmax        | If True, np.argmax is applied to the prediction tensor. Required when you have a vector of probabilities for each class. It is used to determine which class has the highest probability.  
Conditional  
Valid values: Boolean  
Default value: False |
| labels_regex  | The name of the tensor that contains the labels.  
Optional  
Valid values: String  
Default value: ".*labels" |
| predictions_regex | The name of the tensor that contains the predictions.  
Optional  
Valid values: String  
Default value: ".*predictions" |

```python
built_in_rules = [
    Rule.sagemaker(  
        base_config=rule_configs.class_imbalance(),  
        rule_parameters={  
            "threshold_imbalance": "10",  
            "threshold_misprediction": "0.7",  
            "samples": "500",  
            "argmax": "False",  
            "labels_regex": ".*labels",  
            "predictions_regex": ".*predictions"  
        },  
        collections_to_save=[  
            CollectionConfig(  
                name="custom_output_collection",  
                parameters={  
                    "include_regex": ".*labels|.*predictions",  
                    "save_interval": "500"  
                }  
            )  
        ]  
    )
]"
```
**LossNotDecreasing**

This rule detects when the loss is not decreasing in value at an adequate rate. These losses must be scalars.

This rule can be applied either to one of the supported deep learning frameworks (TensorFlow, MXNet, and PyTorch) or to the XGBoost algorithm. You must specify either the `collection_names` or `tensor_regex` parameter. If both the parameters are specified, the rule inspects the union of tensors from both sets.

For an example of how to configure and deploy a built-in rule, see [Configure Debugger Built-in Rules (p. 941)](#).

**Parameter Descriptions for the LossNotDecreasing Rule**

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>base_trial</code></td>
<td>The base trial training job name. This parameter is automatically set to the current training job by Amazon SageMaker Debugger.</td>
</tr>
<tr>
<td></td>
<td><strong>Required</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: String</td>
</tr>
<tr>
<td><code>collection_names</code></td>
<td>The list of collection names whose tensors the rule inspects.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: List of strings or a comma-separated string</td>
</tr>
<tr>
<td></td>
<td>Default value: None</td>
</tr>
<tr>
<td><code>tensor_regex</code></td>
<td>A list of regex patterns that is used to restrict this comparison to specific scalar-valued tensors. The rule inspects only the tensors that match the regex patterns specified in the list. If no patterns are passed, the rule compares all tensors gathered in the trials by default. Only scalar-valued tensors can be matched.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: List of strings or a comma-separated string</td>
</tr>
<tr>
<td></td>
<td>Default value: None</td>
</tr>
<tr>
<td><code>use_losses_collection</code></td>
<td>If set to <code>True</code>, looks for losses in the collection named “losses” when the collection is present.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: Boolean</td>
</tr>
<tr>
<td></td>
<td>Default value: <code>True</code></td>
</tr>
<tr>
<td>Parameter Name</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>num_steps</td>
<td>The minimum number of steps after which the rule checks if the loss has decreased. Rule evaluation happens every num_steps. The rule compares the loss for this step with the loss at a step which is at least num_steps behind the current step. For example, suppose that the loss is being saved every three steps, but num_steps is set to 10. At step 21, loss for step 21 is compared with loss for step 9. The next step at which loss is checked is step 33, because ten steps after step 21 is step 31, and at step 31 and step 32 loss is not saved.</td>
</tr>
<tr>
<td>diff_percent</td>
<td>The minimum percentage difference by which the loss should decrease between num_steps.</td>
</tr>
<tr>
<td>increase_threshold_percent</td>
<td>The maximum threshold percent that loss is allowed to increase in case loss has been increasing</td>
</tr>
<tr>
<td>mode</td>
<td>The name of the Debugger mode to query tensor values for rule checking. If this is not passed, the rule checks in order by default for the mode.EVAL, then mode.TRAIN, and then mode.GLOBAL.</td>
</tr>
</tbody>
</table>

```
built_in_rules = [
    Rule.sagemaker(
        base_config=rule_configs.loss_not_decreasing(),
        rule_parameters={
            "tensor_regex": ".*",
            "use_losses_collection": "True",
            "num_steps": "10",
        }
    )
]```
“diff_percent”: "0.1",
"increase_threshold_percent": "5",
"mode": "GLOBAL"
},
collections_to_save=[
    CollectionConfig(
        name="losses",
        parameters={
            "save_interval": "500"
        }
    )
]
]

Overfit

This rule detects if your model is being overfit to the training data by comparing the validation and training losses.

This rule can be applied either to one of the supported deep learning frameworks (TensorFlow, MXNet, and PyTorch) or to the XGBoost algorithm.

For an example of how to configure and deploy a built-in rule, see Configure Debugger Built-in Rules (p. 941).

Note
A standard way to prevent overfitting is to regularize your model.

Parameter Descriptions for the Overfit Rule

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>base_trial</td>
<td>The base trial training job name. This parameter is automatically set to the current training job by Amazon SageMaker Debugger. Required</td>
</tr>
<tr>
<td>tensor_regex</td>
<td>A list of regex patterns used to restrict this comparison to specific scalar-valued tensors. The rule inspects only the tensors that match the regex patterns specified in the list. If no patterns are passed, the rule compares all tensors gathered in the trials by default. Only scalar-valued tensors can be matched. Optional</td>
</tr>
<tr>
<td>start_step</td>
<td>The step from which to start comparing the validation and training loss. Optional</td>
</tr>
<tr>
<td>Parameter Name</td>
<td>Description</td>
</tr>
<tr>
<td>----------------</td>
<td>-------------</td>
</tr>
<tr>
<td>patience</td>
<td>The number of steps for which the ratio_threshold is allowed to exceed the value set before the model is considered to be overfit. <strong>Optional</strong> Valid values: Integer Default value: 0</td>
</tr>
<tr>
<td>ratio_threshold</td>
<td>The maximum ratio of the difference between the mean validation loss and mean training loss to the mean training loss. If this threshold is exceeded for a patience number of steps, the model is being overfit and the rule returns True. <strong>Optional</strong> Valid values: Float Default value: 0.1</td>
</tr>
</tbody>
</table>

```python
built_in_rules = [
    Rule.sagemaker(
        base_config=rule_configs.overfit(),
        rule_parameters={
            "tensor_regex": ".*",
            "start_step": "0",
            "patience": "1",
            "ratio_threshold": "0.1"
        },
        collections_to_save=[
            CollectionConfig(
                name="losses",
                parameters={
                    "train.save_interval": "100",
                    "eval.save_interval": "10"
                })
        ]
    )
]
```

**Overtraining**

This rule detects if a model is being overtrained. After a number of training iterations on a well-behaved model (both training and validation loss decrease), the model approaches to a minimum of the loss function and does not improve anymore. If the model continues training it can happen that validation loss starts increasing, because the model starts overfitting. This rule sets up thresholds and conditions to determine if the model is not improving, and prevents overfitting problems due to overtraining.

This rule can be applied either to one of the supported deep learning frameworks (TensorFlow, MXNet, and PyTorch) or to the XGBoost algorithm.
For an example of how to configure and deploy a built-in rule, see Configure Debugger Built-in Rules (p. 941).

**Note**
Overtraining can be avoided by early stopping. For information on early stopping, see Stop Training Jobs Early (p. 1040). For an example that shows how to use spot training with Debugger, see Enable Spot Training with Amazon SageMaker Debugger.

### Parameter Descriptions for the Overtraining Rule

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>base_trial</td>
<td>The base trial training job name. This parameter is automatically set to the current training job by Amazon SageMaker Debugger.</td>
</tr>
<tr>
<td></td>
<td><strong>Required</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: String</td>
</tr>
<tr>
<td>patience_train</td>
<td>The number of steps to wait before the training loss is considered to not be improving anymore.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: Integer</td>
</tr>
<tr>
<td></td>
<td>Default value: 5</td>
</tr>
<tr>
<td>patience_validation</td>
<td>The number of steps to wait before the validation loss is considered to not be improving anymore.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: Integer</td>
</tr>
<tr>
<td></td>
<td>Default value: 10</td>
</tr>
<tr>
<td>delta</td>
<td>The minimum threshold by how much the error should improve before it is considered as a new optimum.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: Float</td>
</tr>
<tr>
<td></td>
<td>Default value: 0.01</td>
</tr>
</tbody>
</table>

```python
built_in_rules = [
    Rule.sagemaker(
        base_config=rule_configs.overtraining(),
        rule_parameters={
            "patience_train": "5",
            "patience_validation": "10",
            "delta": "0.01"
        },
        collections_to_save=[
            CollectionConfig(
                name="losses",
                parameters={
                    "save_interval": "500"
                }
            )
        ]
    )
]```

979
### SimilarAcrossRuns

This rule compares tensors gathered from a base trial with tensors from another trial.

This rule can be applied either to one of the supported deep learning frameworks (TensorFlow, MXNet, and PyTorch) or to the XGBoost algorithm.

For an example of how to configure and deploy a built-in rule, see [Configure Debugger Built-in Rules](p. 941).

#### Parameter Descriptions for the SimilarAcrossRuns Rule

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>base_trial</td>
<td>The base trial training job name. This parameter is automatically set to the current training job by Amazon SageMaker Debugger.</td>
</tr>
<tr>
<td></td>
<td><strong>Required</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: String</td>
</tr>
<tr>
<td>other_trial</td>
<td>A completed training job name whose tensors you want to compare to those tensors gathered from the current base_trial.</td>
</tr>
<tr>
<td></td>
<td><strong>Required</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: String</td>
</tr>
<tr>
<td>collection_names</td>
<td>The list of collection names whose tensors the rule inspects.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: List of strings or a comma-separated string</td>
</tr>
<tr>
<td></td>
<td>Default value: None</td>
</tr>
<tr>
<td>tensor_regex</td>
<td>A list of regex patterns used to restrict this comparison to specific scalar-valued tensors. The rule inspects only the tensors that match the regex patterns specified in the list. If no patterns are passed, the rule compares all tensors gathered in the trials by default. Only scalar-valued tensors can be matched.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: List of strings or a comma-separated string</td>
</tr>
<tr>
<td></td>
<td>Default value: None</td>
</tr>
</tbody>
</table>
StalledTrainingRule

StalledTrainingRule detects if there is no progress made on training job, and stops the training job if the rule fires. This rule requires tensors to be periodically saved in a time interval defined by its threshold parameter. This rule keeps on monitoring for new tensors, and if no new tensor has been emitted for threshold interval rule gets fired.

Parameter Descriptions for the StalledTrainingRule Rule

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>base_trial</td>
<td>The base trial training job name. This parameter is automatically set to the current training job by Amazon SageMaker Debugger.</td>
</tr>
<tr>
<td></td>
<td><strong>Required</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: String</td>
</tr>
<tr>
<td>threshold</td>
<td>A threshold that defines by how much time in seconds the rule waits for a tensor output until it fires a stalled training issue. Default value is 1800 seconds.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: Integer</td>
</tr>
<tr>
<td></td>
<td>Default value: 1800</td>
</tr>
<tr>
<td>stop_training_on_fire</td>
<td>If set to True, watches if the base training job outputs tensors in &quot;threshold&quot; seconds.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: Boolean</td>
</tr>
<tr>
<td></td>
<td>Default value: False</td>
</tr>
<tr>
<td>training_job_name_prefix</td>
<td>The prefix of base training job name. If stop_training_on_fire is true, the rule</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Parameter Descriptions for the Stalled Training Rule

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>base_trial</td>
<td>Searches for SageMaker training jobs with this prefix in the same account. If there is an inactivity found, the rule takes a <code>StopTrainingJob</code> action. Note if there are multiple jobs found with same prefix, the rule skips termination. It is important that the prefix is set unique per each training job. Optional Valid values: String</td>
</tr>
</tbody>
</table>

```python
built_in_rules = [
    Rule.sagemaker(
        base_config=rule_configs.stalled_training_rule(),
        rule_parameters={
            "threshold": "1800",
            "stop_training_on_fire": "True",
            "training_job_name_prefix": "<specify-training-base-job-name>"
        },
        collections_to_save=[
            CollectionConfig(
                name="losses",
                parameters={
                    "save_interval": "500"
                }
            )
        ]
    )
]`
```

## TensorVariance

This rule detects if you have tensors with very high or low variances. Very high or low variances in a tensor could lead to neuron saturation, which reduces the learning ability of the neural network. Very high variance in tensors can also eventually lead to exploding tensors. Use this rule to detect such issues early.

This rule can be applied either to one of the supported deep learning frameworks (TensorFlow, MXNet, and PyTorch) or to the XGBoost algorithm. You must specify either the `collection_names` or `tensor_regex` parameter. If both the parameters are specified, the rule inspects the union of tensors from both sets.

For an example of how to configure and deploy a built-in rule, see [Configure Debugger Built-in Rules](#).
<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
</table>
| collection_names | The list of collection names whose tensors the rule inspects.  
**Optional**  
Valid values: List of strings or a comma-separated string  
Default value: None |
| tensor_regex | A list of regex patterns used to restrict this comparison to specific scalar-valued tensors. The rule inspects only the tensors that match the regex patterns specified in the list. If no patterns are passed, the rule compares all tensors gathered in the trials by default. Only scalar-valued tensors can be matched.  
**Optional**  
Valid values: List of strings or a comma-separated string  
Default value: None |
| max_threshold | The threshold for the upper bound of tensor variance.  
**Optional**  
Valid values: Float  
Default value: None |
| min_threshold | The threshold for the lower bound of tensor variance.  
**Optional**  
Valid values: Float  
Default value: None |

```python
built_in_rules = [
    Rule.sagemaker(
        base_config=rule_configs.tensor_variance(),
        rule_parameters={
            "collection_names": "weights",
            "max_threshold": "10",
            "min_threshold": "0.00001",
        },
        collections_to_save=[
            CollectionConfig(
                name="weights",
                parameters={
                    "save_interval": "500"
                }
            )
        ]
    )
]```
UnchangedTensor

This rule detects whether a tensor is no longer changing across steps.

This rule runs the `numpy.allclose` method to check if the tensor isn't changing.

This rule can be applied either to one of the supported deep learning frameworks (TensorFlow, MXNet, and PyTorch) or to the XGBoost algorithm. You must specify either the `collection_names` or `tensor_regex` parameter. If both the parameters are specified, the rule inspects the union of tensors from both sets.

For an example of how to configure and deploy a built-in rule, see Configure Debugger Built-in Rules (p. 941).

**Parameter Descriptions for the UnchangedTensor Rule**

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>base_trial</td>
<td>The base trial training job name. This parameter is automatically set to the current training job by Amazon SageMaker Debugger.</td>
</tr>
<tr>
<td></td>
<td>Required</td>
</tr>
<tr>
<td></td>
<td>Valid values: String</td>
</tr>
<tr>
<td>collection_names</td>
<td>The list of collection names whose tensors the rule inspects.</td>
</tr>
<tr>
<td></td>
<td>Optional</td>
</tr>
<tr>
<td></td>
<td>Valid values: List of strings or a comma-separated string</td>
</tr>
<tr>
<td></td>
<td>Default value: None</td>
</tr>
<tr>
<td>tensor_regex</td>
<td>A list of regex patterns used to restrict this comparison to specific scalar-valued tensors. The rule inspects only the tensors that match the regex patterns specified in the list. If no patterns are passed, the rule compares all tensors gathered in the trials by default. Only scalar-valued tensors can be matched.</td>
</tr>
<tr>
<td></td>
<td>Optional</td>
</tr>
<tr>
<td></td>
<td>Valid values: List of strings or a comma-separated string</td>
</tr>
<tr>
<td></td>
<td>Default value: None</td>
</tr>
<tr>
<td>num_steps</td>
<td>The number of steps across which the rule checks to determine if the tensor has changed.</td>
</tr>
<tr>
<td></td>
<td>This checks the last <code>num_steps</code> that are available. They don't need to be consecutive. If <code>num_steps</code> is not specified, it defaults to 10.</td>
</tr>
<tr>
<td>Parameter Name</td>
<td>Description</td>
</tr>
<tr>
<td>----------------</td>
<td>-------------</td>
</tr>
<tr>
<td><code>num_steps</code></td>
<td>is 2, at step s it doesn’t necessarily check for s-1 and s. If s-1 isn’t available, it checks the last available step along with s. In that case, it checks the last available step with the current step.</td>
</tr>
<tr>
<td><code>rtol</code></td>
<td>The relative tolerance parameter to be passed to the <code>numpy.allclose</code> method.</td>
</tr>
<tr>
<td><code>atol</code></td>
<td>The absolute tolerance parameter to be passed to the <code>numpy.allclose</code> method.</td>
</tr>
<tr>
<td><code>equal_nan</code></td>
<td>Whether to compare NaNs as equal. If <code>True</code>, NaNs in input array a are considered equal to NaNs in input array b in the output array. This parameter is passed to the <code>numpy.allclose</code> method.</td>
</tr>
</tbody>
</table>

```python
built_in_rules = [
    Rule.sagemaker(
        base_config=rule_configs.unchanged_tensor(),
        rule_parameters={
            "collection_names": "losses",
            "tensor_regex": "",
            "num_steps": "3",
            "rtol": "1e-05",
            "atol": "1e-08",
            "equal_nan": "False"
        },
        collections_to_save=[
            CollectionConfig(
                name="losses",
                parameters={
                    "save_interval": "500"
                }
            )
        ]
    )
]```
CheckInputImages

This rule checks if input images have been correctly normalized. Specifically, it detects if the mean of the sample data differs by more than a threshold value from zero. Many computer vision models require that input data has a zero mean and unit variance.

This rule is applicable to deep learning applications.

For an example of how to configure and deploy a built-in rule, see Configure Debugger Built-in Rules (p. 941).

Parameter Descriptions for the CheckInputImages Rule

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>base_trial</td>
<td>The base trial training job name. This parameter is automatically set to the current training job by Amazon SageMaker Debugger.</td>
</tr>
<tr>
<td></td>
<td><strong>Required</strong></td>
</tr>
<tr>
<td>threshold_mean</td>
<td>A threshold that defines by how much mean of the input data can differ from 0.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td>threshold_samples</td>
<td>The number of images that have to be sampled before an error can be thrown. If the value is too low, the estimation of the dataset mean will be inaccurate.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td>regex</td>
<td>The name of the input data tensor.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td>channel</td>
<td>The position of the color channel in the input tensor shape array.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
</tbody>
</table>
List of Built-in Rules

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>base_trial</td>
<td>The base trial training job name. This parameter is automatically set to the current training job by Amazon SageMaker Debugger. Required</td>
</tr>
<tr>
<td>tensor_regex</td>
<td>A list of regex patterns used to restrict this comparison to specific scalar-valued tensors. The rule inspects only the tensors that match the regex patterns specified in the list. If no patterns are passed, the rule compares all tensors gathered</td>
</tr>
</tbody>
</table>

NLPSequenceRatio

This rule calculates the ratio of specific tokens given the rest of the input sequence that is useful for optimizing performance. For example, you can calculate the percentage of padding end-of-sentence (EOS) tokens in your input sequence. If the number of EOS tokens is too high, an alternate bucketing strategy should be performed. You also can calculate the percentage of unknown tokens in your input sequence. If the number of unknown words is too high, an alternate vocabulary could be used.

This rule is applicable to deep learning applications.

For an example of how to configure and deploy a built-in rule, see Configure Debugger Built-in Rules (p. 941).

Parameter Descriptions for the NLPSequenceRatio Rule

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>base_trial</td>
<td>The base trial training job name. This parameter is automatically set to the current training job by Amazon SageMaker Debugger. Required</td>
</tr>
<tr>
<td>tensor_regex</td>
<td>A list of regex patterns used to restrict this comparison to specific scalar-valued tensors. The rule inspects only the tensors that match the regex patterns specified in the list. If no patterns are passed, the rule compares all tensors gathered</td>
</tr>
<tr>
<td>Parameter Name</td>
<td>Description</td>
</tr>
<tr>
<td>----------------</td>
<td>-------------</td>
</tr>
<tr>
<td>tensor_regex</td>
<td>in the trials by default. Only scalar-valued tensors can be matched. <strong>Optional</strong> Valid values: List of strings or a comma-separated string Default value: &quot;.*embedding0_input_0&quot; (assuming an embedding as the initial layer of the network)</td>
</tr>
<tr>
<td>token_values</td>
<td>A string of a list of the numerical values of the tokens. For example, &quot;3, 0&quot;. <strong>Optional</strong> Valid values: Comma-separated string of numerical values Default value: 0</td>
</tr>
<tr>
<td>token_thresholds_percent</td>
<td>A string of a list of thresholds (in percentages) that correspond to each of the token_values. For example, &quot;50.0, 50.0&quot;. <strong>Optional</strong> Valid values: Comma-separated string of floats Default value: &quot;50&quot;</td>
</tr>
</tbody>
</table>

```python
built_in_rules = [
    Rule.sagemaker(
        base_config=rule_configs.nlp_sequence_ratio(),
        rule_parameters={
            "tensor_regex": ".*embedding0_input_0",
            "token_values": "0",
            "token_thresholds_percent": "50"
        },
        collections_to_save=[
            CollectionConfig(
                name="custom_inputs_collection",
                parameters={
                    "include_regex": ".*embedding0_input_0"
                }
            )
        ]
    )
]
```

**Confusion**

This rule evaluates the goodness of a confusion matrix for a classification problem.

It creates a matrix of size **category_no** x **category_no** and populates it with data coming from (labels, predictions) pairs. For each (labels, predictions) pair, the count in
confusion[labels][predictions] is incremented by 1. When the matrix is fully populated, the ratio of data on-diagonal values and off-diagonal values are evaluated as follows:

- For elements on the diagonal: \( \frac{\text{confusion}[i][i]}{\text{sum}_j(\text{confusion}[j][j])} \geq \text{min}_\text{diag} \)
- For elements off the diagonal: \( \frac{\text{confusion}[j][i]}{\text{sum}_j(\text{confusion}[j][i])} \leq \text{max}_\text{off}_\text{diag} \)

This rule can be applied to the XGBoost algorithm.

For an example of how to configure and deploy a built-in rule, see Configure Debugger Built-in Rules (p. 941).

**Parameter Descriptions for the Confusion Rule**

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>base_trial</td>
<td>The base trial training job name. This parameter is automatically set to the current training job by Amazon SageMaker Debugger.</td>
</tr>
<tr>
<td></td>
<td><strong>Required</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: String</td>
</tr>
<tr>
<td>category_no</td>
<td>The number of categories.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: Integer ( \geq 2 )</td>
</tr>
<tr>
<td></td>
<td>Default value: &quot;None&quot;</td>
</tr>
<tr>
<td>labels</td>
<td>The labels tensor collection or an 1-d vector of true labels.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: String</td>
</tr>
<tr>
<td></td>
<td>Default value: &quot;labels&quot;</td>
</tr>
<tr>
<td>predictions</td>
<td>The predictions tensor collection or an 1-d vector of estimated labels.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: String</td>
</tr>
<tr>
<td></td>
<td>Default value: &quot;predictions&quot;</td>
</tr>
<tr>
<td>labels_collection</td>
<td>The rule inspects the tensors in this collection for labels.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: String</td>
</tr>
<tr>
<td></td>
<td>Default value: &quot;labels&quot;</td>
</tr>
<tr>
<td>predictions_collection</td>
<td>The rule inspects the tensors in this collection for predictions.</td>
</tr>
<tr>
<td></td>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td></td>
<td>Valid values: String</td>
</tr>
<tr>
<td></td>
<td>Default value: &quot;predictions&quot;</td>
</tr>
<tr>
<td>Parameter Name</td>
<td>Description</td>
</tr>
<tr>
<td>----------------</td>
<td>-------------</td>
</tr>
<tr>
<td><strong>Optional</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Valid values:</strong> String</td>
<td></td>
</tr>
<tr>
<td><strong>Default value:</strong> &quot;predictions&quot;</td>
<td></td>
</tr>
<tr>
<td>min_diag</td>
<td>The minimum threshold for the ratio of data on the diagonal.</td>
</tr>
<tr>
<td><strong>Optional</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Valid values:</strong> 0 ≤ float ≤ 1</td>
<td></td>
</tr>
<tr>
<td><strong>Default value:</strong> 0.9</td>
<td></td>
</tr>
<tr>
<td>max_off_diag</td>
<td>The maximum threshold for the ratio of data off the diagonal.</td>
</tr>
<tr>
<td><strong>Optional</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Valid values:</strong> 0 ≤ float ≤ 1</td>
<td></td>
</tr>
<tr>
<td><strong>Default value:</strong> 0.1</td>
<td></td>
</tr>
</tbody>
</table>

```python
built_in_rules = [
    Rule.sagemaker(
        base_config=rule_configs.confusion(),
        rule_parameters={
            "category_no": "10",
            "labels": "labels",
            "predictions": "predictions",
            "labels_collection": "labels",
            "predictions_collection": "predictions",
            "min_diag": "0.9",
            "max_off_diag": "0.1"
        },
        collections_to_save=[
            CollectionConfig(
                name="labels",
                parameters={
                    "save_interval": "500"
                }
            ),
            CollectionConfig(
                name="predictions",
                parameters={
                    "include_regex": "500"
                }
            )
        ]
    )
]
```

**Note**

This rule infers default values for the optional parameters if their values aren't specified.
FeatureImportanceOverweight

This rule accumulates the weights of the \( n \) largest feature importance values per step and ensures that they do not exceed the threshold. For example, you can set the threshold for the top 3 features to not hold more than 80 percent of the total weights of the model.

This rule is valid only for the XGBoost algorithm.

For an example of how to configure and deploy a built-in rule, see Configure Debugger Built-in Rules (p. 941).

Parameter Descriptions for the FeatureImportanceOverweight Rule

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>base_trial</td>
<td>The base trial training job name. This parameter is automatically set to the current training job by Amazon SageMaker Debugger. Required</td>
</tr>
<tr>
<td></td>
<td>Valid values: String</td>
</tr>
<tr>
<td>threshold</td>
<td>Defines the threshold for the proportion of the cumulative sum of the ( n ) largest features. The number ( n ) is defined by the nfeatures parameter. Optional</td>
</tr>
<tr>
<td></td>
<td>Valid values: Float</td>
</tr>
<tr>
<td></td>
<td>Default value: 0.8</td>
</tr>
<tr>
<td>nfeatures</td>
<td>The number of largest features. Optional</td>
</tr>
<tr>
<td></td>
<td>Valid values: Integer</td>
</tr>
<tr>
<td></td>
<td>Default value: 3</td>
</tr>
<tr>
<td>tensor_regex</td>
<td>Regular expression (regex) of tensor names the rule to analyze. Optional</td>
</tr>
<tr>
<td></td>
<td>Valid values: String</td>
</tr>
<tr>
<td></td>
<td>Default value: &quot;.*feature_importance/weight&quot;</td>
</tr>
</tbody>
</table>

```python
built_in_rules = [
    Rule.sagemaker(
        base_config=rule_configs.feature_importance_overweight(),
        rule_parameters=
        {
            "threshold": "0.8",
            "nfeatures": "3",
            "tensor_regex": ".*feature_importance/weight"
        }
    )
```
TreeDepth

This rule measures the depth of trees in an XGBoost model. XGBoost rejects splits if they do not improve loss. This regularizes the training. As a result, the tree might not grow as deep as defined by the depth parameter.

This rule is valid only for the XGBoost algorithm.

For an example of how to configure and deploy a built-in rule, see Configure Debugger Built-in Rules (p. 941).

Parameter Descriptions for the TreeDepth Rule

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>base_trial</td>
<td>The base trial training job name. This parameter is automatically set to the current training job by Amazon SageMaker Debugger. Required</td>
</tr>
<tr>
<td>depth</td>
<td>The depth of the tree. The depth of the tree is obtained by computing the base 2 logarithm of the largest node ID. Optional</td>
</tr>
</tbody>
</table>

Default value: 4
Create Debugger Custom Rules for Training Job Analysis

You can create custom rules to monitor your training job using the Debugger Rule APIs and the open source smdebug Python library that provide tools to build your own rule containers.

Topics
- Prerequisites for Creating Debugger Custom Rules (p. 993)
- Use the Debugger Client Library smdebug to Create a Custom Rule Python Script (p. 993)
- Use the Debugger APIs to Run Your Own Custom Rules (p. 993)

Prerequisites for Creating Debugger Custom Rules

To create Debugger custom rules, you need the following prerequisites.

- SageMaker Debugger Rule.custom API
- The open source smdebug Python library
- Your own custom rule python script
- Amazon SageMaker Debugger Registry URLs for Custom Rule Evaluators (p. 1021)

Use the Debugger Client Library smdebug to Create a Custom Rule Python Script

The smdebug Rule API provides an interface to set up your own custom rules. The following python script is a sample of how to construct a custom rule, CustomGradientRule. This tutorial custom rule watches if the gradients are getting too large and set the default threshold as 10. The custom rule takes a base trial created by a SageMaker estimator when it initiates training job.

```python
from smdebug.rules.rule import Rule

class CustomGradientRule(Rule):
    def __init__(self, base_trial, threshold=10.0):
        super().__init__(base_trial)
        self.threshold = float(threshold)

    def invoke_at_step(self, step):
        for tname in self.base_trial.tensor_names(collection="gradients"):
            t = self.base_trial.tensor(tname)
            abs_mean = t.reduction_value(step, "mean", abs=True)
            if abs_mean > self.threshold:
                return True
        return False
```

You can add multiple custom rule classes as many as you want in the same python script and deploy them to any training job trials by constructing custom rule objects in the following section.

Use the Debugger APIs to Run Your Own Custom Rules

The following code sample shows how to configure a custom rule with the Amazon SageMaker Python SDK. This example assumes that the custom rule script you created in the previous step is located at 'path/to/my_custom_rule.py'.

from sagemaker.debugger import Rule, CollectionConfig

custom_rule = Rule.custom(
    name='MyCustomRule',
    image_uri='759209512951.dkr.ecr.us-west-2.amazonaws.com/sagemaker-debugger-rule-evaluator:latest',
    instance_type='ml.t3.medium',
    source='path/to/my_custom_rule.py',
    rule_to_invoke='CustomGradientRule',
    collections_to_save=[CollectionConfig("gradients")],
    rule_parameters={"threshold": "20.0"}
)

The following list explains the Debugger Rule.custom API arguments.

- **name** (str): Specify a custom rule name as you want.
- **image_uri** (str): This is the image of the container that has the logic of understanding your custom rule. It sources and evaluates the specified tensor collections you save in the training job. You can find the list of open source SageMaker rule evaluator images from Amazon SageMaker Debugger Registry URLs for Custom Rule Evaluators (p. 1021).
- **instance_type** (str): You need to specify an instance to build a rule docker container. This spins up the instance in parallel with a training container.
- **source** (str): This is the local path or the Amazon S3 URI to your custom rule script.
- **rule_to_invoke** (str): This specifies the particular Rule class implementation in your custom rule script. SageMaker supports only one rule to be evaluated at a time in a rule job.
- **collections_to_save** (str): This specifies which tensor collections you will save for the rule to run.
- **rule_parameters** (dictionary): This accepts parameter inputs in a dictionary format. You can adjust the parameters that you configured in the custom rule script.

After you set up the custom_rule object, you can use it for building a SageMaker estimator for any training jobs. Specify the entry_point to your training script. You do not need to make any change of your training script.

from sagemaker.tensorflow import TensorFlow

estimator = TensorFlow(
    role=sagemaker.get_execution_role(),
    base_job_name='smdebug-custom-rule-demo-tf-keras',
    entry_point='path/to/your_training_script.py'
    train_instance_type='ml.p2.xlarge',
    ...  
    
    rules = [custom_rule]
)

estimator.fit()

For more variations and advanced examples of using Debugger custom rules, see the following example notebooks.

- Monitor your training job with Amazon SageMaker Debugger custom rules
- PyTorch iterative model pruning of ResNet and AlexNet
- Trigger Amazon CloudWatch Events using Debugger Rules to Take an Action Based on Training Status with TensorFlow
Use Debugger with Custom Training Containers

Amazon SageMaker Debugger is available for any deep learning models that you bring to Amazon SageMaker. The AWS CLI, SageMaker Estimator API, and the Debugger APIs enable you to use any Docker base images to build and customize containers to train your models. To use Debugger with customized containers, you need to make a minimal change to your training script to implement the Debugger hook callback and retrieve tensors from training jobs.

You need the following resources to build a customized container with Debugger.

- Amazon SageMaker Python SDK
- The SMDebug open source client library
- A Docker base image of your choice
- Your training script with a Debugger hook registered – For more information about registering a Debugger hook to your training script, see Register Debugger Hook to Your Training Script (p. 995).

For an end-to-end example of using Debugger with a custom training container, see the following example notebook.

- Build a Custom Training Container and Debug Training Jobs with Debugger

Tip
This custom container with Debugger guide is an extension of the Adapting Your Own Training Container (p. 1373) guide which walks you thorough how to build and push your custom training container to Amazon ECR.

Prepare to Build a Custom Training Container

To build a docker container, the basic structure of files should look like the following:

| ### debugger_custom_container_test_notebook.ipynb | # a notebook to run python snippet codes |
| ### debugger_custom_container_test_folder | # this is a docker folder |
| ### your-training-script.py | # your training script with Debugger hook |
| ### Dockerfile | # a Dockerfile to build your own container |

Register Debugger Hook to Your Training Script

To debug your model training, you need to add a Debugger hook to your training script.

Note
This step is required to collect model parameters (output tensors) for debugging your model training. If you only want to monitor and profile, you can skip this hook registration step and exclude the debugger_hook_config parameter when constructing an estimator.

The following example code shows the structure of a training script using the Keras ResNet50 model and how to pass the Debugger hook as a Keras callback for debugging. To find a complete training script, see TensorFlow training script with SageMaker Debugger hook.

```python
# An example of training script (your-training-script.py)
import tensorflow.compat.v2 as tf
from tensorflow.keras.applications.resnet50 import ResNet50
import smdebug.tensorflow as smd

def train(batch_size, epoch, model, hook):
```
... model.fit(X_train, Y_train, batch_size=batch_size, epochs=epoch, validation_data=(X_valid, Y_valid), shuffle=True,

    # smdebug modification: Pass the Debugger hook in the main() as a Keras callback
    callbacks=[hook])

def main():
    parser=argparse.ArgumentParser(description="Train resnet50 cifar10")

    # hyperparameter settings
    parser.add_argument(...)

    args = parser.parse_args()

    model=ResNet50(weights=None, input_shape=(32,32,3), classes=10)

    # Add the following line to register the Debugger hook for Keras.
    hook=smd.KerasHook.create_from_json_file()

    # Start the training.
    train(args.batch_size, args.epoch, model, hook)

if __name__ == "__main__":
    main()

For more information about registering the Debugger hook for the supported frameworks and algorithm, see the following links in the SMDebug client library:

- SMDebug TensorFlow hook
- SMDebug PyTorch hook
- SMDebug MXNet hook
- SMDebug XGBoost hook

In the following example notebooks' training scripts, you can find more examples about how to add the Debugger hooks to training scripts and collect output tensors in detail:

- Debugger in script mode with the TensorFlow 2.1 framework

  To see the difference between using Debugger in a Deep Learning Container and in script mode, open this notebook and put it and the previous Debugger in a Deep Learning Container TensorFlow v2.1 notebook example side by side.

  In script mode, the hook configuration part is removed from the script in which you set the estimator. Instead, the Debugger hook feature is merged into the training script, TensorFlow Keras ResNet training script in script mode. The training script imports the smdebug library in the required TensorFlow Keras environment to communicate with the TensorFlow ResNet50 algorithm. It also manually implements the smdebug hook functionality by adding the callbacks=[hook] argument inside the train function (in line 49), and by adding the manual hook configuration (in line 89) provided through SageMaker Python SDK.

  This script mode example runs the training job in the TF 2.1 framework for direct comparison with the zero script change in the TF 2.1 example. The benefit of setting up Debugger in script mode is the flexibility to choose framework versions not covered by AWS Deep Learning Containers.
• Using Amazon SageMaker Debugger in a PyTorch Container in Script Mode

This notebook enables Debugger in script mode in PyTorch v1.3.1 framework. PyTorch v1.3.1 is supported by SageMaker containers, and this example shows details of how to modify a training script.

The SageMaker PyTorch estimator is already in script mode by default. In the notebook, the line to activate `script_mode` is not included in the estimator configuration.

This notebook shows detailed steps to change an original PyTorch training script to a modified version with Debugger enabled. Additionally, this example shows how you can use Debugger built-in rules to detect training issues such as the vanishing gradients problem, and the Debugger trial features to call and analyze the saved tensors.

Create and Configure a Dockerfile

Open your SageMaker JupyterLab and create a new folder, `debugger_custom_container_test_folder` in this example, to save your training script and Dockerfile. The following code example is a Dockerfile that includes essential docker build commends. Paste the following code into the Dockerfile text file and save it. Upload your training script to the same folder.

```
# Specify a docker base image
FROM tensorflow/tensorflow:2.2.0rc2-gpu-py3
RUN /usr/bin/python3 -m pip install --upgrade pip
RUN pip install --upgrade protobuf

# Install required packages to enable the SageMaker Python SDK and the smdebug library
RUN pip install sagemaker-training
RUN pip install smdebug

CMD ["bin/bash"]
```

If you want to use a pre-built AWS Deep Learning Container image, see Available AWS Deep Learning Containers Images.

Build and Push the Custom Training Container to Amazon ECR

Create a test notebook, `debugger_custom_container_test_notebook.ipynb`, and run the following code in the notebook cell. This will access the `debugger_byoc_test_docker` directory, build the docker with the specified `algorithm_name`, and push the docker container to your Amazon ECR.

```
import boto3
account_id = boto3.client('sts').get_caller_identity().get('Account')
ecr_repository = 'sagemaker-debugger-mnist-byoc-tf2'
tag = ':latest'
region = boto3.session.Session().region_name
uri_suffix = 'amazonaws.com'
if region in ['cn-north-1', 'cn-northwest-1']:
    uri_suffix = 'amazonaws.com.cn'
byoc_image_uri = '{}.dkr.ecr.{}.amazonaws.com/{}'.format(account_id, region, uri_suffix, ecr_repository + tag)

!docker build -t $ecr_repository docker
!$(aws ecr get-login --region $region --registry-ids $account_id --no-include-email)
!aws ecr create-repository --repository-name $ecr_repository
!docker tag {ecr_repository + tag} $byoc_image_uri
```
Run and Debug Training Jobs Using the Custom Training Container

After you build and push your docker container to Amazon ECR, configure a SageMaker estimator with your training script and the Debugger-specific parameters. After you execute the `estimator.fit()`, Debugger will collect output tensors, monitor them, and detect training issues. Using the saved tensors, you can further analyze the training job by using the `smdebug` core features and tools. Configuring a workflow of Debugger rule monitoring process with Amazon CloudWatch Events and AWS Lambda, you can automate a stopping training job process whenever the Debugger rules spots training issues.

```python
import sagemaker
from sagemaker.estimator import Estimator
from sagemaker.debugger import Rule, DebuggerHookConfig, CollectionConfig, rule_configs

profiler_config = ProfilerConfig(...)  # profiler config
debugger_hook_config = DebuggerHookConfig(...)  # debugger hook config
rules = [  # rules config
    Rule.sagemaker(rule_configs.built_in_rule()),
    ProfilerRule.sagemaker(rule_configs.BuiltInRule())
]

estimator = Estimator(  # estimator config
    image_uri=byoc_image_uri,
    entry_point="./debugger_custom_container_test_folder/your-training-script.py",
    role=sagemaker.get_execution_role(),
    base_job_name='debugger-custom-container-test',
    instance_count=1,
    instance_type='ml.p3.2xlarge',

    # Debugger-specific parameters
    profiler_config=profiler_config,
    debugger_hook_config=debugger_hook_config,
    rules=rules
)

# start training
estimator.fit()
```

Action on Amazon SageMaker Debugger Rules

Based on the Debugger rule evaluation status, you can set up automated actions such as stopping a training job and sending notifications using Amazon Simple Notification Service (Amazon SNS). You can also create your own actions using Amazon CloudWatch Events and AWS Lambda. To learn how to set up automated actions based on the Debugger rule evaluation status, see the following topics.

Topics
- [Debugger Built-in Actions for Rules (p. 999)]
- [Create Actions on Rules Using Amazon CloudWatch and AWS Lambda (p. 1002)]
Debugger Built-in Actions for Rules

Use Debugger built-in actions to respond to issues found by Debugger Rule (p. 949). The Debugger `rule_configs` class provides tools to configure a list of actions, including automatically stopping training jobs and sending notifications using Amazon Simple Notification Service (Amazon SNS) when the Debugger rules find training issues.

Step 1: Set Up Amazon SNS, Create an SMDebugRules Topic, and Subscribe to the Topic

This section walks you through how to set up an Amazon SNS `SMDebugRules` topic, subscribe to it, and confirm the subscription to receive notifications from the Debugger rules.

Note
For more information about billing for Amazon SNS, see Amazon SNS pricing and Amazon SNS FAQs.

To create a SMDebugRules topic

2. In the left navigation pane, choose Topics.
3. On the Topics page, choose Create topic.
4. On the Create topic page, in the Details section, do the following:
   a. For Type, choose Standard for topic type.
   b. In Name, enter SMDebugRules.
5. Skip all other optional settings and choose Create topic. If you want to learn more about the optional settings, see Creating an Amazon SNS topic.

To subscribe to the SMDebugRules topic

2. In the left navigation pane, choose Subscriptions.
3. On the Subscriptions page, choose Create subscription.
4. On the Create subscription page, in the Details section, do the following:
   b. For Protocol, choose Email or SMS.
   c. For Endpoint, enter the endpoint value, such as an email address or a phone number that you want to receive notifications.

   Note
Make sure you type the correct email address and phone number. Phone numbers must include +, a country code, and phone number, with no special characters or spaces. For example, the phone number +1 (222) 333-4444 is formatted as +12223334444.
5. Skip all other optional settings and choose Create subscription. If you want to learn more about the optional settings, see Subscribing to an Amazon SNS topic.

After you subscribe to the SMDebugRules topic, you receive the following confirmation message in email or by phone:
AWS Notification - Subscription Confirmation

SMDebugRules <no-reply@sns.amazonaws.com>
To:

You have chosen to subscribe to the topic:

To confirm this subscription, click or visit the link below (if this was in error no action is necessary):
Confirm subscription

Please do not reply directly to this email. If you wish to remove yourself from receiving all future SNS subscription confirmation requests please send an email to sns-opt-out.

For more information about Amazon SNS, see Mobile text messaging (SMS) and Email notifications in the Amazon SNS Developer Guide.

Step 2: Set Up Your IAM Role to Attach Required Policies

In this step, you add the required policies to your IAM role.

To add the required policies to your IAM role

1. Sign in to the AWS Management Console and open the IAM console at https://console.aws.amazon.com/iam/.
2. In the left navigation pane, choose Policies, and choose Create policy.
3. On the Create policy page, do the following to create a new sns-access policy:
   a. Choose the JSON tab.
   b. Paste the JSON strings formatted in bold in the following code) into the "Statement", replacing the 12-digit AWS account ID with your AWS account ID.

   ```json
   { 
   "Version": "2012-10-17",
   "Statement": [ 
   { 
   "Sid": "VisualEditor0",
   "Effect": "Allow",
   "Action": [ 
   "sns:Publish",
   "sns:CreateTopic",
   "sns:Subscribe"
   ],
   "Resource": "arn:aws:sns::111122223333:SMDebugRules"
   } 
   ]
   }
   ```
   c. At the bottom of the page, choose Review policy.
   d. On the Review policy page, for Name, enter sns-access.
   e. At the bottom of the page, choose Create policy.
4. Go back to the IAM console, and choose Roles in the left navigation pane.
5. Look up the IAM role that you use for SageMaker model training and choose that IAM role.
6. On the Permissions tab of the Summary page, choose Attach policies.
7. Search for the sns-access policy, select the check box next to the policy, and then choose Attach policy.
For more examples of setting up IAM policies for Amazon SNS, see Example cases for Amazon SNS access control.

**Step 3: Configure Debugger Rules with the Built-in Actions**

After successfully finishing the required settings in the preceding steps, you can configure the Debugger built-in actions for debugging rules as shown in the following example script. You can choose which built-in actions to use while building the actions list object. The rule_configs is a helper module that provides high-level tools to configure Debugger built-in rules and actions. The following built-in actions are available for Debugger:

- `rule_configs.StopTraining()` – Stops a training job when the Debugger rule finds an issue.
- `rule_configs.Email("abc@abc.com")` – Sends a notification via email when the Debugger rule finds an issue. Use the email address that you used when you set up your SNS topic subscription.
- `rule_configs.SMS("+1234567890")` – Sends a notification via text message when the Debugger rule finds an issue. Use the phone number that you used when you set up your SNS topic subscription.

**Note**

Make sure you type the correct email address and phone number. Phone numbers must include +, a country code, and a phone number, with no special characters or spaces. For example, the phone number +1 (222) 333-4444 is formatted as +12223334444.

You can use all of the built-in actions or a subset of actions by using the `rule_configs.ActionList()` method, which takes the built-in actions and configures a list of actions.

For example, if you add all of the three actions as shown in the following script, Debugger stops your training jobs, and then sends notifications to your email and phone.

```python
from sagemaker.debugger import Rule, rule_configs

# Configure an action list object for Debugger rules
actions = rule_configs.ActionList(
    rule_configs.StopTraining(),
    rule_configs.Email("abc@abc.com"),
    rule_configs.SMS("+1234567890")
)

# Configure rules for debugging with the actions parameter
rules = [
    Rule.sagemaker(
        base_config=rule_configs.built_in_rule(),  # Required
        rule_parameters={"parameter_key": value},  # Optional
        actions=actions
    )
]

estimator = Estimator(
    ...
    rules = rules
)
estimator.fit(wait=False)
```

While the training job is running, the Debugger built-in action sends notification emails and text messages whenever the rule finds issues with your training job. The following screenshot shows an example of email notification for a training job that has a stalled training job issue.
The following screenshot shows an example text notification that Debugger sends when the rule finds a StalledTraining issue.

Considerations for Using the Debugger Built-in Actions

- To use the Debugger built-in actions, an internet connection is required. This feature is not supported in the network isolation mode provided by Amazon SageMaker or Amazon VPC.
- The built-in actions cannot be used for Debugger ProfilerRule (p. 949).
- The built-in actions cannot be used on training jobs with spot training interruptions.
- In email or text notifications, None appears at the end of messages. This does not have any meaning, so you can disregard the text None.

Create Actions on Rules Using Amazon CloudWatch and AWS Lambda

Amazon CloudWatch collects Amazon SageMaker model training job logs and Amazon SageMaker Debugger rule processing job logs. Configure Debugger with Amazon CloudWatch Events and AWS Lambda to take action based on Debugger rule evaluation status.

CloudWatch Logs for Debugger Rules and Training Jobs

To find training job logs and Debugger rule job logs

2. In the left navigation pane under the Log node, choose Log Groups.
3. In the log groups list, do the following:
   - Choose /aws/sagemaker/TrainingJobs for training job logs.
Choose /aws/sagemaker/ProcessingJobs for Debugger rule job logs.

You can use the training and Debugger rule job status in the CloudWatch logs to take further actions when there are training issues.

For more information about monitoring training jobs using CloudWatch, see Monitor Amazon SageMaker.

**Set Up Debugger for Automated Training Job Termination Using CloudWatch and Lambda**

The Debugger rules monitor training job status, and a CloudWatch Events rule watches the Debugger rule training job evaluation status.

**Step 1: Create a Lambda Function**

**To create a Lambda function**

1. Open the AWS Lambda console at https://console.aws.amazon.com/lambda/.
2. In the left navigation pane, choose Functions and then choose Create function.
3. On the Create function page, choose Author from scratch option.
4. In the Basic information section, enter a Function name (for example, debugger-rule-stop-training-job).
5. For Runtime, choose Python 3.7.
6. For Permissions, expand the drop down option, and choose Choose or create an execution role.
7. For Execution role, choose Use an existing role and choose the IAM role that you used for your SageMaker Notebook instance or SageMaker Studio.
   
   **Note**
   Make sure you use the same execution role for the training environment, otherwise the Lambda function won't properly react to the Debugger rule status changes. If you are unsure which execution role to choose, run the following code in a Jupyter notebook cell to retrieve the execution role output:

   ```python
   import sagemaker
   sagemaker.get_execution_role()
   ```

8. At the bottom of the page, choose Create function.

The following figure shows an example of the Create function page with the input fields and selections completed.
Step 2: Configure the Lambda function

To configure the Lambda function

1. In the **Function code** section of the configuration page, paste the following Python script in the Lambda code editor pane. The `lambda_handler` function monitors the Debugger rule evaluation status collected by CloudWatch and triggers the `StopTrainingJob` API operation. The AWS SDK for Python (Boto3) client for SageMaker provides a high-level method, `stop_training_job`, which triggers the `StopTrainingJob` API operation.

```python
import json
import boto3
import logging

def lambda_handler(event, context):
    training_job_name = event.get("detail").get("TrainingJobName")
    eval_statuses = event.get("detail").get("DebugRuleEvaluationStatuses", None)

    if eval_statuses is None or len(eval_statuses) == 0:
        logging.info("Couldn't find any debug rule statuses, skipping...")
        return {
            'statusCode': 200,
            'body': json.dumps('Nothing to do')
        }
```
client = boto3.client('sagemaker')

for status in eval_statuses:
    if status.get("RuleEvaluationStatus") == "IssuesFound":
        logging.info(
            'Evaluation of rule configuration {} resulted in "IssuesFound". ' 
            'Attempting to stop training job {}.'.format(
                status.get("RuleConfigurationName"), training_job_name
            )
        
try:
    client.stop_training_job(
        TrainingJobName=training_job_name
    )
except Exception as e:
    logging.error(
        "Encountered error while trying to " 
        "stop training job {}": {}".format(
            training_job_name, str(e)
        )
    )

raise e

return None

For more information about the Lambda code editor interface, see Creating functions using the AWS
Lambda console editor.

2. Skip all other settings and choose Save at the top of the configuration page.

Step 3: Create a CloudWatch Events Rule and Link to the Lambda Function for Debugger

To create a CloudWatch Events rule and link to the Lambda function for Debugger

2. In the left navigation pane, choose Rules under the Events node.
3. Choose Create rule.
4. In the Event Source section of the Step 1: Create rule page, choose SageMaker for Service Name,
   and choose SageMaker Training Job State Change for Event Type. The Event Pattern Preview
   should look like the following example JSON strings:

   ```
   {
       "source": [
           "aws.sagemaker"
       ],
       "detail-type": [
           "SageMaker Training Job State Change"
       ]
   }
   ```

5. In the Targets section, choose Add target*, and choose the debugger-rule-stop-training-job
   Lambda function that you created. This step links the CloudWatch Events rule with the Lambda
   function.
6. Choose Configure details and go to the Step 2: Configure rule details page.
7. Specify the CloudWatch rule definition name. For example, debugger-cw-event-rule.
8. Choose Create rule to finish.
9. Go back to the Lambda function configuration page and refresh the page. Confirm that it’s
   configured correctly in the Designer panel. The CloudWatch Events rule should be registered as a
   trigger for the Lambda function. The configuration design should look like the following example:
Run Example Notebooks to Test Automated Training Job Termination

You can run the following example notebooks, which are prepared for experimenting with stopping a training job using Debugger’s built-in rules.

- **Amazon SageMaker Debugger - Reacting to CloudWatch Events from Debugger rules**

  This example notebook runs a training job that has a vanishing gradient issue. The Debugger VanishingGradient (p. 969) built-in rule is used while constructing the SageMaker TensorFlow estimator. When the Debugger rule detects the issue, the training job is terminated.

- **Detect stalled training and stop training job using Debugger rule**

  This example notebook runs a training script with a code line that forces it to sleep for 10 minutes. The Debugger StalledTrainingRule (p. 981) built-in rule invokes issues and stops the training job.

**Disable the CloudWatch Events Rule to Stop Using the Automated Training Job Termination**

If you want to disable the automated training job termination, you need to disable the CloudWatch Events rule. In the Lambda **Designer** panel, choose the **EventBridge (CloudWatch Events)** block linked to the Lambda function. This shows an **EventBridge** panel below the **Designer** panel (for example, see the previous screen shot). Select the check box next to **EventBridge (CloudWatch Events): debugger-cw-event-rule**, and then choose **Disable**. If you want to use the automated termination functionality later, you can enable the CloudWatch Events rule again.
Analyze Data Using the SMDebug Client Library

While your training job is running or after it has completed, you can access the training data collected by Debugger using the Amazon SageMaker Python SDK and the SMDebug client library. The SMDebug library provides analysis and visualization tools that enable you to drill down into your training job data.

To install the library and use the SMDebug analysis tools (in a JupyterLab notebook or an iPython kernel)

```bash
! pip install -U smdebug
```

The following topics walk you through how to use the SMDebug tools to visualize and analyze the training data collected by Debugger.

Analyze system and framework metrics

- Access the Monitoring and Profiling Data (p. 1007)
- Plot the System Metrics and Framework Metrics Data (p. 1008)
- Access the Profiling Data Using the Pandas Data Parsing Tool (p. 1009)
- Access the Python Profiling Stats Data (p. 1009)
- Merge Timelines of Different Profiling Trace Files (p. 1012)
- Profiling Data Loader (p. 1014)

Access the Monitoring and Profiling Data

The SMDebug `TrainingJob` class reads data from the S3 bucket where the system and framework metrics are saved.

To set up a `TrainingJob` object and retrieve profiling event files of a training job

```python
from smdebug.profiler.analysis.notebook_utils.training_job import TrainingJob
tj = TrainingJob(training_job_name, region)
```

Tip
You need to specify the `training_job_name` and `region` parameters to log to a training job. There are two ways to specify the training job information:

- Use the SageMaker Python SDK while the estimator is still attached to the training job.

```python
import sagemaker
training_job_name=estimator.latest_training_job.job_name
region=sagemaker.Session().boto_region_name
```

- Pass strings directly.

```python
training_job_name="your-training-job-name-YYYY-MM-DD-HH-MM-SS-SSS"
region="us-west-2"
```

To retrieve a description of the training job description and the S3 bucket URI where the metric data are saved

```python
tj.describe_training_job()
tj.get_config_and_profiler_s3_output_path()
```
To check if the system and framework metrics are available from the S3 URI

```python
tj.wait_for_sys_profiling_data_to_be_available()
tj.wait_for_framework_profiling_data_to_be_available()
```

To create system and framework reader objects after the metric data become available

```python
system_metrics_reader = tj.get_systems_metrics_reader()
framework_metrics_reader = tj.get_framework_metrics_reader()
```

To refresh and retrieve the latest training event files

The reader objects have an extended method, `refresh_event_file_list()`, to retrieve the latest training event files.

```python
system_metrics_reader.refresh_event_file_list()
framework_metrics_reader.refresh_event_file_list()
```

**Plot the System Metrics and Framework Metrics Data**

You can use the system and algorithm metrics objects for the following visualization classes to plot timeline graphs and histograms.

**Note**

To visualize the data with narrowed-down metrics in the following visualization object plot methods, specify `select_dimensions` and `select_events` parameters. For example, if you specify `select_dimensions=['GPU']`, the plot methods filter the metrics that include the "GPU" keyword. If you specify `select_events=['total']`, the plot methods filter the metrics that include the "total" event tags at the end of the metric names. If you enable these parameters and give the keyword strings, the visualization classes return the charts with filtered metrics.

- The `MetricsHistogram` class

```python
from smdebug.profiler.analysis.notebook_utils.metrics_histogram import MetricsHistogram
metrics_histogram = MetricsHistogram(system_metrics_reader)
metrics_histogram.plot(
    startime=0,
    endtime=system_metrics_reader.get_timestamp_of_latest_available_file(),
    select_dimensions=['CPU', 'GPU', 'I/O'],  # optional
    select_events=['total']                   # optional
)
```

- The `StepTimelineChart` class

```python
from smdebug.profiler.analysis.notebook_utils.step_timeline_chart import StepTimelineChart
view_step_timeline_chart = StepTimelineChart(framework_metrics_reader)
```

- The `StepHistogram` class

```python
from smdebug.profiler.analysis.notebook_utils.step_histogram import StepHistogram
step_histogram = StepHistogram(framework_metrics_reader)
step_histogram.plot(
    startime=step_histogram.last_timestamp - 5 * 1000 * 1000,
```
• The **TimelineCharts** class

```python
from smdebug.profiler.analysis.notebook_utils.timeline_charts import TimelineCharts

view_timeline_charts = TimelineCharts(
    system_metrics_reader,
    framework_metrics_reader,
    select_dimensions=['CPU', 'GPU', 'I/O'], # optional
    select_events=['total'] # optional
)

view_timeline_charts.plot_detailed_profiler_data([700,710])
```

• The **Heatmap** class

```python
from smdebug.profiler.analysis.notebook_utils.heatmap import Heatmap

view_heatmap = Heatmap(
    system_metrics_reader,
    framework_metrics_reader,
    select_dimensions=['CPU', 'GPU', 'I/O'], # optional
    select_events=['total'], # optional
    plot_height=450
)
```

### Access the Profiling Data Using the Pandas Data Parsing Tool

The following **PandasFrame** class provides tools to convert the collected profiling data to Pandas data frame.

```python
from smdebug.profiler.analysis.utils.profiler_data_to_pandas import PandasFrame

pf = PandasFrame(tj.profiler_s3_output_path)
system_metrics_df = pf.get_all_system_metrics()
framework_metrics_df = pf.get_all_framework_metrics(
    selected_framework_metrics=[
        'Step:ModeKeys.TRAIN',
        'Step:ModeKeys.GLOBAL'
    ]
)
```

### Access the Python Profiling Stats Data

The Python profiling provides framework metrics related to Python functions and operators in your training scripts and the SageMaker deep learning frameworks.

#### Training Modes and Phases for Python Profiling

To profile specific intervals during training to partition statistics for each of these intervals, Debugger provides tools to set modes and phases.
For training modes, use the following `PythonProfileModes` class:

```python
from smdebug.profiler.python_profile_utils import PythonProfileModes
```

This class provides the following options:

- `PythonProfileModes.TRAIN` – Use if you want to profile the target steps in the training phase. This mode option available only for TensorFlow.
- `PythonProfileModes.EVAL` – Use if you want to profile the target steps in the evaluation phase. This mode option available only for TensorFlow.
- `PythonProfileModes.PREDICT` – Use if you want to profile the target steps in the prediction phase. This mode option available only for TensorFlow.
- `PythonProfileModes.GLOBAL` – Use if you want to profile the target steps in the global phase, which includes the previous three phases. This mode option available only for PyTorch.
- `PythonProfileModes.PRE_STEP_ZERO` – Use if you want to profile the target steps in the initialization stage before the first training step of the first epoch starts. This phase includes the initial job submission, uploading the training scripts to EC2 instances, preparing the EC2 instances, and downloading input data. This mode option available for both TensorFlow and PyTorch.
- `PythonProfileModes.POST_HOOK_CLOSE` – Use if you want to profile the target steps in the finalization stage after the training job has done and the Debugger hook is closed. This phase includes profiling data while the training jobs are finalized and completed. This mode option available for both TensorFlow and PyTorch.

For training phases, use the following `StepPhase` class:

```python
from smdebug.profiler.analysis.utils.python_profile_analysis_utils import StepPhase
```

This class provides the following options:

- `StepPhase.START` – Use to specify the start point of the initialization phase.
- `StepPhase.STEP_START` – Use to specify the start step of the training phase.
- `StepPhase.FORWARD_PASS_END` – Use to specify the steps where the forward pass ends. This option is available only for PyTorch.
- `StepPhase.STEP_END` – Use to specify the end steps in the training phase. This option is available only for TensorFlow.
- `StepPhase.END` – Use to specify the ending point of the finalization (post-hook-close) phase. If the callback hook is not closed, the finalization phase profiling does not occur.

**Python Profiling Analysis Tools**

Debugger supports the Python profiling with two profiling tools:

- `cProfile` – The standard python profiler. `cProfile` collects framework metrics on CPU time for every function called when profiling was enabled.
- `Pyinstrument` – This is a low overhead Python profiler sampling profiling events every milliseconds.

To learn more about the Python profiling options and what's collected, see [Start a Training Job with the Default System Monitoring and Customized Framework Profiling with Different Profiling Options](p. 932).

The following methods of the `PythonProfileAnalysis`, `cProfileAnalysis`, `PyinstrumentAnalysis` classes are provided to fetch and analyze the Python profiling data. Each function loads the latest data from the default S3 URI.
To set Python profiling objects for analysis, use the cProfileAnalysis or PyinstrumentAnalysis classes as shown in the following example code. It shows how to set a cProfileAnalysis object, and if you want to use PyinstrumentAnalysis, replace the class name.

```python
python_analysis = cProfileAnalysis(
    local_profile_dir=tf_python_stats_dir,
    s3_path=tj.profiler_s3_output_path
)
```

The following methods are available for the cProfileAnalysis and PyinstrumentAnalysis classes to fetch the Python profiling stats data:

- `python_analysis.fetch_python_profile_stats_by_time(start_time_since_epoch_in_secs, end_time_since_epoch_in_secs)` - Takes in a start time and end time, and returns the function stats of step stats whose start or end times overlap with the provided interval.
- `python_analysis.fetch_python_profile_stats_by_step(start_step, end_step, mode, start_phase, end_phase)` - Takes in a start step and end step and returns the function stats of all step stats whose profiled step satisfies start_step <= step < end_step.
  - `start_step` and `end_step` (str) – Specify the start step and end step to fetch the Python profiling stats data.
  - `mode` (str) – Specify the mode of training job using the PythonProfileModes enumerator class. The default is PythonProfileModes.TRAIN. Available options are provided in the Training Modes and Phases for Python Profiling (p. 1009) section.
  - `start_phase` (str) – Specify the start phase in the target step(s) using the StepPhase enumerator class. This parameter enables profiling between different phases of training. The default is StepPhase.STEP_START. Available options are provided in the Training Modes and Phases for Python Profiling (p. 1010) section.
  - `end_phase` (str) – Specify the end phase in the target step(s) using the StepPhase enumerator class. This parameter sets up the end phase of training. Available options are as same as the ones for the `start_phase` parameter. The default is StepPhase.STEP_END. Available options are provided in the Training Modes and Phases for Python Profiling (p. 1010) section.
- `python_analysis.fetch_profile_stats_between_modes(start_mode, end_mode)` - Fetches stats from the Python profiling between the start and end modes.
- `python_analysis.fetch_pre_step_zero_profile_stats()` - Fetches the stats from the Python profiling until step 0.
- `python_analysis.fetch_post_hook_close_profile_stats()` - Fetches stats from the Python profiling after the hook is closed.
- `python_analysis.list_profile_stats()` - Returns a DataFrame of the Python profiling stats. Each row holds the metadata for each instance of profiling and the corresponding stats file (one per step).
- `python_analysis.list_available_node_ids()` - Returns a list the available node IDs for the Python profiling stats.

The cProfileAnalysis class specific methods:

- `fetch_profile_stats_by_training_phase()` - Fetches and aggregates the Python profiling stats for every possible combination of start and end modes. For example, if a training and validation phases are done while detailed profiling is enabled, the combinations are (PRE_STEP_ZERO, TRAIN), (TRAIN, TRAIN), (TRAIN, EVAL), (EVAL, EVAL), and (EVAL, POST_HOOK_CLOSE). All stats files within each of these combinations are aggregated.
• `fetch_profile_stats_by_job_phase()` – Fetches and aggregates the Python profiling stats by job phase. The job phases are initialization (profiling until step 0), training_loop (training and validation), and finalization (profiling after the hook is closed).

### Merge Timelines of Different Profiling Trace Files

The SMDebug client library provide profiling analysis and visualization tools for merging timelines of system metrics, framework metrics, and Python profiling data collected by Debugger.

**Tip**
Before proceeding, you need to set a TrainingJob object that will be utilized throughout the examples in this page. For more information about setting up a TrainingJob object, see Access the Monitoring and Profiling Data (p. 1007).

The `MergedTimeline` class provides tools to integrate and correlate different profiling information in a single timeline. After Debugger captures profiling data and annotations from different phases of a training job, JSON files of trace events are saved in a default `tracefolder` directory.

- For annotations in the Python layers, the trace files are saved in `*pythontimeline.json`.
- For annotations in the TensorFlow C++ layers, the trace files are saved in `*model_timeline.json`.
- Tensorflow profiler saves events in a `*trace.json.gz` file.

**Tip**
If you want to list all of the JSON trace files, use the following AWS CLI command:

```bash
/aws s3 ls {tj.profiler_s3_output_path} --recursive | grep '\.json$'
```

As shown in the following animated screenshot, putting and aligning the trace events captured from the different profiling sources in a single plot can provide an overview of the entire events occurring in different phases of the training job.
Tip
To interact with the merged timeline on the tracing app using a keyboard, use the W key for zooming in, the A key for shifting to the left, the S key for zooming out, and the D key for shifting to the right.

The multiple event trace JSON files can be merged into one trace event JSON file using the following MergedTimeline API operation and class method from the smdebug.profiler.analysis.utils.merge_timelines module.

```python
from smdebug.profiler.analysis.utils.merge_timelines import MergedTimeline

combined_timeline = MergedTimeline(path, file_suffix_filter, output_directory)
combined_timeline.merge_timeline(start, end, unit)
```

The MergedTimeline API operation passes the following parameters:

- `path` (str) – Specify a root folder (/profiler-output) that contains system and framework profiling trace files. You can locate the profiler-output using the SageMaker estimator classmethod or the TrainingJob object. For example, estimator.latest_job_profiler_artifacts_path() or tj.profiler_s3_output_path.

- `file_suffix_filter` (list) – Specify a list of file suffix filters to merge timelines. Available suffix filters are ["model_timeline.json", "pythontimeline.json", "trace.json.gz"]. If this parameter is not manually specified, all of the trace files are merged by default.

- `output_directory` (str) – Specify a path to save the merged timeline JSON file. The default is to the directory specified for the `path` parameter.

The `merge_timeline()` classmethod passes the following parameters to execute the merging process:

- `start` (int) – Specify start time (in microseconds and in Unix time format) or start step to merge timelines.

- `end` (int) – Specify end time (in microseconds and in Unix time format) or end step to merge timelines.

- `unit` (str) – Choose between "time" and "step". The default is "time".

Using the following example codes, execute the `merge_timeline()` method and download the merged JSON file.

- Merge timeline with the "time" unit option. The following example code merges all available trace files between the Unix start time (the absolute zero Unix time) and the current Unix time, which means that you can merge the timelines for the entire training duration.

```python
import time
from smdebug.profiler.analysis.utils.merge_timelines import MergedTimeline
from smdebug.profiler.profiler_constants import CONVERT_TO_MICROSECS

combined_timeline = MergedTimeline(tj.profiler_s3_output_path, output_directory="./")
combined_timeline.merge_timeline(0, int(time.time() * CONVERT_TO_MICROSECS))
```

- Merge timeline with the "step" unit option. The following example code merges all available timelines between step 3 and step 9.

```python
from smdebug.profiler.analysis.utils.merge_timelines import MergedTimeline

combined_timeline = MergedTimeline(tj.profiler_s3_output_path, output_directory="./")
combined_timeline.merge_timeline(3, 9, unit="step")
```
Open the Chrome tracing app at `chrome://tracing` on a Chrome browser, and open the JSON file. You can explore the output to plot the merged timeline.

### Profiling Data Loader

In PyTorch, data loader iterators, such as `SingleProcessingDataLoaderIter` and `MultiProcessingDataLoaderIter`, are initiated at the beginning of every iteration over a dataset. During the initialization phase, PyTorch turns on worker processes depending on the configured number of workers, establishes data queue to fetch data and `pin_memory` threads.

To use the PyTorch data loader profiling analysis tool, import the following `PT_dataloader_analysis` class:

```python
from smdebug.profiler.analysis.utils.pytorch_dataloader_analysis import PT_dataloader_analysis
```

Pass the profiling data retrieved as a Pandas frame data object in the Access the Profiling Data Using the Pandas Data Parsing Tool (p. 1009) section:

```python
pt_analysis = PT_dataloader_analysis(pf)
```

The following functions are available for the `pt_analysis` object:

1. **`pt_analysis.analyze_dataloaderIter_initialization()`**

   The analysis outputs the median and maximum duration for these initializations. If there are outliers, (i.e duration is greater than 2 * median), the function prints the start and end times for those durations. These can be used to inspect system metrics during those time intervals.

   The following list shows what analysis is available from this class method:
   - Which type of data loader iterators were initialized.
   - The number of workers per iterator.
   - Inspect whether the iterator was initialized with or without `pin_memory`.
   - Number of times the iterators were initialized during training.

2. **`pt_analysis.analyze_dataloaderWorkers()`**

   The following list shows what analysis is available from this class method:
   - The number of worker processes that were spun off during the entire training.
   - Median and maximum duration for the worker processes.
   - Start and end time for the worker processes that are outliers.

3. **`pt_analysis.analyze_dataloader_getnext()`**

   The following list shows what analysis is available from this class method:
   - Number of `GetNext` calls made during the training.
   - Median and maximum duration in microseconds for `GetNext` calls.
   - Start time, End time, duration and worker id for the outlier `GetNext` call duration.

4. **`pt_analysis.analyze_batchtime(start_timestamp, end_timestamp, select_events=[".*"], select_dimensions=[".*"]`**

   Debugger collects the start and end times of all the `GetNext` calls. You can find the amount of time spent by the training script on one batch of data. Within the specified time window, you can identify the calls that are not directly contributing to the training. These calls can be from the following...
operations: computing the accuracy, adding the losses for debugging or logging purposes, and printing the debugging information. Operations like these can be compute intensive or time consuming. We can identify such operations by correlating the Python profiler, system metrics, and framework metrics.

The following list shows what analysis is available from this class method:

- Profile time spent on each data batch, `BatchTime_in_seconds`, by finding the difference between start times of current and subsequent GetNext calls.
- Find the outliers in `BatchTime_in_seconds` and start and end time for those outliers.
- Obtain the system and framework metrics during those `BatchTime_in_seconds` timestamps. This indicates where the time was spent.
- `pt_analysis.plot_the_window()`

Plots a timeline charts between a start timestamp and the end timestamp.

Amazon SageMaker Debugger Advanced Topics and Reference Documentation

The following sections contain advanced topics, reference documentation for the API operations, exceptions, and known limitations for Debugger.

Topics

- Amazon SageMaker Debugger API Operations (p. 1015)
- Use the SageMaker and Debugger Configuration API Operations to Create, Update, Debug, and Profile Your Training Job (p. 1016)
- Use Debugger Docker Images for Built-in or Custom Rules (p. 1019)
- Amazon SageMaker Debugger Exceptions (p. 1022)
- Considerations for Amazon SageMaker Debugger (p. 1022)

Amazon SageMaker Debugger API Operations

Amazon SageMaker Debugger has API operations in several locations that are used to implement its monitoring and analysis of model training.

Amazon SageMaker Debugger also provides the open source SMDebug Python library at awslabs/sagemaker-debugger that is used to configure built-in rules, define custom rules, register hooks to collect output tensor data from training jobs.

The Amazon SageMaker Python SDK is a high-level SDK focused on machine learning experimentation. The SDK can be used to deploy built-in or custom rules defined with the SMDebug Python library to monitor and analyze these tensors using SageMaker estimators.

Debugger has added operations and types to the Amazon SageMaker API that enable the platform to use Debugger when training a model and to manage the configuration of inputs and outputs.

- `CreateTrainingJob` and `UpdateTrainingJob` use the following Debugger APIs to configure tensor collections, rules, rule images, and profiling options:
  - `CollectionConfiguration`
  - `DebugHookConfig`
  - `DebugRuleConfiguration`
  - `TensorBoardOutputConfig`
  - `ProfilerConfig`
Use the SageMaker and Debugger Configuration API Operations to Create, Update, Debug, and Profile Your Training Job

The preceding topics focuses on using Debugger through Amazon SageMaker Python SDK, which is a wrapper around AWS boto3 API operations for SageMaker. This offers a high-level experience of accessing the Amazon SageMaker API operations. In case you need to use the SageMaker API operations directly with other SDKs, such as Java, Go, C++, and many others, the following topics cover how to configure CreateTrainingJob, UpdateTrainingJob, Debugger configuration APIs, and their parameters to use the Debugger built-in and custom rules.

Add Debugger Built-in Rule Configuration to the CreateTrainingJob API Operation

Amazon SageMaker Debugger built-in rules can be configured for a training job using the DebugHookConfig, DebugRuleConfiguration, ProfilerConfig, and ProfilerRuleConfiguration objects in the CreateTrainingJob API operation. The built-in and custom rules run in processing containers, and you can find the ECR image URIs at the Use Debugger Docker Images for Built-in or Custom Rules (p. 1019) topic. You need to specify the right image URI in the RuleEvaluatorImage parameter, and the following examples walk you through how to set up the JSON strings to request CreateTrainingJob.

To configure a Debugger rule for debugging model parameters

The following code examples show how to configure a built-in VanishingGradient rule using this SageMaker API.

Specify the Debugger hook configuration as follows:

```json
DebugHookConfig: {
    "S3OutputPath": "s3://bucket/path-to-tensors",
    "CollectionConfigurations": [
        {
            "CollectionName": "gradients",
            "CollectionParameters" : {
                "save_interval": "500"
            }
        }
    ]
}
```

This will make the training job save the tensor collection, gradients, every save_interval of 500 steps. The following code example of the DebugRuleConfigurations API demonstrates how to run the built-in VanishingGradient rule on the saved gradients collection.
With a configuration like the one in this sample, Amazon SageMaker Debugger starts a rule evaluation job for your training job using the SageMaker VanishingGradient rule on the collection of gradients tensor.

To configure a Debugger built-in rule for profiling system and framework metrics:

The following example code shows how to specify the ProfilerConfig API operation to enable collecting system and framework metrics.

**Target Step**

```json
"ProfilerConfig": {
  "ProfilingIntervalInMilliseconds": 500,
  "ProfilingParameters": {
    "DetailedProfilingConfig": {
      "StartTimeInSecSinceEpoch": "1234567890", // The default is the current time
      "DurationInSeconds": "1", // The default is duration of 1 step
      "cProfileTimer": "total_time", // Available options: cpu, off_cpu, total_time
      "ProfilerName": "cProfile", // Available options: cProfile, Pyinstrument
    },
    "PythonProfilingConfig": {
      "StartTimeInSecSinceEpoch": "1234567890", // The default is the current time
      "DurationInSeconds": "1", // The default is duration of 1 step
      "cProfileTimer": "total_time", // Available options: cpu, off_cpu, total_time
    }
  }
}
```

**Target Time Duration**

```json
"ProfilerConfig": {
  "ProfilingIntervalInMilliseconds": 500,
  "ProfilingParameters": {
    "DetailedProfilingConfig": {
      "StartTimeInSecSinceEpoch": "12345677890", // The default is the current time
      "DurationInSeconds": "1", // The default is duration of 1 step
    },
    "PythonProfilingConfig": {
      "StartTimeInSecSinceEpoch": "12345677890", // The default is the current time
      "DurationInSeconds": "1", // The default is duration of 1 step
      "cProfileTimer": "total_time", // Available options: cpu, off_cpu, total_time
    }
  }
}
```
"ProfilerName": "cProfile", // Available options: cProfile, Pyinstrument
},
"DataLoaderProfilingConfig": {
  "StartTimeInSecSinceEpoch": "123456789", // The default is the current time
  "DurationInSeconds": "1" // The default is duration of 1 step
}
}

The following example code shows how to configure the ProfilerReport rule.

"ProfilerRuleConfigurations": [
  {
    "RuleConfigurationName": "ProfilerReport",
    "RuleEvaluatorImage": "503895931360.dkr.ecr.us-east-1.amazonaws.com/sagemaker-debugger-rules:latest",
    "RuleParameters": {
      "rule_to_invoke": "ProfilerReport",
      "CPUBottleneck_cpu_threshold": "90",
      "IOBottleneck_threshold": "90"
    }
  }
]

Update Debugger Profiling Configuration Using the UpdateTrainingJob API Operation

Debugger profiling configuration can be updated while your training job is running by using the UpdateTrainingJob API operation. Configure new ProfilerConfig and ProfilerRuleConfiguration objects, and specify the training job name to the TrainingJobName parameter.

{  "ProfilerConfig": {
    "DisableProfiler": boolean,
    "ProfilingIntervalInMilliseconds": number,
    "ProfilingParameters": {
      "string" : "string"
    }
  },
  "ProfilerRuleConfigurations": [
    {
      "RuleConfigurationName": "string",
      "RuleEvaluatorImage": "string",
      "RuleParameters": {
        "string" : "string"
      }
    }
  ],
  "TrainingJobName": "your-training-job-name-YYYY-MM-DD-HH-MM-SS-SSS"
}

Add Debugger Custom Rule Configuration to the CreateTrainingJob API Operation

A custom rule can be configured for a training job using the DebugHookConfig and DebugRuleConfiguration objects in the CreateTrainingJob API operation. The following code sample shows how to configure a custom ImproperActivation rule written with the smdebug library.
using this SageMaker API operation. This example assumes that you’ve written the custom rule in `custom_rules.py` file and uploaded it to an Amazon S3 bucket. The example provides pre-built Docker images that you can use to run your custom rules. These are listed at Amazon SageMaker Debugger Registry URLs for Custom Rule Evaluators (p. 1021). You specify the URL registry address for the pre-built Docker image in the `RuleEvaluatorImage` parameter.

```
DebugHookConfig: {
    "S3OutputPath": "s3://bucket/",
    "CollectionConfigurations": [
        {
            "CollectionName": "relu_activations",
            "CollectionParameters": {
                "include_regex": "relu",
                "save_interval": "500",
                "end_step": "5000"
            }
        }
    ],
    DebugRulesConfigurations: [
        {
            "RuleConfigurationName": "improper_activation_job",
            "InstanceType": "ml.c4.xlarge",
            "VolumeSizeInGB": 400,
            "RuleParameters": {
                "source_s3_uri": "s3://bucket/custom_rules.py",
                "rule_to_invoke": "ImproperActivation",
                "collection_names": "relu_activations"
            }
        }
    ]
}
```

Use Debugger Docker Images for Built-in or Custom Rules

Amazon SageMaker provides two sets of Docker images for rules: one set for evaluating rules provided by SageMaker (built-in rules) and one set for evaluating custom rules provided in Python source files.

If you use the Amazon SageMaker Python SDK, you can simply use SageMaker high-level Debugger API operations with SageMaker Estimator API operations, without having to manually retrieve the Debugger Docker images and configure the `ConfigureTrainingJob` API.

If you are not using the SageMaker Python SDK, you have to retrieve a relevant pre-built container base image for the Debugger rules. Amazon SageMaker Debugger provides pre-built Docker images for built-in and custom rules, and the images are stored in Amazon Elastic Container Registry (Amazon ECR). To pull an image from an Amazon ECR repository (or to push an image to one), use the full name registry URL of the image using the `CreateTrainingJob` API. SageMaker uses the following URL patterns for the Debugger rule container image registry address.

```
<account_id>.dkr.ecr.<Region>.amazonaws.com/<ECR repository name>:<tag>
```

For the account ID in each AWS Region, Amazon ECR repository name, and tag value, see the following topics.

**Topics**

- Amazon SageMaker Debugger Registry URLs for Built-in Rule Evaluators (p. 1020)
- Amazon SageMaker Debugger Registry URLs for Custom Rule Evaluators (p. 1021)
Amazon SageMaker Debugger Registry URLs for Built-in Rule Evaluators

Use the following values for the components of the registry URLs for the images that provide built-in rules for Amazon SageMaker Debugger. For account IDs, see the following table.

**ECR Repository Name:** sagemaker-debugger-rules

**Tag:** latest

**Example of a full registry URL:**

```
904829902805.dkr.ecr.ap-south-1.amazonaws.com/sagemaker-debugger-rules:latest
```

**Account IDs for Built-in Rules Container Images by AWS Region**

<table>
<thead>
<tr>
<th>Region</th>
<th>account_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>af-south-1</td>
<td>314341159256</td>
</tr>
<tr>
<td>ap-east-1</td>
<td>199566480951</td>
</tr>
<tr>
<td>ap-northeast-1</td>
<td>430734990657</td>
</tr>
<tr>
<td>ap-northeast-2</td>
<td>578805364391</td>
</tr>
<tr>
<td>ap-south-1</td>
<td>904829902805</td>
</tr>
<tr>
<td>ap-southeast-1</td>
<td>972752614525</td>
</tr>
<tr>
<td>ap-southeast-2</td>
<td>184798709955</td>
</tr>
<tr>
<td>ca-central-1</td>
<td>519511493484</td>
</tr>
<tr>
<td>cn-north-1</td>
<td>618459771430</td>
</tr>
<tr>
<td>cn-northwest-1</td>
<td>658757709296</td>
</tr>
<tr>
<td>eu-central-1</td>
<td>482524230118</td>
</tr>
<tr>
<td>eu-north-1</td>
<td>314864569078</td>
</tr>
<tr>
<td>eu-south-1</td>
<td>563282790590</td>
</tr>
<tr>
<td>eu-west-1</td>
<td>929884845733</td>
</tr>
<tr>
<td>eu-west-2</td>
<td>250201462417</td>
</tr>
<tr>
<td>eu-west-3</td>
<td>447278800020</td>
</tr>
<tr>
<td>me-south-1</td>
<td>986000313247</td>
</tr>
<tr>
<td>sa-east-1</td>
<td>818342061345</td>
</tr>
<tr>
<td>us-east-1</td>
<td>503895931360</td>
</tr>
<tr>
<td>us-east-2</td>
<td>91544729597</td>
</tr>
<tr>
<td>us-west-1</td>
<td>685455198987</td>
</tr>
<tr>
<td>us-west-2</td>
<td>895741380848</td>
</tr>
<tr>
<td>us-gov-west-1</td>
<td>515509971035</td>
</tr>
</tbody>
</table>
Amazon SageMaker Debugger Registry URLs for Custom Rule Evaluators

Use the following values for the components of the registry URL for the images that provide custom rule evaluators for Amazon SageMaker Debugger. For account IDs, see the following table.

**ECR Repository Name:** sagemaker-debugger-rule-evaluator  
**Tag:** latest

**Example of a full registry URL:**

552407032007.dkr.ecr.ap-south-1.amazonaws.com/sagemaker-debugger-rule-evaluator:latest

**Account IDs for Custom Rules Container Images by AWS Region**

<table>
<thead>
<tr>
<th>Region</th>
<th>account_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>af-south-1</td>
<td>515950693465</td>
</tr>
<tr>
<td>ap-east-1</td>
<td>645844755771</td>
</tr>
<tr>
<td>ap-northeast-1</td>
<td>670969264625</td>
</tr>
<tr>
<td>ap-northeast-2</td>
<td>326368420253</td>
</tr>
<tr>
<td>ap-south-1</td>
<td>552407032007</td>
</tr>
<tr>
<td>ap-southeast-1</td>
<td>631532610101</td>
</tr>
<tr>
<td>ap-southeast-2</td>
<td>445670767460</td>
</tr>
<tr>
<td>ca-central-1</td>
<td>105842248657</td>
</tr>
<tr>
<td>cn-north-1</td>
<td>617202126805</td>
</tr>
<tr>
<td>cn-northwest-1</td>
<td>658559488188</td>
</tr>
<tr>
<td>eu-central-1</td>
<td>691764027602</td>
</tr>
<tr>
<td>eu-north-1</td>
<td>091235270104</td>
</tr>
<tr>
<td>eu-south-1</td>
<td>335033873580</td>
</tr>
<tr>
<td>eu-west-1</td>
<td>606966180310</td>
</tr>
<tr>
<td>eu-west-2</td>
<td>074613877050</td>
</tr>
<tr>
<td>eu-west-3</td>
<td>224335253976</td>
</tr>
<tr>
<td>me-south-1</td>
<td>050406412588</td>
</tr>
<tr>
<td>sa-east-1</td>
<td>466516958431</td>
</tr>
<tr>
<td>us-east-1</td>
<td>864354269164</td>
</tr>
<tr>
<td>us-east-2</td>
<td>840043622174</td>
</tr>
<tr>
<td>us-west-1</td>
<td>952348334681</td>
</tr>
<tr>
<td>us-west-2</td>
<td>759209512951</td>
</tr>
<tr>
<td>us-gov-west-1</td>
<td>515361955729</td>
</tr>
</tbody>
</table>
Amazon SageMaker Debugger Exceptions

Amazon SageMaker Debugger is designed to be aware that tensors required to execute a rule might not be available at every step. As a result, it raises a few exceptions, which enable you to control what happens when a tensor is missing. These exceptions are available in the `smdebug.exceptions` module. You can import them as follows:

```python
from smdebug.exceptions import *
```

The following exceptions are available:

- **TensorUnavailableForStep** – The tensor requested is not available for the step. This might mean that this step might not be saved at all by the hook, or that this step might have saved some tensors but the requested tensor is not part of them. Note that when you see this exception, it means that this tensor can never become available for this step in the future. If the tensor has reductions saved for the step, it notifies you they can be queried.
- **TensorUnavailable** – This tensor is not being saved or has not been saved by the `smdebug` API. This means that this tensor is never seen for any step in `smdebug`.
- **StepUnavailable** – The step was not saved and Debugger has no data from the step.
- **StepNotYetAvailable** – The step has not yet been seen by `smdebug`. It might be available in the future if the training is still going on. Debugger automatically loads new data as it becomes available.
- **NoMoreData** – Raised when the training ends. Once you see this, you know that there are no more steps and no more tensors to be saved.
- **IndexReaderException** – The index reader is not valid.
- **InvalidWorker** – A worker was invoked that was not valid.
- **RuleEvaluationConditionMet** – Evaluation of the rule at the step resulted in the condition being met.
- **InsufficientInformationForRuleInvocation** – Insufficient information was provided to invoke the rule.

Considerations for Amazon SageMaker Debugger

Consider the following when using Amazon SageMaker Debugger.

Considerations for Distributed Training

- **Horovod support**
  - For debugging – Debugger does not support Horovod distributed training for Keras.
  - For profiling – Debugger does not support Horovod distributed training for Keras and MXNet.
- **Parameter Server support** – Parameter server-based distributed training is not supported.

Considerations for Monitoring and Profiling

- For AWS TensorFlow, data loader metrics cannot be collected using the default `local_path` setting of the `FrameworkProfile` class. The path has to be manually configured ending in `"/"`. For example:

  ```python
  FrameworkProfile(local_path="/opt/ml/output/profiler/")
  ```

- For AWS TensorFlow, the data loader profiling configuration cannot be updated while training job is running.
- For AWS TensorFlow, `NoneType` error might occur when you use analysis tools and notebook examples with TensorFlow 2.3 training jobs and the detailed profiling option.
- Python profiling and detailed profiling are only supported for Keras API.
- To access the deep profiling feature for TensorFlow and PyTorch, currently you need to specify the latest AWS deep learning container images with CUDA 11. For example, you must specify the specific image URI in the TensorFlow and PyTorch estimator as follows:
  - For TensorFlow
    ```
    image_uri = f"763104351884.dkr.ecr.{region}.amazonaws.com/tensorflow-training:2.3.1-gpu-py37-cu110-ubuntu18.04"
    ```
  - For PyTorch
    ```
    image_uri = f"763104351884.dkr.ecr.{region}.amazonaws.com/pytorch-training:1.6.0-gpu-py36-cu110-ubuntu18.04"
    ```

Perform Automatic Model Tuning

Amazon SageMaker automatic model tuning, also known as hyperparameter tuning, finds the best version of a model by running many training jobs on your dataset using the algorithm and ranges of hyperparameters that you specify. It then chooses the hyperparameter values that result in a model that performs the best, as measured by a metric that you choose.

For example, suppose that you want to solve a binary classification problem on a marketing dataset. Your goal is to maximize the area under the curve (auc) metric of the algorithm by training an XGBoost Algorithm (p. 813) model. You don't know which values of the eta, alpha, min_child_weight, and max_depth hyperparameters to use to train the best model. To find the best values for these hyperparameters, you can specify ranges of values that SageMaker hyperparameter tuning searches to find the combination of values that results in the training job that performs the best as measured by the objective metric that you chose. Hyperparameter tuning launches training jobs that use hyperparameter values in the ranges that you specified, and returns the training job with highest auc.

You can use SageMaker automatic model tuning with built-in algorithms, custom algorithms, and SageMaker pre-built containers for machine learning frameworks.

Before you start using hyperparameter tuning, you should have a well-defined machine learning problem, including the following:

- A dataset
- An understanding of the type of algorithm you need to train
- A clear understanding of how you measure success

You should also prepare your dataset and algorithm so that they work in SageMaker and successfully run a training job at least once. For information about setting up and running a training job, see Get Started with Amazon SageMaker (p. 33).

Topics
- How Hyperparameter Tuning Works (p. 1024)
- Define Metrics (p. 1025)
- Define Hyperparameter Ranges (p. 1026)
- Tune Multiple Algorithms to Find the Best Model (p. 1028)
- Example: Hyperparameter Tuning Job (p. 1030)
- Stop Training Jobs Early (p. 1040)
- Run a Warm Start Hyperparameter Tuning Job (p. 1041)
- Automatic Model Tuning Resource Limits (p. 1045)
How Hyperparameter Tuning Works

Random Search

In a random search, hyperparameter tuning chooses a random combination of values from within the ranges that you specify for hyperparameters for each training job it launches. Because the choice of hyperparameter values doesn't depend on the results of previous training jobs, you can run the maximum number of concurrent training jobs without affecting the performance of the search.

For an example notebook that uses random search, see https://github.com/awslabs/amazon-sagemaker-examples/blob/master/hyperparameter_tuning/xgboost_random_log/hpo_xgboost_random_log.ipynb.

Bayesian Search

Bayesian search treats hyperparameter tuning like a [regression] problem. Given a set of input features (the hyperparameters), hyperparameter tuning optimizes a model for the metric that you choose. To solve a regression problem, hyperparameter tuning makes guesses about which hyperparameter combinations are likely to get the best results, and runs training jobs to test these values. After testing the first set of hyperparameter values, hyperparameter tuning uses regression to choose the next set of hyperparameter values to test.

Hyperparameter tuning uses a Amazon SageMaker implementation of Bayesian optimization.

When choosing the best hyperparameters for the next training job, hyperparameter tuning considers everything that it knows about this problem so far. Sometimes it chooses a combination of hyperparameter values close to the combination that resulted in the best previous training job to incrementally improve performance. This allows hyperparameter tuning to exploit the best known results. Other times, it chooses a set of hyperparameter values far removed from those it has tried. This allows it to explore the range of hyperparameter values to try to find new areas that are not well understood. The explore/exploit trade-off is common in many machine learning problems.

For more information about Bayesian optimization, see the following:

Basic Topics on Bayesian Optimization

- A Tutorial on Bayesian Optimization of Expensive Cost Functions, with Application to Active User Modeling and Hierarchical Reinforcement Learning
- Practical Bayesian Optimization of Machine Learning Algorithms
- Taking the Human Out of the Loop: A Review of Bayesian Optimization

Speeding up Bayesian Optimization

- Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization
- Google Vizier: A Service for Black-Box Optimization
- Learning Curve Prediction with Bayesian Neural Networks
- Speeding up automatic hyperparameter optimization of deep neural networks by extrapolation of learning curves

Advanced Modeling and Transfer Learning

- Scalable Hyperparameter Transfer Learning
• Bayesian Optimization with Tree-structured Dependencies
• Bayesian Optimization with Robust Bayesian Neural Networks
• Scalable Bayesian Optimization Using Deep Neural Networks
• Input Warping for Bayesian Optimization of Non-stationary Functions

Note
Hyperparameter tuning might not improve your model. It is an advanced tool for building machine solutions, and, as such, should be considered part of the scientific development process.
When you build complex machine learning systems like deep learning neural networks, exploring all of the possible combinations is impractical. Hyperparameter tuning can accelerate your productivity by trying many variations of a model, focusing on the most promising combinations of hyperparameter values within the ranges that you specify. To get good results, you need to choose the right ranges to explore. Because the algorithm itself is stochastic, it’s possible that the hyperparameter tuning model will fail to converge on the best answer, even if the best possible combination of values is within the ranges that you choose.

Define Metrics

Note
When you use one of the Amazon SageMaker built-in algorithms, you don’t need to define metrics. Built-in algorithms automatically send metrics to hyperparameter tuning. You do need to choose one of the metrics that the built-in algorithm emits as the objective metric for the tuning job. For a list of metrics that a built-in algorithm emits, see the Metrics table for the algorithm in Use Amazon SageMaker Built-in Algorithms (p. 635).

To optimize hyperparameters for a machine learning model, a tuning job evaluates the training jobs it launches by using a metric that the training algorithm writes to logs. Amazon SageMaker hyperparameter tuning parses your algorithm’s stdout and stderr streams to find algorithm metrics, such as loss or validation-accuracy, that show how well the model is performing on the dataset. These are the same metrics that SageMaker sends to CloudWatch Logs. For more information, see Log Amazon SageMaker Events with Amazon CloudWatch (p. 1712).

If you use your own algorithm for hyperparameter tuning, make sure that your algorithm emits at least one metric by writing evaluation data to stdout or stderr.

Note
Hyperparameter tuning sends an additional hyperparameter, _tuning_objective_metric to the training algorithm. This hyperparameter specifies the objective metric being used for the hyperparameter tuning job, so that your algorithm can use that information during training.

You can define up to 20 metrics for your tuning job to monitor. You choose one of those metrics to be the objective metric, which hyperparameter tuning uses to evaluate the training jobs. The hyperparameter tuning job returns the training job that returned the best value for the objective metric as the best training job.

You define metrics for a tuning job by specifying a name and a regular expression for each metric that your tuning job monitors. Design the regular expressions to capture the values of metrics that your algorithm emits. You pass these metrics to the CreateHyperParameterTuningJob operation in the TrainingJobDefinition parameter as the MetricDefinitions field of the AlgorithmSpecification field.

The following example defines 4 metrics:

```json
=[
  {
    
```
Define Hyperparameter Ranges

Define Hyperparameter Ranges

Hyperparameter tuning finds the best hyperparameter values for your model by searching over ranges of hyperparameters. You specify the hyperparameters and range of values over which to search by defining hyperparameter ranges for your tuning job. Choosing hyperparameters and ranges significantly affects the performance of your tuning job. For guidance on choosing hyperparameters and ranges, see Best Practices for Hyperparameter Tuning (p. 1046).

To define hyperparameter ranges by using the low-level API, you specify the names of hyperparameters and ranges of values in the ParameterRanges field of the HyperParameterTuningJobConfig parameter that you pass to the CreateHyperParameterTuningJob operation. The ParameterRanges field has three subfields, one for each of the categorical, integer, and continuous hyperparameter ranges. You can define up to 20 hyperparameters to search over. Each value of a categorical hyperparameter range counts as a hyperparameter against the limit. Hyperparameter ranges have the following structure:

"ParameterRanges": {
  "CategoricalParameterRanges": [
    {"Name": "tree_method", "Values": ["auto", "exact", "approx", "hist"]}
  ],
  "ContinuousParameterRanges": [
    
  ]}
Hyperparameter Scaling

For integer and continuous hyperparameter ranges, you can choose the scale you want hyperparameter tuning to use to search the range of values by specifying a value for the `ScalingType` field of the hyperparameter range. You can choose from the following scaling types:

**Auto**

SageMaker hyperparameter tuning chooses the best scale for the hyperparameter.

**Linear**

Hyperparameter tuning searches the values in the hyperparameter range by using a linear scale. Typically, you choose this if the range of all values from the lowest to the highest is relatively small (within one order of magnitude), because uniformly searching values from the range will give you a reasonable exploration of the entire range.

**Logarithmic**

Hyperparameter tuning searches the values in the hyperparameter range by using a logarithmic scale.

Logarithmic scaling works only for ranges that have only values greater than 0.

Choose logarithmic scaling when you are searching a range that spans several orders of magnitude. For example, if you are tuning a Tune a linear learner model (p. 752) model, and you specify a range of values between .0001 and 1.0 for the `learning_rate` hyperparameter, searching uniformly on a logarithmic scale gives you a better sample of the entire range than searching on a linear scale would, because searching on a linear scale would, on average, devote 90 percent of your training budget to only the values between .1 and 1.0, leaving only 10 percent of your training budget for the values between .0001 and .1.

**ReverseLogarithmic**

Hyperparameter tuning searches the values in the hyperparameter range by using a reverse logarithmic scale. reverse logarithmic scaling is supported only for continuous hyperparameter ranges. It is not supported for integer hyperparameter ranges.

Reverse logarithmic scaling works only for ranges that are entirely within the range 0<x<1.0.

Choose reverse logarithmic scaling when you are searching a range that is highly sensitive to small changes that are very close to 1.

For an example notebook that uses hyperparameter scaling, see https://github.com/awslabs/amazon-sagemaker-examples/blob/master/hyperparameter_tuning/xgboost_random_log/hpo_xgboost_random_log.ipynb.
Tune Multiple Algorithms to Find the Best Model

When you create a new hyperparameter optimization (HPO) job with Amazon SageMaker, you have the option of using the console or the API. You provide one or more job specifications for the different algorithms you’re testing. These are called training definitions. Each training definition has a name, an algorithm source, metrics selection, an objective metric, and a configuration for a set of hyperparameter values. It also has a data configuration for setting up the input data channels for the algorithm you choose, and a setting for the output data location. You select the resources you want to use for the training run.

Topics
- Get Started (p. 1028)
- Managing Hyperparameter Tuning Jobs (p. 1028)
- Create a new single or multi-algorithm HPO tuning job (p. 1029)

Get Started

Using Multi-Algorithm HPO

To use multi-algorithm HPO you must add more than one training definition to your hyperparameter tuning job. Each training definition holds the configuration options for each algorithm you want to try.

In the console, you add training definitions when you create the HPO tuning job by choosing Add training definition, and then following the configuration steps for each algorithm that you want to use.

When you start the configuration steps, please note that the warm start and early stopping features are not available with multi-algorithm HPO. If you want to use these features, you can only tune a single algorithm at a time.

If you’re using an API request, instead of the single TrainingJobDefinition, you must provide a list of training definitions using TrainingJobDefinitions. You must use one or the other, not both.

Managing Hyperparameter Tuning Jobs

You can clone a job, add or edit tags, or create a new hyperparameter tuning job from the console. You can also use the search feature to find jobs by their name, creation time, and status.

Creating a Hyperparameter Tuning Job

To create a new job, open the Amazon SageMaker console, choose Training, choose Hyperparameter tuning jobs, and then choose Create hyperparameter tuning job.

For instructions on using the API to create a tuning job, see Example: Hyperparameter Tuning Job.

Cloning an existing training Job

You can save time by cloning a training job, which copies all of the job’s settings, including data channels, S3 bucket locations, algorithms, and the hyperparameter options.

To clone a training job

- On the Training jobs page or on the Hyperparameter tuning jobs page, choose Actions and then choose Clone.

Editing Tags

You enter tags as key-value pairs. Values are not required. You can use just the key. To see the keys associated with a job, choose the Tags tab on the tuning job’s details page.
Create a new single or multi-algorithm HPO tuning job

Defining job settings

Your tuning job settings are applied across all of the algorithms in the HPO tuning job. Warm start and early stopping are available only when tuning a single algorithm. After you define the job settings you will create individual training definitions for each algorithm or variation you want to tune.

Warm Start

If you cloned this job, you can choose to use the results from a previous tuning job to improve the performance of this tuning job. This is the warm start feature and it is only available when tuning a single algorithm. When you choose this option, you can choose up to five previous hyperparameter tuning jobs to use. Alternatively, you can use transfer learning to add additional data to the parent tuning job. When you select this option, you choose one previous tuning job as the parent.

Warm start is compatible with tuning jobs created after October 1, 2018. For more information, see Run a warm start job.

Early Stopping

Early stopping stops training jobs when they are unlikely to improve the current best objective metric of the hyperparameter tuning job. Like warm start, this feature is only available when tuning a single algorithm. This is an automatic feature without configuration options, and it’s disabled by default.

Tuning Strategy

Tuning strategy can be either random or bayesian. It specifies how the automatic tuning searches over specified hyperparameter ranges. You specify the ranges in a later step. For more information, see How Hyperparameter Tuning Works.

Training Definitions

You must provide at least one training definition for each training job. Each training definition specifies the configuration for an algorithm. To create several definitions for your training job you can clone a definition.

Name

Provide a unique name for the training definition.

Permissions

Amazon SageMaker requires permissions to call other services on your behalf. Choose an IAM role or let AWS create a role that has the AmazonSageMakerFullAccess IAM policy attached.

Optional Security Settings

The network isolation setting prevents the container from making any outbound network calls. This is required for AWS Marketplace machine learning offerings.

You can also choose to use a private VPC.

Note

Inter-container encryption is only available when creating job definitions from the API.

Algorithm Options

You can choose one of the built-in algorithms, your own algorithm, your own container with an algorithm, or you can subscribe to an algorithm from AWS Marketplace.

If you choose a built-in algorithm, it has the ECR image information prepopulated. If you choose your own container, you must specify the ECR image information. You can select the input mode for the
algorithm as file or pipe. If you plan to supply your data using a .CSV file from Amazon S3, you should select the file.

**Metrics**

When you choose a built-in algorithm, metrics are provided for you. If you choose your own algorithm, you need to define your metrics.

**Objective Metric**

To find the best training job, set an objective metric and tuning type. After the training job is complete, you can view the tuning job detail page for a summary of the best training job found using this objective metric.

**Hyperparameter Configuration**

When you choose a built-in algorithm, hyperparameters' default values are set for you, using ranges that are optimized for the particular algorithm. You can change these values as you see fit. Instead of a range, you can set a fixed value for a hyperparameter by setting the parameter's type to **static**. Each algorithm has different required and optional parameters. For more information, see best practices and ranges.

**Input Data Configuration**

Input data is defined by channels, each with their own source location (Amazon S3 or Amazon Elastic File System), compression, and format options. You can define up to 20 channels of input sources. If the algorithm you chose supports multiple input channels, you can specify those too.

For example, when using the XGBoost churn prediction notebook, you could add two channels: train and validation.

**Checkpoint Configuration**

Checkpoints are periodically generated during training. You must choose an Amazon S3 location for the checkpoints to be saved. Checkpoints are used in metrics reporting, and are also used to resume managed spot training jobs.

**Output Data Configuration**

You must define an Amazon S3 location for the artifacts of the training job to be stored. You have the option of adding encryption to the output using an AWS Key Management Service (AWS KMS) key.

**Resource Limits and Configuration**

Each training definition can have a different resource configuration. You choose the instance type and number of nodes.

**Finalizing the Job Settings**

You can run parallel jobs and limit the total number of jobs. The number of parallel jobs should not exceed the number of nodes you have requested across all of your training definitions. The total number of jobs can’t exceed the number of jobs that your definitions are expected to run.

**Example: Hyperparameter Tuning Job**

This example shows how to create a new notebook for configuring and launching a hyperparameter tuning job. The tuning job uses the XGBoost Algorithm (p. 813) to train a model to predict whether a customer will enroll for a term deposit at a bank after being contacted by phone.

You use the low-level AWS SDK for Python (Boto) to configure and launch the hyperparameter tuning job, and the AWS Management Console to monitor the status of hyperparameter tuning jobs. You can also use the Amazon SageMaker high-level Amazon SageMaker Python SDK to configure, run, monitor, and analyze hyperparameter tuning jobs. For more information, see https://github.com/aws/sagemaker-python-sdk.
Prerequisites

To run the code in this example, you need

- An AWS account and an administrator user (p. 33)
- An Amazon S3 bucket for storing your training dataset and the model artifacts created during training (p. 59)
- A running SageMaker notebook instance (p. 60)

Topics

- Create a Notebook (p. 1031)
- Get the Amazon SageMaker Boto 3 Client (p. 1031)
- Get the SageMaker Execution Role (p. 1032)
- Specify a Bucket and Data Output Location (p. 1032)
- Download, Prepare, and Upload Training Data (p. 1032)
- Configure and Launch a Hyperparameter Tuning Job (p. 1033)
- Monitor the Progress of a Hyperparameter Tuning Job (p. 1037)
- Clean up (p. 1039)

Create a Notebook

Create a Jupyter notebook that contains a preinstalled environment with the default Anaconda installation and Python3.

To create a Jupyter notebook

1. Open the Amazon SageMaker console at https://console.aws.amazon.com/sagemaker/.
2. Open a running notebook instance, by choosing Open next to its name. The Jupyter notebook server page appears:

   ![Jupyter Notebook Server](image)

3. To create a notebook, choose Files, New, and conda_python3.
4. Name the notebook.

Next Step

Get the Amazon SageMaker Boto 3 Client (p. 1031)

Get the Amazon SageMaker Boto 3 Client

Import Amazon SageMaker Python SDK, AWS SDK for Python (Boto3), and other Python libraries. In a new Jupyter notebook, paste the following code to the first cell:

```python
import sagemaker
```
import boto3
import numpy as np                                # For performing matrix operations and
       numerical processing
import pandas as pd                               # For manipulating tabular data
from time import gmtime, strftime
import os

region = boto3.Session().region_name
smclient = boto3.Session().client('sagemaker')

The preceding code cell defines `region` and `smclient` objects that you will use to call the built-in
XGBoost algorithm and set the SageMaker hyperparameter tuning job.

Next Step

Get the SageMaker Execution Role (p. 1032)

Get the SageMaker Execution Role

Get the execution role for the notebook instance. This is the IAM role that you created for your notebook instance. You pass the role to the tuning job.

```python
from sagemaker import get_execution_role
role = get_execution_role()
print(role)
```

Next Step

Specify a Bucket and Data Output Location (p. 1032)

Specify a Bucket and Data Output Location

Specify the name of the Amazon S3 bucket where you want to store the output of the training jobs that the tuning job launches. The name of the bucket must contain `sagemaker`, and be globally unique. The bucket must be in the same AWS Region as the notebook instance that you use for this example. You can use the bucket that you created when you set up Amazon SageMaker, or you can create a new bucket. For information, see Step 1: Create an Amazon S3 Bucket (p. 59).

Note
The name of the bucket doesn't need to contain `sagemaker` if the role that you use to run the hyperparameter tuning job has a policy that gives the SageMaker service principle S3FullAccess permission.

```python
sess = sagemaker.Session()
bucket = sess.default_bucket() # Set a default S3 bucket
prefix = 'sagemaker/DEMO-automatic-model-tuning-xgboost-dm'
```

Next Step

Download, Prepare, and Upload Training Data (p. 1032)

Download, Prepare, and Upload Training Data

For this example, you use a training dataset of information about bank customers that includes the customer's job, marital status, and how they were contacted during the bank's direct marketing
campaign. To use a dataset for a hyperparameter tuning job, you download it, transform the data, and then upload it to an Amazon S3 bucket.

For more information about the dataset and the data transformation that the example performs, see the `hpo_xgboost_direct_marketing_sagemaker_APIs` notebook in the Hyperparameter Tuning section of the SageMaker Examples tab in your notebook instance.

**Download and Explore the Training Dataset**

To download and explore the dataset, run the following code in your notebook:

```bash
!unzip -o bank-additional.zip
data = pd.read_csv('./bank-additional/bank-additional-full.csv', sep=';')
pd.set_option('display.max_columns', 500)  # Make sure we can see all of the columns
pd.set_option('display.max_rows', 5)      # Keep the output on one page
data
```

**Prepare and Upload Data**

Before creating the hyperparameter tuning job, prepare the data and upload it to an S3 bucket where the hyperparameter tuning job can access it.

Run the following code in your notebook:

```python
data['no_previous_contact'] = np.where(data['pdays'] == 999, 1, 0)  # Indicator variable to capture when pdays takes a value of 999
data['not_working'] = np.where(np.in1d(data['job'], ['student', 'retired', 'unemployed']), 1, 0)  # Indicator for individuals not actively employed
model_data = pd.get_dummies(data)  # Convert categorical variables to sets of indicators
model_data = model_data.drop(['duration', 'emp.var.rate', 'cons.price.idx', 'cons.conf.idx', 'euribor3m', 'nr.employed'], axis=1)
train_data, validation_data, test_data = np.split(model_data.sample(frac=1, random_state=1729), [int(0.7 * len(model_data)), int(0.9*len(model_data))])
train_data['y'] = train_data['y'].apply(lambda x: int(x))
validation_data['y'] = validation_data['y'].apply(lambda x: int(x))
test_data['y'] = test_data['y'].apply(lambda x: int(x))
train_data.to_csv('train.csv', index=False)
validation_data.to_csv('validation.csv', index=False)
test_data.to_csv('test.csv', index=False)
boto3.Session().resource('s3').Bucket(bucket).Object(os.path.join(prefix, 'train/train.csv')).upload_file('train.csv')
boto3.Session().resource('s3').Bucket(bucket).Object(os.path.join(prefix, 'validation/validation.csv')).upload_file('validation.csv')
```

**Next Step**

Configure and Launch a Hyperparameter Tuning Job (p. 1033)

**Configure and Launch a Hyperparameter Tuning Job**

To configure and launch a hyperparameter tuning job, complete the following steps.

**Topics**

- Specify the Hyperparameter Tuning Job Settings (p. 1034)
Specify the Hyperparameter Tuning Job Settings

To specify settings for the hyperparameter tuning job, you define a JSON object. You pass the object as the value of the HyperParameterTuningJobConfig parameter to CreateHyperParameterTuningJob when you create the tuning job.

In this JSON object, you specify:

- The ranges of hyperparameters that you want to tune. For more information, see Define Hyperparameter Ranges (p. 1026)
- The limits of the resource that the hyperparameter tuning job can consume.
- The objective metric for the hyperparameter tuning job. An objective metric is the metric that the hyperparameter tuning job uses to evaluate the training job that it launches.

**Note**
To use your own algorithm for hyperparameter tuning, you need to define metrics for your algorithm. For information, see Define Metrics (p. 1025).

The hyperparameter tuning job defines ranges for the eta, alpha, min_child_weight, and max_depth hyperparameters of the XGBoost Algorithm (p. 813) built-in algorithm. The objective metric for the hyperparameter tuning job maximizes the validation:auc metric that the algorithm sends to CloudWatch Logs.

```json
tuning_job_config = {
    "ParameterRanges": {
        "CategoricalParameterRanges": [],
        "ContinuousParameterRanges": [
            {
                "MaxValue": "1",
                "MinValue": "0",
                "Name": "eta"
            },
            {
                "MaxValue": "2",
                "MinValue": "0",
                "Name": "alpha"
            },
            {
                "MaxValue": "10",
                "MinValue": "1",
                "Name": "min_child_weight"
            }
        ],
        "IntegerParameterRanges": [
            {
                "MaxValue": "10",
                "MinValue": "1",
                "Name": "max_depth"
            }
        ]
    },
    "ResourceLimits": {
        "MaxNumberOfTrainingJobs": 20,
        "MaxParallelTrainingJobs": 3
    },
    "Strategy": "Bayesian"
}
```
"HyperParameterTuningJobObjective": {  
  "MetricName": "validation:auc",  
  "Type": "Maximize"  
}

Configure the Training Jobs

To configure the training jobs that the tuning job launches, define a JSON object that you pass as the value of the TrainingJobDefinition parameter of the CreateHyperParameterTuningJob call.

In this JSON object, you specify:

- Optional—Metrics that the training jobs emit.

  **Note**
  
  Define metrics only when you use a custom training algorithm. Because this example uses a built-in algorithm, you don't specify metrics. For information about defining metrics, see Define Metrics (p. 1025).

- The container image that specifies the training algorithm.
- The input configuration for your training and test data.
- The storage location for the algorithm's output. Specify the S3 bucket where you want to store the output of the training jobs.
- The values of algorithm hyperparameters that are not tuned in the tuning job.
- The type of instance to use for the training jobs.
- The stopping condition for the training jobs. This is the maximum duration for each training job.

In this example, we set static values for the eval_metric, num_round, objective, rate_drop, and tweedie_variance_power parameters of the XGBoost Algorithm (p. 813) built-in algorithm.

```python
from sagemaker.amazon.amazon_estimator import get_image_uri
training_image = get_image_uri(region, 'xgboost', repo_version='1.0-1')
s3_input_train = 's3://{}/{}\train'.format(bucket, prefix)
s3_input_validation = 's3://{}/{}\validation/'.format(bucket, prefix)

training_job_definition = {
    "AlgorithmSpecification": {
        "TrainingImage": training_image,
        "TrainingInputMode": "File"
    },
    "InputDataConfig": [
    {
        "ChannelName": "train",
        "CompressionType": "None",
        "ContentType": "csv",
        "DataSource": {
            "S3DataSource": {
                "S3DataDistributionType": "FullyReplicated",
                "S3DataType": "S3Prefix",
                "S3Uri": s3_input_train
            }
        }
    },
    {
        "ChannelName": "validation",
        "CompressionType": "None",
```
Example: Hyperparameter Tuning Job

```json
"ContentType": "csv",
"DataSource": {
  "S3DataSource": {
    "S3DataDistributionType": "FullyReplicated",
    "S3DataType": "S3Prefix",
    "S3Uri": s3_input_validation
  }
},
"OutputDataConfig": {
  "S3OutputPath": "s3://{}/{}/output".format(bucket, prefix)
},
"ResourceConfig": {
  "InstanceCount": 2,
  "InstanceType": "ml.c4.2xlarge",
  "VolumeSizeInGB": 10
},
"RoleArn": role,
"StaticHyperParameters": {
  "eval_metric": "auc",
  "num_round": "100",
  "objective": "binary:logistic",
  "rate_drop": "0.3",
  "tweedie_variance_power": "1.4"
},
"StoppingCondition": {
  "MaxRuntimeInSeconds": 43200
}
```

SageMaker Python SDK v2

```python
training_image = sagemaker.image_uris.retrieve('xgboost', region, '1.0-1')
s3_input_train = 's3://{}/{}//train'.format(bucket, prefix)
s3_input_validation = 's3://{}/{}//validation/'.format(bucket, prefix)

training_job_definition = {
  "AlgorithmSpecification": {
    "TrainingImage": training_image,
    "TrainingInputMode": "File"
  },
  "InputDataConfig": [
    {
      "ChannelName": "train",
      "CompressionType": "None",
      "ContentType": "csv",
      "DataSource": {
        "S3DataSource": {
          "S3DataDistributionType": "FullyReplicated",
          "S3DataType": "S3Prefix",
          "S3Uri": s3_input_train
        }
      }
    },
    {
      "ChannelName": "validation",
      "CompressionType": "None",
      "ContentType": "csv",
      "DataSource": {
        "S3DataSource": {
          "S3DataDistributionType": "FullyReplicated",
          "S3DataType": "S3Prefix",
          "S3Uri": s3_input_validation
        }
      }
    }
  ]
}```
Name and Launch the Hyperparameter Tuning Job

Now you can provide a name for the hyperparameter tuning job and then launch it by calling the `CreateHyperParameterTuningJob` API. Pass `tuning_job_config`, and `training_job_definition` that you created in previous steps as the values of the parameters.

```python
tuning_job_name = "MyTuningJob"
smclient.create_hyper_parameter_tuning_job(HyperParameterTuningJobName = tuning_job_name,
                                        HyperParameterTuningJobConfig = tuning_job_config,
                                        TrainingJobDefinition = training_job_definition)
```

Next Step

Monitor the Progress of a Hyperparameter Tuning Job (p. 1037)

Monitor the Progress of a Hyperparameter Tuning Job

To monitor the progress of a hyperparameter tuning job and the training jobs that it launches, use the Amazon SageMaker console.

Topics

- View the Status of the Hyperparameter Tuning Job (p. 1037)
- View the Status of the Training Jobs (p. 1038)
- View the Best Training Job (p. 1039)

View the Status of the Hyperparameter Tuning Job

To view the status of the hyperparameter tuning job

1. Open the Amazon SageMaker console at https://console.aws.amazon.com/sagemaker/
2. Choose Hyperparameter tuning jobs.
3. In the list of hyperparameter tuning jobs, check the status of the hyperparameter tuning job you launched. A tuning job can be:

- **Completed**—The hyperparameter tuning job successfully completed.
- **InProgress**—The hyperparameter tuning job is in progress. One or more training jobs are still running.
- **Failed**—The hyperparameter tuning job failed.
- **Stopped**—The hyperparameter tuning job was manually stopped before it completed. All training jobs that the hyperparameter tuning job launched are stopped.
- **Stopping**—The hyperparameter tuning job is in the process of stopping.

### View the Status of the Training Jobs

#### To view the status of the training jobs that the hyperparameter tuning job launched

1. In the list of hyperparameter tuning jobs, choose the job that you launched.
2. Choose **Training jobs**.

3. View the status of each training job. To see more details about a job, choose it in the list of training jobs. To view a summary of the status of all of the training jobs that the hyperparameter tuning job launched, see **Training job status counter**.
A training job can be:

- **Completed**—The training job successfully completed.
- **InProgress**—The training job is in progress.
- **Stopped**—The training job was manually stopped before it completed.
- **Failed (Retriable)**—The training job failed, but can be retried. A failed training job can be retried only if it failed because an internal service error occurred.
- **Failed (Non-retriable)**—The training job failed and can't be retried. A failed training job can't be retried when a client error occurs.

**View the Best Training Job**

A hyperparameter tuning job uses the objective metric that each training job returns to evaluate training jobs. While the hyperparameter tuning job is in progress, the best training job is the one that has returned the best objective metric so far. After the hyperparameter tuning job is complete, the best training job is the one that returned the best objective metric.

To view the best training job, choose **Best training job**.

To deploy the best training job as a model that you can host at a SageMaker endpoint, choose **Create model**.

**Next Step**

**Clean up (p. 1039)**

**Clean up**

To avoid incurring unnecessary charges, when you are done with the example, use the AWS Management Console to delete the resources that you created for it.

**Note**

If you plan to explore other examples, you might want to keep some of these resources, such as your notebook instance, S3 bucket, and IAM role.
1. Open the SageMaker console at https://console.aws.amazon.com/sagemaker/ and delete the notebook instance. Stop the instance before deleting it.
2. Open the Amazon S3 console at https://console.aws.amazon.com/s3/ and delete the bucket that you created to store model artifacts and the training dataset.
3. Open the IAM console at https://console.aws.amazon.com/iam/ and delete the IAM role. If you created permission policies, you can delete them, too.
4. Open the Amazon CloudWatch console at https://console.aws.amazon.com/cloudwatch/ and delete all of the log groups that have names starting with /aws/sagemaker/.

Stop Training Jobs Early

Stop the training jobs that a hyperparameter tuning job launches early when they are not improving significantly as measured by the objective metric. Stopping training jobs early can help reduce compute time and helps you avoid overfitting your model. To configure a hyperparameter tuning job to stop training jobs early, do one of the following:

- If you are using the AWS SDK for Python (Boto 3), set the TrainingJobEarlyStoppingType field of the HyperParameterTuningJobConfig object that you use to configure the tuning job to AUTO.
- If you are using the Amazon SageMaker Python SDK, set the early_stopping_type parameter of the HyperParameterTuner object to Auto.
- In the Amazon SageMaker console, in the Create hyperparameter tuning job workflow, under Early stopping, choose Auto.

For a sample notebook that demonstrates how to use early stopping, see https://github.com/awslabs/amazon-sagemaker-examples/blob/master/hyperparameter_tuning/image_classification_early_stopping/hpo_image_classification_early_stopping.ipynb or open the hpo_image_classification_early_stopping.ipynb notebook in the Hyperparameter Tuning section of the SageMaker Examples in a notebook instance. For information about using sample notebooks in a notebook instance, see Example Notebooks (p. 134).

How EarlyStopping Works

When you enable early stopping for a hyperparameter tuning job, SageMaker evaluates each training job the hyperparameter tuning job launches as follows:

- After each epoch of training, get the value of the objective metric.
- Compute the running average of the objective metric for all previous training jobs up to the same epoch, and then compute the median of all of the running averages.
- If the value of the objective metric for the current training job is worse (higher when minimizing or lower when maximizing the objective metric) than the median value of running averages of the objective metric for previous training jobs up to the same epoch, SageMaker stops the current training job.

Algorithms That Support Early Stopping

To support early stopping, an algorithm must emit objective metrics for each epoch. The following built-in SageMaker algorithms support early stopping:

- Linear Learner Algorithm (p. 732)—Supported only if you use objective_loss as the objective metric.
- XGBoost Algorithm (p. 813)
- Image Classification Algorithm (p. 690)
• Object Detection Algorithm (p. 770)
• Sequence-to-Sequence Algorithm (p. 801)
• IP Insights (p. 701)

Note
This list of built-in algorithms that support early stopping is current as of December 13, 2018. Other built-in algorithms might support early stopping in the future. If an algorithm emits a metric that can be used as an objective metric for a hyperparameter tuning job (preferably a validation metric), then it supports early stopping.

To use early stopping with your own algorithm, you must write your algorithms such that it emits the value of the objective metric after each epoch. The following list shows how you can do that in different frameworks:

TensorFlow
Use the `tf.keras.callbacks.ProgbarLogger` class. For information, see https://www.tensorflow.org/api_docs/python/tf/keras/callbacks/ProgbarLogger.

MXNet
Use the `mxnet.callback.LogValidationMetricsCallback`. For information, see https://mxnet.apache.org/api/python/callback/callback.html.

Chainer
Extend chainer by using the `extensions.Evaluator` class. For information, see https://docs.chainer.org/en/v1.24.0/reference/extensions.html#evaluator.

PyTorch and Spark
There is no high-level support. You must explicitly write your training code so that it computes objective metrics and writes them to logs after each epoch.

Run a Warm Start Hyperparameter Tuning Job

Use warm start to start a hyperparameter tuning job using one or more previous tuning jobs as a starting point. The results of previous tuning jobs are used to inform which combinations of hyperparameters to search over in the new tuning job. Hyperparameter tuning uses either Bayesian or random search to choose combinations of hyperparameter values from ranges that you specify. For more information, see How Hyperparameter Tuning Works (p. 1024). Using information from previous hyperparameter tuning jobs can help increase the performance of the new hyperparameter tuning job by making the search for the best combination of hyperparameters more efficient.

Note
Warm start tuning jobs typically take longer to start than standard hyperparameter tuning jobs, because the results from the parent jobs have to be loaded before the job can start. The increased time depends on the total number of training jobs launched by the parent jobs.

Reasons you might want to consider warm start include:

• You want to gradually increase the number of training jobs over several tuning jobs based on the results you see after each iteration.
• You get new data, and want to tune a model using the new data.
• You want to change the ranges of hyperparameters that you used in a previous tuning job, change static hyperparameters to tunable, or change tunable hyperparameters to static values.
• You stopped a previous hyperparameter job early or it stopped unexpectedly.

**Topics**

- Types of Warm Start Tuning Jobs (p. 1042)
- Warm Start Tuning Restrictions (p. 1042)
- Warm Start Tuning Sample Notebook (p. 1043)
- Create a Warm Start Tuning Job (p. 1043)

**Types of Warm Start Tuning Jobs**

There are two different types of warm start tuning jobs:

**IDENTICAL_DATA_AND_ALGORITHM**

The new hyperparameter tuning job uses the same input data and training image as the parent tuning jobs. You can change the hyperparameter ranges to search and the maximum number of training jobs that the hyperparameter tuning job launches. You can also change hyperparameters from tunable to static, and from static to tunable, but the total number of static plus tunable hyperparameters must remain the same as it is in all parent jobs. You cannot use a new version of the training algorithm, unless the changes in the new version do not affect the algorithm itself. For example, changes that improve logging or adding support for a different data format are allowed.

Use identical data and algorithm when you use the same training data as you used in a previous hyperparameter tuning job, but you want to increase the total number of training jobs or change ranges or values of hyperparameters.

When you run an warm start tuning job of type **IDENTICAL_DATA_AND_ALGORITHM**, there is an additional field in the response to DescribeHyperParameterTuningJob named OverallBestTrainingJob. The value of this field is the TrainingJobSummary for the training job with the best objective metric value of all training jobs launched by this tuning job and all parent jobs specified for the warm start tuning job.

**TRANSFER_LEARNING**

The new hyperparameter tuning job can include input data, hyperparameter ranges, maximum number of concurrent training jobs, and maximum number of training jobs that are different than those of its parent hyperparameter tuning jobs. You can also change hyperparameters from tunable to static, and from static to tunable, but the total number of static plus tunable hyperparameters must remain the same as it is in all parent jobs. The training algorithm image can also be a different version from the version used in the parent hyperparameter tuning job. When you use transfer learning, changes in the dataset or the algorithm that significantly affect the value of the objective metric might reduce the usefulness of using warm start tuning.

**Warm Start Tuning Restrictions**

The following restrictions apply to all warm start tuning jobs:

• A tuning job can have a maximum of 5 parent jobs, and all parent jobs must be in a terminal state (Completed, Stopped, or Failed) before you start the new tuning job.

• The objective metric used in the new tuning job must be the same as the objective metric used in the parent jobs.

• The total number of static plus tunable hyperparameters must remain the same between parent jobs and the new tuning job. Because of this, if you think you might want to use a hyperparameter...
Run a Warm Start Hyperparameter Tuning Job

As tunable in a future warm start tuning job, you should add it as a static hyperparameter when you create a tuning job.

- The type of each hyperparameter (continuous, integer, categorical) must not change between parent jobs and the new tuning job.
- The number of total changes from tunable hyperparameters in the parent jobs to static hyperparameters in the new tuning job, plus the number of changes in the values of static hyperparameters cannot be more than 10. Each value in a categorical hyperparameter counts against this limit. For example, if the parent job has a tunable categorical hyperparameter with the possible values red and blue, you change that hyperparameter to static in the new tuning job, that counts as 2 changes against the allowed total of 10. If the same hyperparameter had a static value of red in the parent job, and you change the static value to blue in the new tuning job, it also counts as 2 changes.
- Warm start tuning is not recursive. For example, if you create MyTuningJob3 as a warm start tuning job with MyTuningJob2 as a parent job, and MyTuningJob2 is itself a warm start tuning job with a parent job MyTuningJob1, the information that was learned when running MyTuningJob1 is not used for MyTuningJob3. If you want to use the information from MyTuningJob1, you must explicitly add it as a parent for MyTuningJob3.
- The training jobs launched by every parent job in a warm start tuning job count against the 500 maximum training jobs for a tuning job.
- Hyperparameter tuning jobs created before October 1, 2018 cannot be used as parent jobs for warm start tuning jobs.

Warm Start Tuning Sample Notebook

For a sample notebook that shows how to use warm start tuning, see https://github.com/awslabs/amazon-sagemaker-examples/blob/master/hyperparameter_tuning/image_classification_warmstart/hpo_image_classification_warmstart.ipynb. For instructions how to create and access Jupyter notebook instances that you can use to run the example in SageMaker, see Example Notebooks (p. 134). Once you have created a notebook instance and opened it, select the SageMaker Examples tab to see a list of all the SageMaker samples. The warm start tuning example notebook is located in the Hyperparameter tuning section, and is named hpo_image_classification_warmstart.ipynb. To open a notebook, click on its Use tab and select Create copy.

Create a Warm Start Tuning Job

You can use either the low-level AWS SDK for Python (Boto 3) or the high-level SageMaker Python SDK to create a warm start tuning job.

Topics
- Create a Warm Start Tuning Job (Low-level SageMaker API for Python (Boto 3)) (p. 1043)
- Create a Warm Start Tuning Job (SageMaker Python SDK) (p. 1044)

Create a Warm Start Tuning Job (Low-level SageMaker API for Python (Boto 3))

To use warm start tuning, you specify the values of a HyperParameterTuningJobWarmStartConfig object, and pass that as the WarmStartConfig field in a call to CreateHyperParameterTuningJob.

The following code shows how to create a HyperParameterTuningJobWarmStartConfig object and pass it to CreateHyperParameterTuningJob job by using the low-level SageMaker API for Python (Boto 3).

Create the HyperParameterTuningJobWarmStartConfig object:

```python
warm_start_config = {
```
Create the warm start tuning job:

```python
smclient = boto3.Session().client('sagemaker')
smclient.create_hyper_parameter_tuning_job(HyperParameterTuningJobName =
  'MyWarmStartTuningJob',
  HyperParameterTuningJobConfig = tuning_job_config, # See notebook for tuning
  configuration
  TrainingJobDefinition = training_job_definition, # See notebook for job definition
  WarmStartConfig = warm_start_config)
```

Create a Warm Start Tuning Job (SageMaker Python SDK)

To use the Amazon SageMaker Python SDK to run a warm start tuning job, you:

- Specify the parent jobs and the warm start type by using a WarmStartConfig object.
- Pass the WarmStartConfig object as the value of the warm_start_config argument of a
  HyperparameterTuner object.
- Call the fit method of the HyperparameterTuner object.

For more information about using the Amazon SageMaker Python SDK for hyperparameter tuning, see

This example uses an estimator that uses the Image Classification Algorithm (p. 690) algorithm for
training. The following code sets the hyperparameter ranges that the warm start tuning job searches
within to find the best combination of values. For information about setting hyperparameter ranges, see
Define Hyperparameter Ranges (p. 1026).

```python
hyperparameter_ranges = {'learning_rate': ContinuousParameter(0.0, 0.1),
                        'momentum': ContinuousParameter(0.0, 0.99)}
```

The following code configures the warm start tuning job by creating a WarmStartConfig object.

```python
from sagemaker.tuner import WarmStartConfig,
    WarmStartTypes

parent_tuning_job_name = "MyParentTuningJob"
warm_start_config = WarmStartConfig(type=WarmStartTypes.IDENTICAL_DATA_AND_ALGORITHM,
    parents={parent_tuning_job_name})
```

Now set the values for static hyperparameters, which are hyperparameters that keep the same
value for every training job that the warm start tuning job launches. In the following code,
imageclassification is an estimator that was created previously.

```python
imageclassification.set_hyperparameters(num_layers=18,
    num_classes=128,
    num_training_samples=15420,
    mini_batch_size=128,
    epochs=30,
    optimizer='sgd',
```
top_k='2',
precision_dtype='float32',
augmentation_type='crop')

Now create the HyperparameterTuner object and pass the WarmStartConfig object that you previously created as the warm_start_config argument.

tuner_warm_start = HyperparameterTuner(imageclassification,
   'validation:accuracy',
   hyperparameter_ranges,
   objective_type='Maximize',
   max_jobs=10,
   max_parallel_jobs=2,
   base_tuning_job_name='warmstart',
   warm_start_config=warm_start_config)

Finally, call the fit method of the HyperparameterTuner object to launch the warm start tuning job.

tuner_warm_start.fit(
   {'train': s3_input_train, 'validation': s3_input_validation},
   include_cls_metadata=False)

Automatic Model Tuning Resource Limits

SageMaker sets default limits for the following resources:

- Number of concurrent hyperparameter tuning jobs - 100
- Number of hyperparameters that can be searched - 20

**Note**
Every possible value in a categorical hyperparameter counts against this limit.

- Number of metrics defined per hyperparameter tuning job - 20
- Number of concurrent training jobs per hyperparameter tuning job - 10
- Number of training jobs per hyperparameter tuning job - 500
- Maximum run time for a hyperparameter tuning job - 30 days

When you plan hyperparameter tuning jobs, you also have to take the limits on training resources into account. For information about the default resource limits for SageMaker training jobs, see [SageMaker Limits](#). Every concurrent training instance that all of your hyperparameter tuning jobs run on count against the total number of training instances allowed. For example, suppose you run 10 concurrent hyperparameter tuning jobs. Each of those hyperparameter tuning jobs runs 100 total training jobs, and runs 20 concurrent training jobs. Each of those training jobs runs on one ml.m4.xlarge instance. The following limits apply:

- Number of concurrent hyperparameter tuning jobs - You don't need to increase the limit, because 10 tuning jobs is below the limit of 100.
- Number of training jobs per hyperparameter tuning job - You don't need to increase the limit, because 100 training jobs is below the limit of 500.
- Number of concurrent training jobs per hyperparameter tuning job - You need to request a limit increase to 20, because the default limit is 10.
- SageMaker training ml.m4.xlarge instances - You need to request limit increase to 200, because you have 10 hyperparameter tuning jobs, with each of them running 20 concurrent training jobs. The default limit is 20 instances.
• SageMaker training total instance count - You need to request a limit increase to 200, because you have 10 hyperparameter tuning jobs, with each of them running 20 concurrent training jobs. The default limit is 20 instances.

For information about requesting limit increases for AWS resources, see AWS Service Limits.

Best Practices for Hyperparameter Tuning

Hyperparameter optimization is not a fully-automated process. To improve optimization, use the following guidelines when you create hyperparameters.

Topics

• Choosing the Number of Hyperparameters (p. 1046)
• Choosing Hyperparameter Ranges (p. 1046)
• Using Logarithmic Scales for Hyperparameters (p. 1046)
• Choosing the Best Number of Concurrent Training Jobs (p. 1046)
• Running Training Jobs on Multiple Instances (p. 1046)

Choosing the Number of Hyperparameters

The computational complexity of a hyperparameter tuning job depends primarily on the number of hyperparameters whose range of values Amazon SageMaker has to search through during optimization. Although you can simultaneously specify up to 20 hyperparameters to optimize for a tuning job, limiting your search to a much smaller number is likely to give you better results.

Choosing Hyperparameter Ranges

The range of values for hyperparameters that you choose to search can significantly affect the success of hyperparameter optimization. Although you might want to specify a very large range that covers every possible value for a hyperparameter, you get better results by limiting your search to a small range of values. If you know that you get the best metric values within a subset of the possible range, consider limiting the range to that subset.

Using Logarithmic Scales for Hyperparameters

During hyperparameter tuning, SageMaker attempts to figure out if your hyperparameters are log-scaled or linear-scaled. Initially, it assumes that hyperparameters are linear-scaled. If they are in fact log-scaled, it might take some time for SageMaker to discover that fact. If you know that a hyperparameter is log-scaled and can convert it yourself, doing so could improve hyperparameter optimization.

Choosing the Best Number of Concurrent Training Jobs

When setting the resource limit `MaxParallelTrainingJobs` for the maximum number of concurrent training jobs that a hyperparameter tuning job can launch, consider the following tradeoff. Running more hyperparameter tuning jobs concurrently gets more work done quickly, but a tuning job improves only through successive rounds of experiments. Typically, running one training job at a time achieves the best results with the least amount of compute time.

Running Training Jobs on Multiple Instances

When a training job runs on multiple instances, hyperparameter tuning uses the last-reported objective metric value from all instances of that training job as the value of the objective metric for that training job. Design distributed training jobs so that the objective metric reported is the one that you want.
Distributed Training

Topics
- Get Started with Distributed Training (p. 1047)
- Basic Distributed Training Concepts (p. 1047)
- Advanced Concepts (p. 1048)
- Strategies (p. 1049)
- Optimize Distributed Training (p. 1050)
- Scenarios (p. 1051)
- SageMaker Built-In Distributed Training Features (p. 1054)
- SageMaker Distributed Data Parallel (p. 1054)
- SageMaker Distributed Model Parallel (p. 1066)
- Distributed Training Jupyter Notebook Examples (p. 1083)
- Troubleshooting (p. 1084)

Get Started with Distributed Training

If you're familiar with distributed training, follow one of the links to your preferred strategy or framework to get started. Otherwise, continue on to the next section to learn some distributed training concepts.

SageMaker distributed training libraries:
- SageMaker Distributed Data Parallel (p. 1054)
- SageMaker Distributed Model Parallel (p. 1066)

Basic Distributed Training Concepts

SageMaker’s distributed training libraries use the following distributed training terms and features.

**Batch size**: The number of records from the dataset selected for each interval to send to the GPUs in the cluster. In an image example, if you send out 32 images per GPU, your batch size is 32. Advanced implementations provide some guidance on optimizing batch size in a warm-up period and throughout training. These suggestions vary depending on the data you are using. For example, how you optimize for image data will differ from how you handle language data.

**Cluster size**: The number of instances multiplied by the number of GPUs in each instance. This is the amount of raw processing power you use to train a model in a single training job. The number of GPUs you bring to bear on a training job has the most impact on how long it takes to train a model. However, just throwing processing power at the problem is not enough. Communication between the instances (usually referred to as nodes in a cluster) can create so much overhead that the training time is worse than if you had fewer nodes. This is where SageMaker distributed training and other advanced distributed training frameworks step in with optimized communication solutions.

**Data parallel**: A strategy in distributed training where the dataset is split up across multiple processing nodes. Each node sees different batches from the dataset, makes its calculations, and shares them back to the other nodes for synchronization before moving on to the next batch and ultimately another epoch.

**Dataset**: All of the data you're training. The dataset is often split into subsets: one for training and one for validation. For example, you might split the dataset and use 80% of it for training (the training set),
then use the remaining 20% to test the current accuracy of the model (the validation set) while it is still training.

**Device batch**: The number of records seen by a device during an iteration. This is a fraction of the mini-batch.

**Epoch**: One cycle through the entire dataset. Multiple intervals complete a batch, and multiple batches eventually complete an epoch. Multiple epochs are run until the accuracy of the model reaches an acceptable level—or, looked at another way, when the error rate drops below an acceptable level.

**Interval or iteration**: A distribution of a batch of data. A large amount of data must be broken up into batches and sent to the GPUs in mini-batches. The number of intervals is equal to the number of batches it takes to get through the entire dataset.

**Learning rate**: The amount that values should be changed between epochs. As the model is refined, its internal weights are nudged and error rates are checked to see if the model improves. The optimal learning rate for a specific use case depends on model, number of GPUs used during training, and other hyperparameters. Often, learning rates are set to 0.1 or 0.01; 0.01 is a much smaller adjustment and could cause the training to take a long time to converge, whereas 0.1 is much larger and can cause the training to overshoot. It is one of the primary hyperparameters that you can adjust for training your model. You may also want to try a warm-up period in which you set an initial low learning rate, so that the neural net has a chance to stabilize newly initialized layers, and with each subsequent epoch you gradually increase the learning rate.

**Mini-batch size**: The number of records over which the model error gradient is estimated at each iteration. It is the number of GPUs times the batch size. For example, if batch size is 32 and you have 256 GPUs, your mini-batch size is 8,192. Changing the mini-batch size will generally change the performance of your model and the required time-to-results for a given performance goal.

**Microbatch**: A smaller subset of a given training mini-batch. Model parallel pipelining is based on splitting a mini-batch into microbatches, which are fed into the training pipeline one-by-one and follow an execution schedule defined by SMP runtime.

**Model parallel**: A strategy in distributed training in which the model is split up across multiple processing nodes. The model may be very complex and have a huge number of layers and weights inside it, making it unable to fit in the memory of a single node. Each node carries a subset of the model, through which the data flows and the transformations are shared and compiled.

**Pipeline Execution Schedule (Pipelining)**: This defines pipelined execution when training a model with model parallelism. The pipeline execution schedule determines the order in which computations are made and data is processed across devices during model training. Pipelining is a technique to achieve true parallelization in model parallelism and overcome the performance loss due to sequential computation by having the GPUs compute simultaneously on different data samples.

### Advanced Concepts

Machine Learning (ML) practitioners commonly face two scaling challenges when training models: *scaling model size* and *scaling training data*. While model size and complexity can result in better accuracy, there is a limit to the model size you can fit into a single CPU or GPU. Furthermore, scaling model size may result in more computations and longer training times.

Not all models handle training data scaling equally well. Some algorithms need to ingest all the training data in memory for training. They only scale vertically, and to bigger and bigger instance types. Some algorithms have the disadvantage of scaling super-linearly with dataset size. For example, the computational load of a nearest-neighbor search for all records of an N-rows dataset scales in N squared. In most cases, scaling training data results in longer training times.

Deep Learning (DL) is a specific family of ML algorithms consisting of several layers of artificial neural networks. The most common training method is with mini-batch Stochastic Gradient Descent (SGD).
In mini-batch SGD, the model is trained by conducting small iterative changes of its coefficients in the direction that reduces its error. Those iterations are conducted on equally sized subsamples of the training dataset called *mini-batches*. For each mini-batch, the model is run in each record of the mini-batch, its error measured and the gradient of the error estimated. Then the average gradient is measured across all the records of the mini-batch and provides an update direction for each model coefficient. One full pass over the training dataset is called an *epoch*. Model trainings commonly consist of dozens to hundreds of epochs. Mini-batch SGD has several benefits: First, its iterative design makes training time theoretically linear of dataset size. Second, in a given mini-batch each record is processed individually by the model without need for inter-record communication other than the final gradient average. The processing of a mini-batch is consequently particularly suitable for parallelization and distribution.

Parallelizing SGD training by distributing the records of a mini-batch over different computing devices is called *data parallel distributed training*, and is the most commonly used DL distribution paradigm. Data parallel training is a relevant distribution strategy to scale the mini-batch size and process each mini-batch faster. However, data parallel training comes with the extra complexity of having to compute the mini-batch gradient average with gradients coming from all the workers and communicating it to all the workers, a step called *allreduce* that can represent a growing overhead, as the training cluster is scaled, and that can also drastically penalize training time if improperly implemented or implemented over improper hardware substracts.

Data parallel SGD still requires developers to be able to fit at least the model and a single record in a computing devices, like a single CPU or GPU. When training very large models such as large transformers in Natural Language Processing (NLP), or segmentation models over high-resolution images, there may be situations in which this is not feasible. An alternative way to break up the workload is to partition the model over multiple computing devices, an approach called *model-parallel distributed training*.

**Strategies**

Distributed training is usually split by two approaches: data parallel and model parallel. *Data parallel* is the most common approach to distributed training: You have a lot of data, batch it up, and send blocks of data to multiple CPUs or GPUs (nodes) to be processed by the neural network or ML algorithm, then combine the results. The neural network is the same on each node. A *model parallel* approach is used with large models that won’t fit in a node’s memory in one piece; it breaks up the model and places different parts on different nodes. In this situation, you need to send your batches of data out to each node so that the data is processed on all parts of the model.

The terms *network* and *model* are often used interchangeably: A large model is really a large network with many layers and parameters. Training with a large network produces a large model, and loading the model back onto the network with all your pre-trained parameters and their weights loads a large model into memory. When you break apart a model to split it across nodes, you’re also breaking apart the underlying network. A network consists of layers, and to split up the network, you put layers on different compute devices.

A common pitfall of naively splitting layers across devices is severe GPU under-utilization. Training is inherently sequential in both forward and backward passes, and at a given time, only one GPU can actively compute, while the others wait on the activations to be sent. Modern model parallel libraries solve this problem by using pipeline execution schedules to improve device utilization. However, only the SageMaker Distributed Model Parallel library (SMP) includes automatic model splitting. The two core features of SMP, automatic model splitting and pipeline execution scheduling, simplifies the process of implementing model parallelism by making automated decisions that lead to efficient device utilization.

**Which Is “Better”: Data Parallel or Model Parallel?**

Despite their names, data parallel training is truly parallel computing, whereas model-parallel training is more akin to serialized computing. If your model can fit into memory, you can use data parallel training, and you should, as training is faster. However, this isn’t always possible, and so for large models, you
must use model-parallel training. There are parts of a model-parallel flow that can be parallelized, but some parts of the network bottleneck the others and require you to wait for the network to finish processing data in a serial fashion. However, SageMaker’s model parallelism library uses sophisticated pipelining, enabling the library to parallelize work by having devices compute on different microbatches at the same time. This approaches true parallelism, even if the underlying model consists of sequential, non-parallelizable computational steps.

**Choosing Between Data Parallel and Model Parallel**

Start with a data parallel approach. If you run out of memory during training, you may want to switch to a model parallel approach. However, consider these alternatives before trying model-parallel training:

- Change your model’s hyperparameters.
- Reduce the batch size.
- Keep reducing the batch size until it fits. If you reduce batch size to 1, and still run out of memory, then you should try model-parallel training.

Try gradient compression (fp16, fp8):

- On NVIDIA TensorCore-equipped hardware, using mixed-precision training creates both speed-up and memory consumption reduction.

Try reducing the input size:

- Reduce the NLP sequence length if you increase the sequence link, need to adjust the batch size down, or adjust the GPUs up to spread the batch.
- Reduce image resolution.

Check if you use batch normalization, since this can impact convergence. When you use distributed training, your batch is split across GPUs and the effect of a much lower batch size can be a higher error rate thereby disrupting the model from converging. For example, if you prototyped your network on a single GPU with a batch size of 64, then scaled up to using four p3dn.24xlarge, you now have 32 GPUs and your per-GPU batch size drops from 64 to 2. This will likely break the convergence you saw with a single node.

Start with model-parallel training when:

- Your model does not fit on a single device.
- Due to your model size, you’re facing limitations in choosing larger batch sizes, such as if your model weights take up most of your GPU memory and you are forced to choose a smaller, suboptimal batch size.

To learn more about the SageMaker distributed libraries, see the following:

- SageMaker Distributed Data Parallel (p. 1054)
- SageMaker Distributed Model Parallel (p. 1066)

**Optimize Distributed Training**

Customize hyperparameters for your use case and your data to get the best scaling efficiency. In the following discussion, we highlight some of the most impactful training variables and provide references to state-of-the-art implementations so you can learn more about your options. Also, we recommend that you refer to your preferred framework’s distributed training documentation.
Batch Size

SageMaker distributed toolkits generally allow you to train on bigger batches. For example, if a model fits within a single device but can only be trained with a small batch size, using either model-parallel training or data parallel training enables you to experiment with larger batch sizes.

Be aware that batch size directly influences model accuracy by controlling the amount of noise in the model update at each iteration. Increasing batch size reduces the amount of noise in the gradient estimation, which can be beneficial when increasing from very small batches sizes, but can result in degraded model accuracy as the batch size increases to large values.

Tip
Adjust your hyperparameters to ensure that your model trains to a satisfying convergence as you increase its batch size.

A number of techniques have been developed to maintain good model convergence when batch is increased.

To learn more about these techniques, see the following papers:

- Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour, Goya et al.
- PowerAI DDL, Cho et al.
- Scale Out for Large Minibatch SGD: Residual Network Training on ImageNet-1K with Improved Accuracy and Reduced Time to Train, Codreanu et al.
- ImageNet Training in Minutes, You et al.
- Large Batch Training of Convolutional Networks, You et al.
- Large Batch Optimization for Deep Learning: Training BERT in 76 Minutes, You et al.
- Accelerated Large Batch Optimization of BERT Pretraining in 54 minutes, Zheng et al.
- Deep Gradient Compression, Lin et al.

Mini-Batch Size

In SGD, the mini-batch size quantifies the amount of noise present in the gradient estimation. A small mini-batch results in a very noisy mini-batch gradient, which is not representative of the true gradient over the dataset. A large mini-batch results in a mini-batch gradient close to the true gradient over the dataset and potentially not noisy enough—likely to stay locked in irrelevant minima.

Scenarios

The following sections cover scenarios in which you may want to scale up training, and how you can do so using AWS resources.

Scaling from a Single GPU to Many GPUs

The amount of data or the size of the model used in machine learning can create situations in which the time to train a model is longer that you are willing to wait. Sometimes, the training doesn’t work at all because the model or the training data is too large. One solution is to increase the number of GPUs you
use for training. On an instance with multiple GPUs, like a p3.16xlarge that has eight GPUs, the data and processing is split across the eight GPUs, and producing a near-linear speedup in the time it takes to train your model. It takes slightly over 1/8 the time it would have taken on p3.2xlarge with one GPU.

<table>
<thead>
<tr>
<th>Instance type</th>
<th>GPUs</th>
</tr>
</thead>
<tbody>
<tr>
<td>p3.2xlarge</td>
<td>1</td>
</tr>
<tr>
<td>p3.8xlarge</td>
<td>4</td>
</tr>
<tr>
<td>p3.16xlarge</td>
<td>8</td>
</tr>
<tr>
<td>p3dn.24xlarge</td>
<td>8</td>
</tr>
</tbody>
</table>

**Scaling from a Single Instance to Multiple Instances**

If you want to scale your training even further, you can use more instances. However, you should choose a larger instance type before you add more instances. Review the previous table to see how many GPUs are in each p3 instance type.

If you have made the jump from a single GPU on a p3.2xlarge to four GPUs on a p3.8xlarge, but decide that you require more processing power, you should always choose a p3.16xlarge before trying to increase instance count. When you keep your training on a single instance, performance is better than when you use multiple instances. Also, your costs are lower.

When you are ready to scale the number of instances, you can do this with SageMaker PythonSDK’s estimator function by setting your instance_type = p3.16xlarge and instance_count = 2. Instead of the eight GPUs on a single p3.16xlarge, you have 16 GPUs across two identical instances. The following chart shows scaling and throughput starting with eight GPUs on a single instance and increasing to 64 instances for a total of 256 GPUs.
Availability Zones and Network Backplane

With multiple instances, it's important to understand the network that connects the instances, how they read the training data, and how they share information between themselves (for example, communication between the nodes in the cluster when doing an AllReduce operation).

First, your instances need to be in the same Region and same Availability Zone. For example, instances in us-west-2 must all be in us-west-2a. When you use the SageMaker Python SDK, this is handled for you. If you use Amazon EC2 and orchestrate your own training clusters, you need to be aware of this, or your training speeds suffer.

Your training data should also be in the same availability zone. When you use a SageMaker estimator, you pass in the Region and the S3 bucket, and if the data is not in the Region you set, you get an error.

Optimized GPU, Network, and Storage

The p3dn.24xlarge instance type was designed for fast local storage and a fast network backplane with up to 100 gigabits, and we highly recommend it as the most performant option for distributed training. SageMaker supports streaming data modes from S3, referred to as pipe mode. For HPC loads like distributed training, we recommend Amazon Fsx for your file storage.

Custom Training Scripts

While SageMaker makes it simple to deploy and scale the number of instances and GPUs, depending on your framework of choice, managing the data and results can be very challenging, which is why external supporting libraries are often used. This most basic form of distributed training requires modification of your training script to manage the data distribution.

SageMaker also supports Horovod and implementations of distributed training native to each major deep learning framework. If you choose to use examples from these frameworks, you can follow SageMaker's container guide for Deep Learning Containers, and various example notebooks that demonstrate implementations.

SageMaker Built-In Distributed Training Features

The SageMaker built-in libraries of algorithms consists of 18 popular machine learning algorithms. Many of them were rewritten from scratch to be scalable and distributed out of the box. If you want to use distributed deep learning training code, we recommend Amazon SageMaker's distributed training libraries. SageMaker's distributed training libraries make it easier for you to write highly scalable and cost-effective custom data parallel and model parallel deep learning training jobs.

SageMaker distributed training libraries offer both data parallel and model parallel training strategies. It combines software and hardware technologies to improve inter-GPU and inter-node communications. It extends SageMaker's training capabilities with built-in options that require only small code changes to your training scripts.

- SageMaker Distributed Data Parallel (p. 1054)
- SageMaker Distributed Model Parallel (p. 1066)

SageMaker Distributed Data Parallel

Important
To use new features with an existing notebook instance or Studio application, restart the notebook instance or Studio application to get the latest updates.
SageMaker distributed data parallel (SDP) extends SageMaker’s training capabilities on deep learning models with near-linear scaling efficiency, achieving fast time-to-train with minimal code changes.

- SDP optimizes your training job for AWS network infrastructure and Amazon EC2 instance topology.
- SDP takes advantage of gradient updates to communicate between nodes with a custom AllReduce algorithm.

When training a model on a large amount of data, machine learning practitioners often turn to distributed training to reduce the time to train. In some cases, where time is of the essence, the business requirement is to finish training as quickly as possible or at least within a constrained time period. Then, distributed training is scaled to use a cluster of multiple nodes—not just multiple GPUs in a computing instance, but multiple instances with multiple GPUs. As the cluster size increases, so does the significant drop in performance. This drop in performance is primarily caused by the communications overhead between nodes in a cluster.

SageMaker distributed (SMD) offers two options for distributed training: SageMaker Distributed Model Parallel (SMP) and SageMaker Data Parallel (SDP). This guide focuses on how to train models using a data parallel strategy. For more information on training with a model-parallel strategy, refer to SageMaker Distributed Model Parallel (p. 1066).

Topics
- Introduction to SageMaker Distributed Data Parallel (p. 1055)
- Use the SageMaker Distributed Data Parallel API (p. 1057)
- Modify Your Training Script for SDP (p. 1060)
- Data Parallelism FAQ (p. 1065)

Introduction to SageMaker Distributed Data Parallel

Why Use SageMakerDistributed Data Parallel?

SageMaker Distributed Data Parallel (SDP) addresses communications overhead in two ways:

1. SDP performs AllReduce, a key operation during distributed training that is responsible for a large portion of communication overhead.
2. SDP performs optimized node-to-node communication by fully utilizing AWS's network infrastructure and Amazon EC2 instance topology.

Use SDP to increase speed by up to 25% in training models such as BERT. While implementations like Horovod offer sub-linear performance at scale, SDP offers near-linear performance at scale. This means that you get a faster training time and a lower cost to train a model.

Training Benchmarks

PyTorch with SDP
Using instance type `p3dn.24xlarge` and on 2, 4, and 8 node clusters:

- **BERT**: PyTorch-SDP is 41%, 52%, and 13% faster than PyTorch-DDP.
- **MaskRCNN**: PyTorch-SDP is 4%, 19%, and 15% faster than PyTorch-DDP.

These benchmarks were run on PyTorch v1.6 using `ml.p3dn.24xlarge` instances. You can find the training code on the SageMaker examples website. The examples website also has benchmark training code for these models using TensorFlow 2.3.

### Optimal Bandwidth Use with Balanced Fusion Buffer

SDP uses a communication pattern similar to parameter servers to reduce the amount of data transferred and the number of steps involved in averaging gradients from multiple GPUs. It also uses a new technique called balanced fusion buffers to make optimal use of the bandwidth available across all nodes in the cluster.

One key disadvantage of traditional parameter servers is their suboptimal use of available network bandwidth. Parameter servers treat variables as atomic units and place each variable on one server. Since gradients become available sequentially during the backward pass, at any given instant, there is imbalance in the volume of data being sent and received from different servers. Some servers are receiving and sending more data, some less, and some none. This problem becomes worse as the number of parameter servers increases.

SDP addresses these problems by introducing *balanced fusion buffers*. A balanced fusion buffer is a buffer in the GPU that holds the gradients until the size of the buffer exceeds a threshold. In a setup with N parameter servers, when the buffer exceeds the threshold, the balanced fusion buffer is copied to CPU memory, sharded into N parts, and the ith part is sent to the ith parameter server. Each server receives exactly the same number of bytes from a balanced fusion buffer. The ith server receives the ith partition of the balanced fusion buffer from all workers, sums them up, and sends the results back to all workers. Since all the servers participate equally in averaging each balanced fusion buffer, server bandwidth is efficiently utilized.

<table>
<thead>
<tr>
<th>Model</th>
<th>Setup</th>
<th>Throughput</th>
<th>Scaling Eff</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>PT-DDP</td>
<td>SageMaker</td>
</tr>
<tr>
<td>BERT Large</td>
<td>2 node p3dn.24x1</td>
<td>1752</td>
<td>2479</td>
</tr>
<tr>
<td>(seqs/sec)</td>
<td>4 node p3dn.24x1</td>
<td>3017</td>
<td>4603</td>
</tr>
<tr>
<td></td>
<td>8 node p3dn.24x1</td>
<td>7409</td>
<td>8551</td>
</tr>
<tr>
<td>MaskRCNN</td>
<td>2 node p3dn.24x1</td>
<td>152</td>
<td>158</td>
</tr>
<tr>
<td>(samples/sec)</td>
<td>4 node p3dn.24x1</td>
<td>258</td>
<td>307</td>
</tr>
<tr>
<td></td>
<td>8 node p3dn.24x1</td>
<td>545</td>
<td>617</td>
</tr>
</tbody>
</table>
Optimal GPU Usage with Efficient AllReduce Overlapping with a Backward Pass

SDP achieves optimal overlapping of the AllReduce operation with the backward pass, significantly improving the GPU utilization, and achieves near-linear scaling efficiency and faster time to train by optimizing tasks between CPUs and GPUs. SDP performs AllReduce in parallel while GPU is computing gradients without taking away additional GPU cycles, which makes SDP faster.

- **Leverages CPUs**: SDP uses CPUs to AllReduce gradients, offloading this task from the GPUs.
- **Improved GPU usage**: The cluster’s GPUs focus on computing gradients, improving their utilization throughout training.

Use the SageMaker Distributed Data Parallel API

Use Distributed Data Parallel with SageMaker

To use distributed data parallel (SDP) with Amazon SageMaker, you start with a training script. Just as with any training job on SageMaker, you use your training script to launch a SageMaker training job. To get started, we recommend that you use SDP on SageMaker in the following ways:

- Using a SageMaker notebook instance
- Using SageMaker Studio

With either option, you can then use the Distributed Training Jupyter Notebook Examples (p. 1083) provided with this documentation to get started.

SageMaker Python SDK's SDP APIs

To use the SageMaker Distributed Data Parallel (SDP) library, you must create a training script for one of the supported frameworks and launch the training job using the SageMaker Python SDK. To learn how you can incorporate SMP into a training script, see Modify Your Training Script to Use SMP (p. 1075). The SDP API documentation is located in the SageMaker Python SDK. See the SDP API documentation.

SageMaker supports the following training environment configurations:

- You can use a prebuilt TensorFlow or PyTorch container.
- You can customize SageMaker prebuilt containers or extend them to handle any additional functional requirements for your algorithm or model that the prebuilt SageMaker Docker image doesn't support. For an example of how you can extend a pre-built container, see Extend a Prebuilt Container.

To extend a pre-built container, or adapt your own container to use SDP, you must the following Pytorch 1.6.0 or TensorFlow 2.3.1 GPU general framework base-images:

- 763104351884.dkr.ecr.us-east-1.amazonaws.com/tensorflow-training:2.3.1-gpu-py37-cu110-ubuntu18.04
- 763104351884.dkr.ecr.us-east-1.amazonaws.com/pytorch-training:1.6.0-gpu-py36-cu110-ubuntu18.04

For example, if you are using PyTorch 1.6.0, your Dockerfile should contain a FROM statement similar to the following:

```python
# SageMaker PyTorch image
FROM 763104351884.dkr.ecr.us-east-1.amazonaws.com/pytorch-training:1.6.0-gpu-py36-cu110-ubuntu18.04
ENV PATH="/opt/ml/code:${PATH}"
```
# this environment variable is used by the SageMaker PyTorch container to determine our user code directory.
ENV SAGEMAKER_SUBMIT_DIRECTORY /opt/ml/code

# /opt/ml and all subdirectories are utilized by SageMaker, use the /code subdirectory to store your user code.
COPY cifar10.py /opt/ml/code/cifar10.py

# Defines cifar10.py as script entrypoint
ENV SAGEMAKER_PROGRAM cifar10.py

- You can adapt your own Docker container to work with SageMaker using the SageMaker Training toolkit. For an example, see Adapting Your Own Training Container.

In all cases, you launch your job using a SageMaker Python SDK Estimator.

If you are using the SageMaker pre-built PyTorch or TensorFlow containers, see the following section to learn how configure an Estimator to use SDP.

If you bring your own container or extend a pre-built container, you can create an instance of the Estimator class with the desired Docker image.

**TensorFlow Estimator**

You can use the SDP API by specifying `smdistributed dataparallel` as the distribution strategy in the SageMaker TensorFlow Estimator API. Please refer to SageMaker TensorFlow Estimator API specification for full details about the API.

```python
class sagemaker.tensorflow.estimator.TensorFlow(py_version=None, framework_version=None, model_dir=None, image_uri=None, distribution=None, **kwargs)
```

The following two parameters of the SageMaker Python SDK TensorFlow estimator are required to use `smdistributed.dataparallel` with TensorFlow.

- `distribution dict (optional)`: A dictionary with information on how to run distributed training (default: None).
  - To use `smdistributed.dataparallel` as a distribution strategy, use the following option:

    ```python
distribution = { "smdistributed": { "dataparallel": { "enabled": True } } }
```

- `train_instance_type str (required)`: A type of Amazon EC2 instance to use.
  - If the `smdistributed.dataparallel` distribution strategy is used, the allowed instance types are: `ml.p4d.24xlarge`, `ml.p3dn.24xlarge`, and `ml.p3.16xlarge`.

**Example**

```python
from sagemaker.tensorflow import TensorFlow

tf_estimator = TensorFlow(  
    base_job_name = "training_job_name_prefix",  
    entry_point="tf-train.py",  
    role="SageMakerRole",  
    framework_version="2.3",  
    # You must set `py_version` to `py36`  
    py_version='py36',  
    # For training with multi node distributed training, set this count.
```

1058
# Data Parallel Training

## Example: 2,3,4,...8

```python
instance_count=1,
instance_type="ml.p3.16xlarge",
```

## Training using smdistributed.dataparallel Distributed Training Framework

```python
distribution={"smdistributed": {
   "dataparallel": {
      "enabled": True
   }
}}
```

```python
tf_estimator.fit("s3://bucket/path/to/training/data")
```

### Note

If you are using the SageMaker Python SDK 1.x, you need to use hyperparameters to specify to use `smdistributed dataparallel` as the distributed training strategy.

```python
from sagemaker.tensorflow import TensorFlow
tf_estimator = TensorFlow(
    ...
    # Training using smdistributed.dataparallel Distributed Training Framework
    hyperparameters={"sagemaker_distributed_dataparallel_enabled": True},
    ...
)
tf_estimator.fit("s3://bucket/path/to/training/data")
```

## PyTorch Estimator

You can use the SDP API by specifying `smdistributed dataparallel` as the distribution strategy in SageMaker PyTorch Estimator API. Please refer to the SageMaker PyTorch Estimator API specification for full details of the API.

```python
class sagemaker.pytorch.estimator.PyTorch(py_version=None, framework_version=None, model_dir=None, image_uri=None, hyperparameters=None, distribution=None, **kwargs)
```

The following example shows how to use `smdistributed.dataparallel` as a distribution strategy in SageMaker with PyTorch.

### distribution (dict)(optional):
A dictionary with information on how to run distributed training (default: None).

- To use `smdistributed.dataparallel` as a distribution strategy, use the following option:

```python
distribution={"smdistributed": { "dataparallel": { "enabled": True } }}
```

### train_instance_type (str)(required):
A type of Amazon EC2 instance to use.

- If the `smdistributed.dataparallel` distribution strategy is used, the allowed instance types are: `ml.p4d.24xlarge`, `ml.p3dn.24xlarge`, and `ml.p3.16xlarge`.

### Example

```python
from sagemaker.pytorch import PyTorch
```
pt_estimator = PyTorch(
    base_job_name="training_job_name_prefix",
    entry_point="pt-train.py",
    role="SageMakerRole",
    # You must set py_version to py36
    py_version='py36',
    framework_version='1.6.0',
    # For training with multi node distributed training, set this count.
    # Example: 2,3,4,...8
    instance_count=1,
    # For training with p3dn instance use - ml.p3dn.24xlarge
    instance_type="ml.p3.16xlarge",
    # Training using smdistributed.dataparallel Distributed Training
    Framework
distribution="smdistributed": {
        "dataparallel": {
            "enabled": True
        }
    }
)
pt_estimator.fit("s3://bucket/path/to/training/data")

Note
If you are using SageMaker Python SDK 1.x, you need to use hyperparameters to specify
smdistributed.dataparallel as the distributed training strategy.

from sagemaker.pytorch import PyTorch
pt_estimator = PyTorch(...
    # Training using smdistributed.dataparallel Distributed Training
    Framework
    hyperparameters="sagemaker_distributed_dataparallel_enabled": True),
    ...
)
pt_estimator.fit("s3://bucket/path/to/training/data")

Modify Your Training Script for SDP

Script Modification Overview
SDP APIs are designed for ease of use, and to provide seamless integration with existing distributed
training toolkits.

- **SageMaker Python SDK with the SageMaker SDP API:** In most cases, all you have to change in your
training script is the Horovod or SageMaker Distributed Data Parallel (SDP) import statements. Swap
these out with the SDP equivalents.
- **Focus on your model training without infrastructure management:** When training a deep learning
model with SDP on SageMaker, you can focus on your model training, while SageMaker does cluster
management: brings up the nodes and creates the cluster, completes the training, then tears down the
cluster.

To customize your own training script, you need to do the following:

- You must provide TensorFlow/PyTorch training scripts that are adapted to use SDP. The following
sections provide example code for this.
- Your input data must be in an S3 bucket or in FSx in the AWS Region that you will use to launch your
training job. If you use the Jupyter Notebooks provided, create a SageMaker notebook instance in the
same Region as the bucket that contains your input data. For more information about storing your training data, refer to the SageMaker Python SDK data inputs documentation.

Tip
Consider using FSx instead of Amazon S3 to increase training performance. It has higher throughput and lower latency than Amazon S3.

Use the following sections to see examples of training scripts that can be used to convert your TensorFlow or PyTorch training scripts. Then, use one of the example notebooks for your template to launch a training job. You'll need to swap your training script with the one that came with the notebook and modify any input functions as necessary. Once you have launched a training job, you can monitor it using Amazon CloudWatch.

Then you can see how to deploy your trained model to an endpoint by following one of the example notebooks for deploying a model.

Finally, you can follow an example notebook to test inference on your deployed model.

Modify a TensorFlow 2.x Training Script to use SMD Data Parallel

The following steps show you how to convert a TensorFlow 2.x training script to utilize SDP.

The SDP APIs are designed to be similar to Horovod APIs. Refer to the SDP TensorFlow API specification for additional details on each API that SDP offers for TensorFlow.

1. Import SDP's TensorFlow client and initialize it:

   ```python
   import smdistributed.dataparallel.tensorflow as sdp
   sdp.init()
   ```

2. Pin each GPU to a single `smdistributed.dataparallel` process with `local_rank`—this refers to the relative rank of the process within a given node. The `sdp.tensorflow.local_rank()` API provides you the local rank of the device. The leader node is rank 0, and the worker nodes are rank 1, 2, 3, and so on. This is invoked in the next code block as `sdp.local_rank()`.

   ```python
   gpus = tf.config.experimental.list_physical_devices('GPU')
   for gpu in gpus:
       tf.config.experimental.set_memory_growth(gpu, True)
   if gpus:
       tf.config.experimental.set_visible_devices(gpus[sdp.local_rank()], 'GPU')
   ```

3. Scale the learning rate by the number of workers. The `sdp.tensorflow.size()` API provides you the number of workers in the cluster. This is invoked in the next code block as `sdp.size()`.

   ```python
   learning_rate = learning_rate * sdp.size()
   ```

4. Use SDP's `DistributedGradientTape` to optimize `AllReduce` operations during training. This wraps `tf.GradientTape`.

   ```python
   with tf.GradientTape() as tape:
       output = model(input)
       loss_value = loss(label, output)
   ```
5. Broadcast the initial model variables from the leader node (rank 0) to all the worker nodes (ranks 1 through n). This is needed to ensure a consistent initialization across all the worker ranks. For this, you use the `sdp.tensorflow.broadcast_variables` API after the model and optimizer variables are initialized. This is invoked in the next code block as `sdp.broadcast_variables()`. 

```python
sdp.broadcast_variables(model.variables, root_rank=0)
sdp.broadcast_variables(opt.variables(), root_rank=0)
```

6. Finally, modify your script to save checkpoints only on the leader node. The leader node has a synchronized model. This also avoids worker nodes overwriting the checkpoints and possibly corrupting the checkpoints.

```python
if sdp.rank() == 0:
    checkpoint.save(checkpoint_dir)
```

The following is an example TensorFlow 2 training script for distributed training with SDP:

```python
import tensorflow as tf
# SDP: Import SDP TF API
import smdistributed.dataparallel.tensorflow as sdp
# SDP: Initialize SDP
sdp.init()
gpus = tf.config.experimental.list_physical_devices('GPU')
for gpu in gpus:
    tf.config.experimental.set_memory_growth(gpu, True)
if gpus:
    # SDP: Pin GPUs to a single SDP process
    tf.config.experimental.set_visible_devices(gpus[sdp.local_rank()], 'GPU')

# Prepare Dataset
dataset = tf.data.Dataset.from_tensor_slices(...)

# Define Model
mnist_model = tf.keras.Sequential(...)
loss = tf.losses.SparseCategoricalCrossentropy()

# SDP: Scale Learning Rate
# LR for 8 node run : 0.000125
# LR for single node run : 0.001
opt = tf.optimizers.Adam(0.000125 * sdp.size())

@tf.function
def training_step(images, labels, first_batch):
    with tf.GradientTape() as tape:
        probs = mnist_model(images, training=True)
        loss_value = loss(labels, probs)

    # SDP: Wrap tf.GradientTape with SDP's DistributedGradientTape
tape = sdp.DistributedGradientTape(tape)

    grads = tape.gradient(loss_value, mnist_model.trainable_variables)

    opt.apply_gradients(zip(grads, mnist_model.trainable_variables))
```
if first_batch:
    # SDP: Broadcast model and optimizer variables
    sdp.broadcast_variables(mnist_model.variables, root_rank=0)
    sdp.broadcast_variables(opt.variables(), root_rank=0)

    return loss_value
...

# SDP: Save checkpoints only from master node.
if sdp.rank() == 0:
    checkpoint.save(checkpoint_dir)

For more advanced usage, refer to SageMaker Distributed Data Parallel TensorFlow API documentation.

Modify a PyTorch Training Script to Use SMD Data Parallel

The following steps show you how to convert a PyTorch training script to utilize SageMaker distributed data parallel (SDP).

The SDP APIs are designed to be similar to PyTorch Distributed Data Parallel (DDP) APIs. For additional details on what each API SDP offers for PyTorch, see the SageMaker distributed data parallel PyTorch API documentation.

1. Import SDP's PyTorch client and initialize it, then import the SDP module for distributed training.

   ```python
   import smdistributed.dataparallel.torch.distributed as dist
   from smdistributed.dataparallel.torch.parallel.distributed import DistributedDataParallel as DDP
   dist.init_process_group()
   ```

2. Pin each GPU to a single SDP process with `local_rank`—this refers to the relative rank of the process within a given node.

   The `smdistributed.dataparallel.torch.get_local_rank()` API provides you the local rank of the device. The leader node is rank 0, and the worker nodes are rank 1, 2, 3, and so on. This is invoked in the next code block as `dist.get_local_rank()`.

   ```python
   torch.cuda.set_device(dist.get_local_rank())
   ```

3. Wrap the PyTorch model with SDP's DDP.

   ```python
   model = ...
   # Wrap model with SDP DistributedDataParallel
   model = DDP(model)
   ```

4. Modify the `torch.utils.data.distributed.DistributedSampler` to include the cluster's information. Set `num_replicas` to the total number of GPUs participating in training across all the nodes in the cluster. This is called `world_size`. You can get `world_size` with the `smdistributed.dataparallel.torch.get_world_size()` API. This is invoked in the following code as `dist.get_world_size()`. Also supply the node rank using `smdistributed.dataparallel.torch.get_rank()`. This is invoked as `dist.get_rank()`.
5. Modify your script to save checkpoints only on the leader node. The leader node has a synchronized model. This also avoids worker nodes overwriting the checkpoints and possibly corrupting the checkpoints.

The following is an example PyTorch training script for distributed training with SDP:

```python
# SDP: Import SDP PyTorch API
import smdistributed.dataparallel.torch.distributed as dist

# SDP: Import SDP PyTorch DDP
from smdistributed.dataparallel.torch.parallel.distributed import DistributedDataParallel as DDP

# SDP: Initialize SDP
dist.init_process_group()

class Net(nn.Module):
    ...
    # Define model

def train(...):
    ...
    # Model training

def test(...):
    ...
    # Model evaluation

def main():
    # SDP: Scale batch size by world size
    batch_size //= dist.get_world_size() // 8
    batch_size = max(batch_size, 1)

    # Prepare dataset
    train_dataset = torchvision.datasets.MNIST(...)

    # SDP: Set num_replicas and rank in DistributedSampler
    train_sampler = torch.utils.data.distributed.DistributedSampler(
        train_dataset,
        num_replicas=dist.get_world_size(),
        rank=dist.get_rank())

    train_loader = torch.utils.data.DataLoader(..)

    # SDP: Wrap the PyTorch model with SDP’s DDP
    model = DDP(Net().to(device))

    # SDP: Pin each GPU to a single SDP process.
    torch.cuda.set_device(local_rank)
    model.cuda(local_rank)

    # Train
    optimizer = optim.Adadelta(...)
    scheduler = StepLR(...)
    for epoch in range(1, args.epochs + 1):
        train(...)
        if rank == 0:
            test(...)  
            scheduler.step()```
Data Parallel Training

# SDP: Save model on master node.
if dist.get_rank() == 0:
    torch.save(...)

if __name__ == '__main__':
    main()

For more advanced usage, see the SageMaker Distributed Data Parallel PyTorch API documentation.

Launch a Training Job

To launch the training job using your SDP configured training script, you use the `Estimator.fit()` function. We recommend that you launch a training job using a SageMaker notebook instance or SageMaker Studio. To see an example of how you can launch a training job using a Studio or a notebook instance, see Distributed Training Jupyter Notebook Examples (p. 1083).

Use the following resources to learn more about using the SageMaker Python SDK with these frameworks:

- Use TensorFlow with the SageMaker Python SDK
- Using PyTorch with the SageMaker Python SDK

Data Parallelism FAQ

Q: When using SageMaker Distributed Data Parallelism (SDP), how are the allreduce-supporting CPU instances managed? Do I have to create heterogenous CPU-GPU clusters, or does the SageMaker service create extra CSs for SDP-enabled jobs?

SDP uses the CPUs available with GPU instances. No additional C5 or CPU instances are launched; if your SageMaker training job is 8 node `ml.p3dn.24xlarge` clusters, only 8 `ml.p3dn.24xlarge` instances are used. No additional instances are provisioned.

Q: I have a training job taking 5 days on a single `ml.p3.24xlarge` instance with a set of hyperparameters H1 (learning rate, batch size, optimizer, etc). Is enabling SDP and using a 5x bigger cluster enough to experience an approximate 5x speedup? Or will I have to revisit its training hyperparameters after enabling SDP?

Enabling SDP would change the overall batch size. The new overall batch size is scaled linearly with the number of training instances used. As a result of this, hyperparameters, such as learning rate, have to be changed to ensure convergence.

Q: Does SDP support Spot?

Yes. You can use managed spot training. You specify the path to the checkpoint file in the SageMaker training job. You enable save/restore checkpoints in their training script as mentioned in the last step of the section called “Script Modification Overview” (p. 1060).

Q: What is the difference between SDP balanced fusion buffers and PyTorch Distributed Data Parallel (DDP) “Gradient Buckets”?

The primary difference is that DDP’s fusion buffer is used for `AllReduce` and SDP’s balanced fusion buffer is used for parameter server style synchronization. SDP is the first framework to shard fusion buffers in parameter server-based gradient synchronization.

From the PyTorch DDP documentation: “To improve communication efficiency, the `Reducer` organizes parameter gradients into buckets, and reduces one bucket at a time. Bucket size can be configured by..."
setting the bucket_cap_mb argument in DDP constructor. The mapping from parameter gradients to buckets is determined at the construction time, based on the bucket size limit and parameter sizes. Model parameters are allocated into buckets in (roughly) the reverse order of Model.parameters() from the given model. The reason for using the reverse order is because DDP expects gradients to become ready during the backward pass in approximately that order.”

Q: Is SDP relevant in single-host, multi-device setup?

SDP can be used with a single-host, multi-device setup. However, in two or more nodes, SDP's AllReduce operation gives you significant performance improvement. Also, on a single host, NVLink already contributes to in-node AllReduce efficiency.

Q: Can SDP be used with PyTorch Lightning?

No. However, with SDP's DDP for PyTorch, you can write custom DDP to achieve the functionality.

Q: Where should the training dataset be stored?

The training dataset can be stored in an S3 bucket or on an FSx drive. See this document for various supported input filesystem for a training job.

Q: With SageMaker SDP, is it mandatory to have training data in FSx for Lustre? Can EFS and S3 be used?

We recommend using FSx to cut down the time to kickstart the training. It is not mandatory.

Q: Can SDP be used with CPU nodes?

No. SDP supports ml.p3.16xlarge, ml.p3dn.24xlarge, and ml.p4d.24xlarge instances at this time.

Q: What frameworks and framework versions are currently supported by SageMaker SDP at launch?

SDP currently supports PyTorch v1.6 and Tensorflow v2.3.1. Support for Tensorflow 1.x.

**SageMaker Distributed Model Parallel**

**Important**

To use new features with an existing notebook instance or Studio app, restart it to get the latest updates.

Amazon SageMaker distributed model parallel (SMP) is a model parallelism library for training large deep learning models that were previously difficult to train due to GPU memory limitations. SMP automatically and efficiently splits a model across multiple GPUs and instances and coordinates model training, allowing you to increase prediction accuracy by creating larger models with more parameters.

You can use SMP to automatically partition your existing TensorFlow and PyTorch workloads across multiple GPUs with minimal code changes. You can access the SMP API through the SageMaker SDK.

Use the following sections to learn more about model parallelism and the SMP library.

**Topics**

- Introduction to Model Parallelism (p. 1067)
- Core Features of SageMaker Distributed Model Parallel (p. 1067)
- Use the SageMaker Distributed Model Parallel API (p. 1072)
- Modify Your Training Script to Use SMP (p. 1075)
### Introduction to Model Parallelism

**Model parallelism** is the process of splitting a model up between multiple devices or nodes (such as GPU-equipped instances) and creating an efficient pipeline to train the model across these devices to maximize GPU utilization.

Use the sections on this page to learn more about model parallelism, including how it can be used to overcome issues that arise when training large ML models, and important considerations when integrating it into your ML training script.

#### What is Model Parallelism?

Increasing deep learning model size (layers and parameters) can result in better accuracy, however there is a limit to the maximum model size you can fit in a single GPU. When training deep learning models, GPU memory limitations can be a bottleneck in the following ways:

- They can limit the size of the model you train. Given that larger models tend to achieve higher accuracy, this directly translates to trained model accuracy.
- They can limit the batch size you train with, leading to lower GPU utilization and slower training.

To overcome the limitations associated with training a model on a single GPU, you can use model parallelism to distribute and train your model on multiple computing devices.

#### Important Considerations when Using Model Parallelism

When you use model parallelism, you must consider the following:

- **How you split your model across devices**: The computational graph of your model, sizes of model parameters and activations, and your resource constraints (for example, time vs. memory) determine the best partitioning strategy.

  To reduce the time and effort required to efficiently split your model, you can use automated model splitting features offered by SMP.

- **Achieving parallelization**: Model training—that is, forward computations and backward propagation—is inherently sequential, where each operation must wait for its inputs to be computed by another operation. For this reason, forward and backward pass stages of deep learning training are not easily parallelizable, and naively splitting a model across multiple GPUs may lead to poor device utilization. For example, a layer on GPU $i+1$ has to wait for the output from a layer on GPU $i$, and so GPU $i+1$ remains idle during this waiting period.

  SMP can achieve true parallelization by implementing pipelined execution by building an efficient computation schedule where different devices can work on forward and backward passes for different data samples at the same time.

To learn how you can use SMP to efficiently split your model across devices and improve device utilization during training, see [Core Features of SageMaker Distributed Model Parallel](p. 1067).

#### Core Features of SageMaker Distributed Model Parallel

Amazon SageMaker distributed model parallel (SMP) makes model parallelism more accessible by providing automated model splitting and sophisticated pipeline execution scheduling. The model splitting algorithms can optimize for speed or memory consumption. SMP also supports manual partitioning. When you use SMP, training is executed in a pipelined fashion over microbatches to maximize GPU usage.
You can configure these features using a few lines of code when you create your training script and define your SageMaker PyTorch or TensorFlow Estimator. Use the following sections to learn more about these core features of the SMP library.

**Automated Model Splitting**

When you use SMP with SageMaker, you can take advantage of **automated model splitting**, also referred to as **automated model partitioning**. SMP uses a partitioning algorithm that balances memory, minimizes communication between devices, and optimizes performance. You can configure the automated partitioning algorithm to optimize for speed or memory.

Alternatively, you can use manual model splitting. We recommend automated model splitting, unless you are very familiar with the model architecture and have a good idea of how to efficiently partition your model.

**How It Works**

Auto-partitioning occurs during the first training step, when the `smp.step`-decorated function is first called. During this call, SMP first constructs a version of the model on the CPU RAM (to avoid GPU memory limitations), and then analyzes the model graph and makes a partitioning decision. Based on this decision, each model partition is loaded on a GPU, and only then the first step is executed. Because of these analysis and partitioning steps, the first training step might take longer.

In either framework, SMP manages the communication between devices through its own backend, which is optimized for AWS infrastructure.

The auto-partition design adapts to the characteristics of the framework, and SMP does the partitioning at the granularity level that is more natural in each framework. For instance, in TensorFlow, each specific operation can be assigned to a different device, whereas in PyTorch, the assignment is done at the module level, where each module consists of multiple operations. The follow section reviews the specifics of the design in each framework.

**Automated Model Splitting with TensorFlow**

SMP analyzes the sizes of the trainable variables and the graph structure, and internally uses a graph partitioning algorithm. This algorithm comes up with a device assignment for each operation, with the objective of minimizing the amount of communication needed across devices, subject to two constraints:

- Balancing the number of variables stored in each device
- Balancing the number of operations executed in each device

If you specify `speed` for `optimize` (in the model parallel parameters in the Python SDK), SMP tries to balance the number of operations and `tf.Variable` objects in each device. Otherwise, it tries to balance the total size of `tf.Variables`.

Once the partitioning decision is made, SMP creates a serialized representation of the subgraph that each device needs to execute and imports them onto each device. While partitioning, SMP places operations that consume the same `tf.Variable` and operations that are part of the same Keras layer onto the same device. It also respects the colocation constraints imposed by TensorFlow. This means that, for example, if there are two Keras layers that share a `tf.Variable`, then all operations that are part of these layers are placed on a single device.

**Automated Model Splitting with PyTorch**

During the first training step, SMP internally runs a tracing step that is meant to construct the model graph and determine the tensor and parameter shapes. After this tracing step, SMP constructs a tree, which consists of the nested `nn.Module` objects in the model, as well as additional data gathered from tracing, such as the amount of stored `nn.Parameters`, and execution time for each `nn.Module`. 
Next, SMP traverses this tree from the root and runs a partitioning algorithm that assigns each
nn.Module to a device, which balances computational load (measured by module execution time) and
memory use (measured by the total stored nn.Parameter size and activations). If multiple nn.Modules
share the same nn.Parameter, then these modules are placed on the same device to avoid maintaining
multiple versions of the same parameter. Once the partitioning decision is made, the assigned modules
and weights are loaded to their devices.

**Comparison of Automated Model Splitting Between Frameworks**

In TensorFlow, the fundamental unit of computation is a tf.Operation, and TensorFlow represents
the model as a directed acyclic graph (DAG) of tf.Operations, and therefore SMP partitions this
DAG so that each node goes to one device. Crucially, tf.Operation objects are sufficiently rich with
customizable attributes, and they are universal in the sense that every model is guaranteed to consist of
a graph of such objects.

PyTorch on the other hand, does not have an equivalent notion of operation that is sufficiently rich and
universal. The closest unit of computation in PyTorch that has these characteristics is an nn.Module,
which is at a much higher granularity level, and this is why SMP does partitioning at this level in PyTorch.

**Manual Model Splitting**

If you want to manually specify how your model is partitioned across devices, you can use manual model
splitting by using smp.partition context managers.

To use this option, set auto_partition to False, and define a default_partition in the
SageMaker Python SDK. Any operation that is not explicitly placed on a partition through the
smp.partition context manager is executed on the default_partition. In this case, the automated
splitting logic is bypassed, and each operation is placed based on your specification. Based on the
resulting graph structure, SMP creates a pipelined execution schedule automatically.

**Pipeline Execution Schedule**

A core feature of SMP is pipelined execution, which determines the order in which computations are
made and data is processed across devices during model training. Pipelining is a technique to achieve
true parallelization in model parallelism, by having the GPUs compute simultaneously on different data
samples, and to overcome the performance loss due to sequential computation.

Pipelining is based on splitting a mini-batch into microbatches, which are fed into the training pipeline
one-by-one and follow an execution schedule defined by the SMP runtime. A microbatch is a smaller
subset of a given training mini-batch. The pipeline schedule determines which microbatch is executed by
which device for every time slot.

For example, depending on the pipeline schedule and the model partition, GPU i might perform
(forward or backward) computation on microbatch b while GPU i+1 performs computation on
microbatch b+1, thereby keeping both GPUs active at the same time. During a single forward or
backward pass, execution flow for a single microbatch might visit the same device multiple times,
depending on the partitioning decision. For instance, an operation that is at the beginning of the model
might be placed on the same device as an operation at the end of the model, while the operations in
between are on different devices, which means this device is visited twice.

SMP offers two different pipeline schedules, simple and interleaved, which can be configured using the
pipeline parameter in the SageMaker Python SDK. In most cases, interleaved pipeline can achieve
better performance by utilizing the GPUs more efficiently.

**Interleaved Pipeline**

In an interleaved pipeline, backward execution of the microbatches is prioritized whenever possible. This
allows quicker release of the memory used for activations, using memory more efficiently. It also allows
for scaling the number of microbatches higher, reducing the idle time of the GPUs. At steady-state, each device alternates between executing forward and backward passes. This means that the backward pass of one microbatch may execute before the forward pass of another microbatch finishes.

The preceding figure illustrates an example execution schedule for the interleaved pipeline over 2 GPUs. In the figure, F0 represents the forward pass for microbatch 0, and B1 represents the backward pass for microbatch 1. **Update** represents the optimizer update of the parameters. GPU0 always prioritizes backward passes whenever possible (for instance, executes B0 before F2), which allows for clearing of the memory used for activations earlier.

**Simple Pipeline**

A simple pipeline, by contrast, finishes executing the forward pass for each microbatch before starting the backward pass. This means that it only pipelines the forward pass and backward pass stages within themselves. The following figure illustrates an example of how this works, over 2 GPUs.

**Pipelining Execution in Specific Frameworks**

Use the following sections to learn about the framework-specific pipeline scheduling decisions SMP makes for Tensorflow and PyTorch.

**Pipeline Execution with TensorFlow**

The following image is an example of a TensorFlow graph partitioned by SMP, using automated model splitting. When a graph is split, each resulting subgraph is replicated B times (except for the variables), where B is the number of microbatches. In this figure, each subgraph is replicated 2 times (B=2). An **SMPInput** operation is inserted at each input of a subgraph, and an **SMPOutput** operation is inserted at each output. These operations communicate with the SMP backend to transfer tensors to and from each other.

The following image is an example of 2 subgraphs split with B=2 with gradient operations added. The gradient of a **SMPInput** op is a **SMPOutput** op, and vice versa. This enables the gradients to flow backwards during back-propagation.
This GIF demonstrates an example interleaved pipeline execution schedule with $B=2$ microbatches and 2 subgraphs. Each device sequentially executes one of the subgraph replicas to improve GPU utilization. As $B$ grows larger, the fraction of idle time slots goes to zero. Whenever it is time to do (forward or backward) computation on a specific subgraph replica, the pipeline layer signals to the corresponding blue SMP\texttt{Input} operations to start executing.

Once the gradients from all microbatches in a single mini-batch are computed, SMP combines the gradients across microbatches, which can then be applied to the parameters.

**Pipeline Execution with PyTorch**

Conceptually, pipelining follows a similar idea in PyTorch. However, since PyTorch does not involve static graphs, SMP-PyTorch features a more dynamic pipelining paradigm.

As in TensorFlow, each batch is split into a number of microbatches, which are executed one at a time on each device. However, the execution schedule is handled via execution servers launched on each device. Whenever the output of a submodule that is placed on another device is needed on the current device, an execution request is sent to the execution server of the remote device along with the input tensors to the submodule. The server then executes this module with the given inputs and returns the response to the current device.

Since the current device is idle during the remote submodule execution, the local execution for the current microbatch pauses, and the SMP runtime switches execution to another microbatch which the
current device can actively work on. The prioritization of microbatches is determined by the chosen pipeline schedule. For an interleaved pipeline schedule, microbatches that are in the backward stage of the computation are prioritized whenever possible.

**Use the SageMaker Distributed Model Parallel API**

To use SMP with Amazon SageMaker, you must create a training script for one of the supported frameworks and launch the training job using the SageMaker Python SDK. To learn how you can incorporate SMP into a training script, see **Modify Your Training Script to Use SMP**. The SMP API documentation is located in the SageMaker Python SDK. Refer to the **SMP API documentation**.

SageMaker supports the following training environment configurations.

1. You can use a prebuilt TensorFlow or PyTorch container. This option is recommended for new users of SMP, and is demonstrated in the example **MNIST with PyTorch 1.6 and SMP**.
2. You can extend SageMaker prebuilt containers to handle any additional functional requirements for your algorithm or model that the prebuilt SageMaker Docker image doesn't support. To see an example of how you can extend a pre-built container, see **Extend a Prebuilt Container**.
3. You can adapt your own Docker container to work with SageMaker using the **SageMaker Training toolkit**. For an example, see **Adapting Your Own Training Container**.

For options 2 and 3 in the preceding list, refer to **Extend or Adapt A Docker Container that Contains SMP** to learn how to install SMP in an extended or customized Docker container.

In all cases, you launch your job using a SageMaker Python SDK TensorFlow or PyTorch Estimator to initialize SMP and launch a training job. See the following section, **Launch a Training Job with the SageMaker Python SDK** (p. 1072), to learn more.

**Launch a Training Job with the SageMaker Python SDK**

The SageMaker Python SDK supports managed training of TensorFlow and PyTorch models. To launch a training job using one of these frameworks, you can define a TensorFlow Estimator or a PyTorch Estimator.

The TensorFlow and PyTorch Estimator object contains a distribution parameter, which is used to enable and specify parameters for the initialization of the SMP library. SMP internally uses MPI, so in order to use model parallelism, MPI must be enabled using the distribution parameter.

The following is an example of how you can launch a new PyTorch training job with SMP.

```python
sagemaker_session = sagemaker.session.Session(boto_session=session)

mpi_options = {
    "enabled": True,
    "processes_per_host": 8,
    "custom_mpi_options": "--mca btl_vader_single_copy_mechanism none"
}

smp_options = {
    "enabled": True,
    "parameters": {
        "microbatches": 4,
        "placement_strategy": "spread",
        "pipeline": "interleaved",
        "optimize": "speed",
        "partitions": 2,
        "ddp": True,
    }
}
```
smd_mp_estimator = PyTorch(
    entry_point="pt_mnist.py", # Pick your train script
    source_dir='utils',
    role=role,
    instance_type='ml.p3.16xlarge',
    sagemaker_session=sagemaker_session,
    framework_version='1.6.0',
    # You must set py_version to py36
    py_version='py36',
    instance_count=1,
    distribution={
        "smdistributed": smp_options,
        "mpi": mpi_options
    },
    base_job_name="SMD-MP-demo",
)
smd_mp_estimator.fit('s3://my_bucket/my_training_data/')

To enable SMP, a dictionary with the keys "mpi" and "smdistributed" needs to be passed as the distribution argument of the TensorFlow and PyTorch Estimator constructors in Python SDK. For the "mpi" key, a dict must be passed which contains:

- "enabled": True to launch the training job with MPI.
- "processes_per_host": Set this to a number less than or equal to the number of GPUs available in your chosen instance type. SMP maintains a one-to-one mapping between processes and GPUs.
- "custom_mpi_options": Use this key to pass any custom MPI options you might need. To avoid Docker warnings from contaminating your training logs, we recommend the following flag.

```
--mca btl_vader_single_copy_mechanism none
```

For the "smdistributed" key, a dictionary must be passed which has the only key "modelparallel". Using "modelparallel" and "dataparallel" in the same training job is not supported.

For the "modelparallel" dictionary, an inner dictionary must be passed, which enables the modelparallel library by setting "enabled": True, as well as a dict of "parameters" to customize SMP. Among the parameters, the "partitions" key is required, which specifies how many model partitions are requested. To learn more about values you can use in parameters, see the SMP API documentation.

To launch the training job using your SMP configured training script, you use the `Estimator.fit()` function. You can launch a SageMaker training job using a SageMaker notebook instance, or locally.

- Using a SageMaker notebook instance is recommended for new users. To see an example of how you can launch a training job using a SageMaker notebook instance, see Distributed Training Jupyter Notebook Examples (p. 1083).
- You can launch locally if you have set up your AWS credentials and Region for development.

Use the following resources to learn more about using the SageMaker Python SDK with these frameworks:

- Use TensorFlow with the SageMaker Python SDK
- Use PyTorch with the SageMaker Python SDK
Extend or Adapt A Docker Container that Contains SMP

To extend a pre-built container, or adapt your own container to use SMP, you must the following Pytorch 1.6.0 or TensorFlow 2.3.1 GPU general framework base-images:

- 763104351884.dkr.ecr.us-east-1.amazonaws.com/tensorflow-training:2.3.1-gpu-py37-cu110-ubuntu18.04
- 763104351884.dkr.ecr.us-east-1.amazonaws.com/pytorch-training:1.6.0-gpu-py36-cu110-ubuntu18.04

For example, if you were using Pytorch 1.6.0, your Dockerfile should contain a FROM statement similar to the following:

```plaintext
# SageMaker PyTorch image
FROM 763104351884.dkr.ecr.us-east-1.amazonaws.com/pytorch-training:1.6.0-gpu-py36-cu110-ubuntu18.04

# Add your dependencies here
RUN ...

ENV PATH="/opt/ml/code:${PATH}"

# this environment variable is used by the SageMaker PyTorch container to determine our user code directory.
ENV SAGEMAKER_SUBMIT_DIRECTORY /opt/ml/code

# /opt/ml and all subdirectories are utilized by SageMaker, use the /code subdirectory to store your user code.
COPY cifar10.py /opt/ml/code/cifar10.py
```

Additionally, when you define a PyTorch or TensorFlow estimator, you must specify that the entry_point for your training script. This should be the same path identified with ENV SAGEMAKER_SUBMIT_DIRECTORY in your Dockerfile.

You must push this docker container to Amazon Elastic Container Registry (Amazon ECR) and use the image URI (image_uri) to define an estimator. See Extend a Prebuilt Container for more information.

For example, if your Dockerfile was defined using the preceding code block and used `name` and `tag` to push it to Amazon ECR, you would define a PyTorch estimator as follows. This example assumes you have already defined `smp_options` and `mpi_options`.

```python
smd_mp_estimator = PyTorch(
    entry_point='"/opt/ml/code/pt_mnist.py"',  # Identify
    source_dir='utils',
    role=role,
    instance_type='ml.p3.16xlarge',
    sagemaker_session=sagemaker_session,
    image_uri='aws_account_id.dkr.ecr.region.amazonaws.com/name:tag'
    instance_count=1,
    distribution={
        "smdistributed": smp_options,
        "mpi": mpi_options
    },
    base_job_name="SMD-MP-demo",
)

smd_mp_estimator.fit('s3://my_bucket/my_training_data/)
```
Modify Your Training Script to Use SMP

Use this page to learn how to customize your training script with SMP-specific API functions and parameters. To learn more about the SMP API, refer to SMP API documentation.

Modify a TensorFlow Training Script

The following are examples of training scripts that you can use to configure SMP with TensorFlow version 2.3, with auto-partitioning and manual partitioning. This selection of examples also includes an example integrated with Horovod for hybrid model and data parallelism. We recommend that you review Unsupported Framework Features (p. 1075) before creating a training script.

Note that auto-partitioning is enabled by default. Unless otherwise specified, the following scripts use auto-partitioning.

Topics

• Unsupported Framework Features (p. 1075)
• TensorFlow 2.3 (p. 1075)
• TensorFlow 2.3 with Horovod (p. 1076)
• Manual partitioning with TF 2.3 (p. 1077)

Unsupported Framework Features

The following TensorFlow features are unsupported by SMP:

• `tf.GradientTape()` is currently not supported. You can use `Optimizer.get_gradients()` or `Optimizer.compute_gradients()` instead to compute gradients.

• The `tf.train.Checkpoint.restore()` API is currently not supported. For checkpointing, use `smp.CheckpointManager` instead, which provides the same API and functionality. Note that checkpoint restores with `smp.CheckpointManager` should take place after the first step.

TensorFlow 2.3

```python
# SMP: Import TF2 API
import smdistributed.modelparallel.tensorflow as smp
smp.init()

class Model(smp.DistributedModel):
    def __init__(self):
        # define layers
        def call(self, x):
            # define forward pass

    loss_obj = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)

# SMP: If the last batch will not be divisible by num. microbatches,
# set drop_remainder=True.
train_ds = tf.data.Dataset.from_tensor_slices(...).batch(128, drop_remainder=True)

# SMP: Define the smp.step consisting of forward and backward passes
@smp.step
def forward_backward(images, labels):
    predictions = model(images, training=True)
    loss = loss_obj(labels, predictions)
```

1075
grads = optimizer.get_gradients(loss, model.trainable_variables)
return grads, loss

@tf.function
def train_step(images, labels):
    gradients, loss = forward_backward(images, labels)
    # SMP: accumulate the gradients across microbatches
    gradients = [g.accumulate() for g in gradients]
    optimizer.apply_gradients(zip(gradients, model.trainable_variables))
    # SMP: Average the loss across microbatches
    return loss.reduce_mean()

for images, labels in train_ds:
    loss = train_step(images, labels)
    # SMP: Only print loss at rank 0.
    if smp.rank() == 0:
        print(f"Loss: {loss}"

TensorFlow 2.3 with Horovod

SMP can be used with Horovod for hybrid model and data parallelism. In this case, the total number of GPUs must be divisible by the number of partitions. The quotient is inferred to be the number of model replicas or data parallelism degree. For instance, if SMP is launched with 8 processes (which corresponds to 8 GPUs), and partitions is 2, then SMP applies 2-way model parallelism and 4-way data parallelism over the 8 GPUs. To access the data parallel rank and model-parallel rank of a process, you can use smp.dp_rank() and smp.mp_rank(), respectively.

If you are using Horovod, you should not directly call hvd.init. Instead, set "horovod" to True in the SageMaker Python SDK modelparallel parameters, and SMP internally initializes Horovod. This is because Horovod must be initialized based on the device assignments of model partitions, and calling hvd.init() directly results in problems.

Furthermore, using hvd.DistributedOptimizer directly results in poor performance or hangs, since this implicitly places the AllReduce operation inside smp.step. The recommended way to use SMP with Horovod is by directly calling hvd.allreduce after calling accumulate() or reduce_mean() on the gradients returned from smp.step, as seen in the following example.

The four main changes needed in the script are:

* Adding hvd.allreduce
* Broadcasting variables after the first batch, as required by Horovod
* Setting the "horovod" parameter to True in the modelparallel dict in the Python SDK
* Seeding shuffling and/or sharding operations in the data pipeline with smp.dp_rank().

```python
# SMP: Import TF2 API
import smdistributed.modelparallel.tensorflow as smp
import horovod.tensorflow as hvd

# SMP: Initialize.
smp.init()

# SMP: Define the DistributedModel
class Model(smp.DistributedModel):
    def __init__(self):
        # define layers
```
def call(self, x):
    # define forward pass
    loss_obj = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
    train_ds = tf.data.Dataset.from_tensor_slices(...)

    # SMP: Seed the randomness in data pipeline with smp.dp_rank()
    train_ds = train_ds.shuffle(10000, seed=smp.dp_rank())

    # SMP: If the last batch will not be divisible by num. microbatches,
    # set drop_remainder=True.
    train_ds = train_ds.batch(128, drop_remainder=True)

    # SMP: Define the smp.step consisting of forward and backward passes
    @smp.step
    def forward_backward(images, labels):
        predictions = model(images, training=True)
        loss = loss_obj(labels, predictions)

        grads = optimizer.get_gradients(loss, model.trainable_variables)

        return grads, loss

    @tf.function
    def train_step(images, labels, first_batch):
        gradients, loss = forward_backward(images, labels)

        # SMP: Accumulate the gradients across microbatches.
        # Horovod: Allreduce gradients across model replicas.
        gradients = [hvd.allreduce(g.accumulate()) for g in gradients]
        optimizer.apply_gradients(zip(gradients, model.trainable_variables))

        # SMP: Broadcast variables.
        if first_batch:
            hvd.broadcast_variables(model.variables, root_rank=0)
            hvd.broadcast_variables(optimizer.variables(), root_rank=0)

        # SMP: Average the loss across microbatches.
        return loss.reduce_mean()

    for step, (images, labels) in enumerate(train_ds):
        loss = train_step(images, labels, step == 0)

        # SMP: Only print loss at rank 0.
        if smp.rank() == 0:
            print(f"Loss: {loss}"

---

**Manual partitioning with TF 2.3**

Use `smp.partition` context managers to place operations in specific partition. Any operation not placed in any `smp.partition` contexts is placed in the `default_partition`

```python
# SMP: Import TF2 API.
import smdistributed.modelparallel.tensorflow as smp

# SMP: Initialize
smp.init()

# SMP: Define the DistributedModel
class Model(smp.DistributedModel):
    def __init__(self):
        # define layers
```
def call(self, x):
    with smp.partition(0):
        x = self.layer0(x)
    with smp.partition(1):
        return self.layer1(x)

loss_obj = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)

# SMP: If the last batch will not be divisible by num. microbatches, set drop_remainder=True.
train_ds = tf.data.Dataset.from_tensor_slices(...).batch(128, drop_remainder=True)

# SMP: Define the smp.step consisting of forward and backward passes.
@smp.step
def forward_backward(images, labels):
    predictions = model(images, training=True)
    loss = loss_obj(labels, predictions)
    grads = optimizer.get_gradients(loss, model.trainable_variables)
    return grads, loss

@tf.function
def train_step(images, labels):
    gradients, loss = forward_backward(images, labels)

    # SMP: Accumulate the gradients across microbatches.
    gradients = [g.accumulate() for g in gradients]
    optimizer.apply_gradients(zip(gradients, model.trainable_variables))

    # SMP: Average the loss across microbatches.
    return loss.reduce_mean()

for images, labels in train_ds:
    loss = train_step(images, labels)

    # SMP: Only print loss at rank 0.
    if smp.rank() == 0:
        print(f"Loss: {loss}"

Modify a PyTorch Training Script

The following are examples of training scripts that you can use to configure SMP with PyTorch version 1.6, with auto-partitioning and manual partitioning. We recommend that you review the Important Considerations (p. 1078) and Unsupported Framework Features (p. 1079) before creating a training script.

Note that auto-partitioning is enabled by default. Unless otherwise specified, the following scripts use auto-partitioning.

Topics
- Important Considerations (p. 1078)
- Unsupported Framework Features (p. 1079)
- PyTorch 1.6.0 (p. 1080)
- Manual Partitioning with PyTorch 1.6.0 (p. 1081)

Important Considerations

When you configure a PyTorch training script, you should be aware of the following:
• If you are using an optimization technique that relies on global gradient norms, for example gradient norm from the entire model, such as some variants of LAMB optimizer or global gradient clipping, you need to gather all the norms across the model partitions for correctness. You can use SMP’s communication basic data types to do this.

• All torch.Tensor arguments to the forward methods of the nn.Modules in your model must be used in the computation of the module output. In other words, SMP does not support the case where there is a torch.Tensor argument to a module on which the module output does not depend.

• The argument to the smp.DistributedModel.backward() call must depend on all model outputs. In other words, there cannot be an output from the smp.DistributedModel.forward call that is not used in the computation of the tensor that is fed into the smp.DistributedModel.backward call.

• If there are torch.cuda.synchronize() calls in your code, you might need to call torch.cuda.set_device(smp.local_rank()) immediately before the synchronize call. Otherwise unnecessary CUDA contexts might be created in device 0, which will needlessly consume memory.

• Since SMP places nn.Modules on different devices, the modules in the model must not depend on any global state that is modified inside smp.step. Any state that remains fixed throughout training, or that is modified outside smp.step in a way that is visible to all processes, is allowed.

• You don’t need to move the model to GPU (using .to() API) when using SMP. If you try to move the model to GPU before the model is partitioned (before the first smp.step call), the move call is ignored. SMP automatically moves the part of the model assigned to a rank to its GPU. Once training with SMP starts, don’t move the model to CPU and use it, as it won’t have correct parameters for modules not assigned to the partition held by the process. If you want to use an SMP-trained model without SMP later, such as for inference, the recommended way is to save the full model using our checkpointing API and load it back to a regular PyTorch Module.

• If you have a list of modules such that output of one feeds into another, replacing that list with nn.Sequential can significantly improve performance.

• The weight update (optimizer.step()) needs to happen outside of smp.step because that is when the entire backward pass is done and gradients are ready. When using a hybrid model with model and data parallelism, at this point, Allreduce of gradients is also guaranteed to finish.

• When using SMP in combination with data parallelism, make sure that the number of batches on all data parallel ranks is the same so that Allreduce does not hang waiting for a rank which is not participating in the step.

• The inputs to smp.step must be the model inputs generated by DataLoader. This is because smp.step internally splits the input tensors along the batch dimension and pipelines them. This means that passing DataLoader itself to the smp.step function to generate the model inputs inside does not work.

• The input tensors to smp.step must be moved to the current device using .to() API, which must take place after the torch.cuda.set_device(local_rank()) call.

Unsupported Framework Features

The following PyTorch features are unsupported by SMP:

• When using data parallelism with DDP, the DistributedDataParallel wrapper is not supported. SMP internally manages integrating with DDP, including parameter broadcast and gradient Allreduce. Please note that with SMP, module buffers are only broadcast once at the start of training. If your model has module buffers and you require module buffers to be synchronized across data parallel groups at each step, you can do so through the torch.distributed API, using the process group that can be obtained via smp.get_dp_process_group().

• For mixed precision training, the apex.amp module is not supported. The recommended way to use SMP with automatic mixed-precision is to use torch.cuda.amp, with the exception of using smp.amp.GradScaler instead of the implementation in torch.

• torch.jit.ScriptModules or ScriptFunctions are not supported by smp.DistributedModel.
• apex: FusedLayerNorm, FusedAdam, FusedLAMB, and FusedNovoGrad from apex are not supported. You can use the SMP implementations of these through smp.optimizers and smp.nn APIs instead.

PyTorch 1.6.0

The following changes are required to run a PyTorch model with SMP:

1. Import and initialize SMP with smp.init().
2. Wrap the model with smp.DistributedModel.
3. Wrap the optimizer with smp.DistributedOptimizer.
4. Use the returned DistributedModel object instead of a user model.
5. Put the forward and backward logic in a step function and decorate it with smp.step.
6. Restrict each process to its own device through torch.cuda.set_device(smp.local_rank()).
7. Move the input tensors to the GPU using the .to() API before the smp.step call (see example below).
9. Perform post-processing on the outputs across microbatches using StepOutput methods such as reduce_mean.

```python
# SMP: Import SMP API
import smdistributed.modelparallel.torch as smp

# SMP: Initialize SMP.
smp.init()

class Model(nn.Module):
    def __init__(self):
        # define child modules

    def forward(self, x):
        # define forward pass

model = Model()

# Download dataset and create dataloader
dataset2 = datasets.MNIST("../data", train=False, transform=transform)
train_loader = torch.utils.data.DataLoader(dataset2, drop_last=True, **kwargs)

# SMP: Instantiate DistributedModel object using the model.
# This handles distributing the model among multiple ranks
# behind the scenes.
# If ddp is enabled this will take care of broadcasting parameters
# and does overlapping_all_reduce by default.
model = smp.DistributedModel(model)

# SMP: Define the smp.step consisting of forward and backward passes.
@smp.step()
def forward_backward(model, images, labels):
    loss = model(images, labels)
    # SMP: Call backward on the model instead of the output tensor.
    # If loss is not a scalar or a 0d tensor, the backward call requires
    # out grad tensors in addition to the output tensors,
    # similar to torch.autograd.backward call.
    model.backward(loss)
    return loss
```
for batch_idx, (image, label) in enumerate(train_loader):
    image, label = image.to(device), label.to(device)
    optimizer.zero_grad()
    loss_mb = forward_backward(model, image, label)

    # SMP: average the loss across microbatches
    loss = loss_mb.reduce_mean()
    optimizer.step()

    # SMP: print the loss only at rank 0
    if smp.rank() == 0:
        print(f"Loss: {loss}"

Manual Partitioning with PyTorch 1.6.0

Use smp.partition context managers to place modules in specific devices. Any module not placed
in any smp.partition contexts is placed in the default_partition. The default_partition
needs to be provided if auto_partition is set to False. The modules that are created within a specific
smp.partition context are placed on the corresponding partition.

# SMP: Import SMP API.
import smdistributed.modelparallel.torch as smp

# SMP: Initialize SMP
smp.init()

class Model(nn.Module):
    def __init__(self):
        with smp.partition(0):
            # define child modules on device 0
        with smp.partition(1):
            # define child modules on device 1
        def forward(self, x):
            # define forward pass
            return output

model = Model()

# Download dataset and create dataloader.
dataset2 = datasets.MNIST("../data", train=False, transform=transform)
train_loader = torch.utils.data.DataLoader(dataset2, drop_last=True, **kwargs)

# SMP: Instantiate DistributedModel object using the model.
# This handles distributing the model among multiple ranks
# behind the scenes.
# If horovod is enabled this will do an overlapping_all_reduce by
# default.
model = smp.DistributedModel(model)

# SMP: Define the smp.step consisting of forward and backward passes.
@smp.step()
def forward_backward(model, images, labels):
    loss = model(images, labels)
    # SMP: Call backward on the model instead of the output tensor.
    # If loss is not a scalar or a 0d tensor, the backward call requires
    # out grad tensors in addition to the output tensors,
    # similar to torch.autograd.backward call.
    model.backward(loss)
    return loss

for batch_idx, (image, label) in enumerate(train_loader):
    # SMP: move the inputs to the current device
    image, label = image.to(device), label.to(device)
    optimizer.zero_grad()
loss_mb = forward_backward(model, images, labels)

# SMP: Average the loss across microbatches.
loss = loss_mb.reduce_mean()
optimizer.step()

# SMP: Print the loss only at rank 0.
if smp.rank() == 0:
    print(f"Loss: {loss}"

---

**Configuration Tips and Pitfalls**

Review the following tips and pitfalls before using SMP. This list includes tips that are applicable across frameworks. For TensorFlow and PyTorch specific tips, see [Modify a TensorFlow Training Script](p. 1075) and [Modify a PyTorch Training Script](p. 1078), respectively.

**Batch Size and Number of Microbatches**

- SMP is most efficient when the batch size is increased. For use cases where the model fits within a single device, but can only be trained with a small batch size, batch size can and should be increased after SMP is integrated. Model parallelism saves memory for large models, enabling you to train using batch sizes that previously did not fit in memory.

- Choosing a number of microbatches that is too small or too large can hurt performance. SMP executes each microbatch sequentially in each device, so microbatch size (batch size divided by number of microbatches) must be large enough to fully utilize each GPU. At the same time, pipeline efficiency increases with the number of microbatches, so striking the right balance is important. Typically, a good starting point is to try 2 or 4 microbatches, increasing the batch size to the memory limit, and then experiment with larger batch sizes and numbers of microbatches. As the number of microbatches is increased, larger batch sizes might become feasible if an interleaved pipeline is used.

- Your batch size must be always divisible by the number of microbatches. Note that depending on the size of the dataset, sometimes the last batch of every epoch can be of a smaller size than the rest, and this smaller batch needs to be divisible by the number of microbatches as well. If it is not, you can set `drop_remainder=True` in the `tf.Dataset.batch()` call (in TensorFlow), or set `drop_last=True` in `DataLoader` (in PyTorch), so that this last, small batch is not used. If you are using a different API for the data pipeline, you might need to manually skip the last batch whenever it is not divisible by the number of microbatches.

**Manual Partitioning**

- If you use manual partitioning, be mindful of the parameters that are consumed by multiple operations and modules in your model, such as the embedding table in transformer architectures. Modules that share the same parameter must be placed in the same device for correctness. When auto-partitioning is used, SMP automatically enforces this constraint.

**Data Preparation**

- If the model takes multiple inputs, make sure you seed the random operations in your data pipeline (e.g., shuffling) with `smp.dp_rank()`. If the dataset is being deterministically sharded across data parallel devices, make sure that the shard is indexed by `smp.dp_rank()`. This is to make sure that the order of the data seen on all ranks that form a model partition is consistent.

**Returning Tensors from smp.DistributedModel**

- Any tensor that is returned from the `smp.DistributedModel.call` (for TensorFlow) or `smp.DistributedModel.forward` (for PyTorch) function is broadcast to all other ranks, from the rank that computed the particular tensor. As a result, any tensor that is not needed outside the call
and forward methods (intermediate activations, for example) should not be returned, as this causes needless communication and memory overhead and hurts performance.

**Distributed Training Jupyter Notebook Examples**

The following notebooks provide example Amazon SageMaker distributed implementations for popular deep learning frameworks and models. For vision (image) models, try MNIST or MaskRCNN. For language (text) models, try BERT.

These notebooks are provided in the [SageMaker examples repository](https://github.com/aws/sagemaker-examples). You can also browse them on the SageMaker examples website.

The examples are set up to use `p3.16xlarge` instances for the worker nodes, but you may choose other `p3` instance types if you wish. You can test a notebook using a cluster with only 1 node, but to see any performance benefit, you should use a cluster of multiple nodes (2 or more). The examples call out the section in which you modify this configuration.

**PyTorch Examples**

**SageMaker Distributed Data Parallel (SDP)**
- MNIST with PyTorch 1.6 and SDP
- MaskRCNN with PyTorch 1.6 and SDP
- BERT with PyTorch 1.6 and SDP

**SageMaker Distributed Model Parallel (SMP)**
- MNIST with PyTorch 1.6 and SMP
- BERT with PyTorch 1.6 and SMP

**TensorFlow Examples**

**SageMaker Distributed Data Parallel (SDP)**
- MNIST with TensorFlow 2.3.1 and SDP
- MaskRCNN with TensorFlow 2.3.1 and SDP
- BERT with TensorFlow 2.3.1 and SDP

**SageMaker Distributed Model Parallel (SMP)**
- MNIST with TensorFlow 2.3.1 and SMP

**Use a SageMaker Notebook Instance**

To use the provided examples, we recommend that you use an Amazon SageMaker notebook instance. A notebook instance is a machine learning (ML)–optimized Amazon EC2 instance running the Jupyter Notebook and JupyterServer apps. If you do not have an active notebook instance, follow the instructions in [Create a Notebook Instance](https://docs.aws.amazon.com/sagemaker/latest/dg/nb_instance.html) in the SageMaker developer guide to create one.

After you have created an instance, in the **Notebook instances** area of the SageMaker console, do the following:
1. Open JupyterLab.
2. Select the examples icon ( ) in the left tray.
3. Browse the examples for **Training** and look for notebooks titled *Distributed Data Parallel* or *Distributed Model Parallel*.

### Use SageMaker Studio

You can run these example Jupyter Notebooks in SageMaker Studio. To download and use an example notebook, do the following in Studio:

1. Open a terminal.
2. In the command line, navigate to the SageMaker folder.
   ```bash
   $ cd SageMaker
   ```
3. Clone the SageMaker examples repo.
   ```bash
   git clone https://github.com/aws/amazon-sagemaker-examples.git
   ```
4. In the JupyterLab interface, navigate into the `amazon-sagemaker-examples` folder.
5. In the `training/distributed_training` folder, there are folders for frameworks, and in each of these, there are folders for `data_parallel` and `model_parallel`. Choose the example of your choice and follow the instructions to launch distributed model training with SDP on SageMaker.

### Troubleshooting

If you run into an error, you can use the following list to try to troubleshoot your training job. If the problem persists, contact AWS Support.

- When Debugger is enabled (which is enabled by default for all SageMaker TensorFlow and PyTorch jobs) you might see an error that looks like the following:
  ```bash
  FileNot_foundError: [Errno 2] No such file or directory: '/opt/ml/checkpoints/metadata.json.sagemaker-uploading
  ```
  To fix this issue, disable Debugger by passing `debugger_hook_config=False` when creating the framework estimator in the SageMaker Python SDK as seen in the following example.

  ```python
  estimator = TensorFlow(
      role=role,
      instance_count=1,
      instance_type=instance_type,
      debugger_hook_config=False
  )
  ```

- You might run into the following error when saving checkpoints of a large model on SageMaker:
  ```bash
  InternalServerError: We encountered an internal error. Please try again
  ```
  This could be caused by a SageMaker limitation while uploading the local checkpoint to Amazon S3 during training. To disable checkpointing in SageMaker and follow the following example to explicitly upload the checkpoints.
If you run into above error, do not use `checkpoint_s3_uri` with the SageMaker estimator call. While saving checkpoints for larger models, we recommend saving checkpoints to a custom directory and passing the same to the helper function (as a `local_path` argument).

```python
import os

def aws_s3_sync(source, destination):
    """aws s3 sync in quiet mode and time profile""
    import time, subprocess
    cmd = ['aws', 's3', 'sync', '--quiet', source, destination]
    start_time = time.time()
    p = subprocess.Popen(cmd, stdout=subprocess.PIPE, stderr=subprocess.PIPE)
    p.wait()
    end_time = time.time()
    print("Time Taken to Sync: ", (end_time-start_time))
    return

def sync_local_checkpoints_to_s3(local_path="/opt/ml/checkpoints",
                                s3_uri=os.path.dirname(os.path.dirname(os.getenv('SM_MODULE_DIR', '')))+'/checkpoints'):
    """ sample function to sync checkpoints from local path to s3 """

    import boto3
    #check if local path exists
    if not os.path.exists(local_path):
        raise RuntimeError("Provided local path {local_path} does not exist. Please check")

    #check if s3 bucket exists
    s3 = boto3.resource('s3')
    if not s3_uri.startswith("s3://"):
        raise ValueError(f"Provided s3 uri {s3_uri} is not valid.")

    s3_bucket = s3_uri.replace("s3://", '').split('/')[0]
    print(f"S3 Bucket: {s3_bucket}")
    try:
        s3.meta.client.head_bucket(Bucket=s3_bucket)
    except Exception as e:
        raise e
    aws_s3_sync(local_path, s3_uri)
    return

def sync_s3_checkpoints_to_local(local_path="/opt/ml/checkpoints",
                                 s3_uri=os.path.dirname(os.path.dirname(os.getenv('SM_MODULE_DIR', '')))+'/checkpoints'):
    """ sample function to sync checkpoints from s3 to local path """

    import boto3
    #try to create local path if it does not exist
    if not os.path.exists(local_path):
        print("Provided local path {local_path} does not exist. Creating...")
        try:
            os.makedirs(local_path)
        except Exception as e:
            raise RuntimeError(f"Failed to create {local_path}")

    #check if s3 bucket exists
    s3 = boto3.resource('s3')
    if not s3_uri.startswith("s3://"):
        raise ValueError(f"Provided s3 uri {s3_uri} is not valid.")

    s3_bucket = s3_uri.replace("s3://", '').split('/')[0]
    print(f"S3 Bucket: {s3_bucket}")
    try:
        s3.meta.client.head_bucket(Bucket=s3_bucket)
    except Exception as e:
        raise e
```

1085
except Exception as e:
    raise e
aws_s3_sync(s3_uri, local_path)
return

Usage of helper functions:

```python
#base_s3_uri - user input s3 uri or save to model directory (default)
curr_host - to save checkpoints of current host
iteration - current step/epoch during which checkpoint is saved

# save checkpoints on every node using local_rank
if smp.local_rank() == 0:
    base_s3_uri = os.path.dirname(os.path.dirname(os.getenv('SM_MODULE_DIR', '')))
curr_host = os.environ['SM_CURRENT_HOST']
full_s3_uri = f'{base_s3_uri}/checkpoints/{curr_host}/{iteration}'
sync_local_checkpoints_to_s3(local_path=checkpoint_dir, s3_uri=full_s3_uri)
```

Detect Posttraining Data and Model Bias

Posttraining bias analysis can help reveal biases that might have emanated from biases in the data, or from biases introduced by the classification and prediction algorithms. These analyzes take into consideration the data, including the labels, and the predictions of a model. You assess performance by analyzing predicted labels or by comparing the predictions with the observed target values in the data with respect to groups with different attributes. There are different notions of fairness, each requiring different bias metrics to measure.

There are legal concepts of fairness that might not be easy to capture because they are hard to detect. For example, the US concept of disparate impact that occurs when a group, referred to as a less favored facet, experiences an adverse effect even when the approach taken appears to be fair. This type of bias might not be due to a machine leaning model, but might still be detectable by posttraining bias analysis.

Amazon SageMaker Clarify tries to ensure a consistent use of terminology. For a list of terms and their definitions, see Amazon SageMaker Clarify Terms for Bias and Fairness (p. 584).

For additional information about posttraining bias metrics, see Fairness Measures for Machine Learning in Finance.

Sample Notebooks

Amazon SageMaker Clarify provides the following sample notebook for posttraining bias detection:

- Explainability and bias detection with Amazon SageMaker Clarify – Use SageMaker Clarify to create a processing job for the detecting bias and explaining model predictions with feature attributions.

This notebook has been verified to run in Amazon SageMaker Studio only. If you need instructions on how to open a notebook in Amazon SageMaker Studio, see Create or Open an Amazon SageMaker Studio Notebook (p. 87). If you’re prompted to choose a kernel, choose Python 3 (Data Science).

Topics

- Measure Posttraining Data and Model Bias (p. 1087)
- Configure an Amazon SageMaker Clarify Processing Jobs for Fairness and Explainability (p. 1105)
Measure Posttraining Data and Model Bias

Amazon SageMaker Clarify provides eleven posttraining data and model bias metrics to help quantify various conceptions of fairness. These concepts cannot all be satisfied simultaneously and the selection depends on specifics of the cases involving potential bias being analyzed. Most of these metrics are a combination of the numbers taken from the binary classification confusion matrices for the different demographic groups. Because fairness and bias can be defined by a wide range of metrics, human judgment is required to understand and choose which metrics are relevant to the individual use case, and customers should consult with appropriate stakeholders to determine the appropriate measure of fairness for their application.

We use the following notation to discuss the bias metrics. The conceptual model described here is for binary classification, where events are labeled as having only two possible outcomes in their sample space, referred to as positive (with value 1) and negative (with value 0). This framework is usually extensible to multicategory classification in a straightforward way or to cases involving continuous valued outcomes when needed. In the binary classification case, positive and negative labels are assigned to outcomes recorded in a raw dataset for a favored facet \( a \) and for a disfavored facet \( d \). These labels \( y \) are referred to as observed labels to distinguish them from the predicted labels \( y' \) that are assigned by a machine learning model during the training or inference stages of the ML lifecycle. These labels are used to define probability distributions \( P_a(y) \) and \( P_d(y) \) for their respective facet outcomes.

- **labels:**
  - \( y \) represents the \( n \) observed labels for event outcomes in a training dataset.
  - \( y' \) represents the predicted labels for the \( n \) observed labels in the dataset by a trained model.

- **outcomes:**
  - A positive outcome (with value 1) for a sample, such as an application acceptance.
    - \( n^{(1)} \) is the number of observed labels for positive outcomes (acceptances).
    - \( n'^{(1)} \) is the number of predicted labels for positive outcomes (acceptances).
  - A negative outcome (with value 0) for a sample, such as an application rejection.
    - \( n^{(0)} \) is the number of observed labels for negative outcomes (rejections).
    - \( n'^{(0)} \) is the number of predicted labels for negative outcomes (rejections).

- **facet values:**
  - facet \( a \) – The feature value that defines a demographic that bias favors.
    - \( n_a \) is the number of observed labels for the favored facet value: \( n_a = n_a^{(1)} + n_a^{(0)} \) the sum of the positive and negative observed labels for the value facet \( a \).
    - \( n'_a \) is the number of predicted labels for the favored facet value: \( n'_a = n'_a^{(1)} + n'_a^{(0)} \) the sum of the positive and negative predicted outcome labels for the facet value \( a \).
  - facet \( d \) – The feature value that defines a demographic that bias disfavors.
    - \( n_d \) is the number of observed labels for the disfavored facet value: \( n_d = n_d^{(1)} + n_d^{(0)} \) the sum of the positive and negative observed labels for the value facet \( d \).
    - \( n'_d \) is the number of predicted labels for the disfavored facet value: \( n'_d = n'_d^{(1)} + n'_d^{(0)} \) the sum of the positive and negative predicted labels for the facet value \( d \).

- **probability distributions for outcomes of the labeled facet data outcomes:**
  - \( P_a(y) \) is the probability distribution of the observed labels for facet \( a \). For binary labeled data, this distribution is given by the ratio of the number of samples in facet \( a \) labeled with positive outcomes to the total number, \( P_a(y^{(1)}) = n_a^{(1)}/n_a \), and the ratio of the number of samples with negative outcomes to the total number, \( P_a(y^{(0)}) = n_a^{(0)}/n_a \).
  - \( P_d(y) \) is the probability distribution of the observed labels for facet \( d \). For binary labeled data, this distribution is given by the number of samples in facet \( d \) labeled with positive outcomes to the total...
number, \( P_d(y^1) = \frac{n_d^{(1)}}{n_d} \) and the ratio of the number of samples with negative outcomes to the total number, \( P_d(y^0) = \frac{n_d^{(0)}}{n_d} \).

The following table contains a cheat sheet for quick guidance and links to the posttraining bias metrics.

### Posttraining Bias Metrics

<table>
<thead>
<tr>
<th>Posttraining bias metric</th>
<th>Description</th>
<th>Example question</th>
<th>Interpreting metric values</th>
</tr>
</thead>
</table>
| **Difference in Positive Proportions in Predicted Labels (DPPL) (p. 1095)** | Measures the difference in the proportion of positive predictions between the favored facet \( a \) and the disfavored facet \( d \). | Has there been an imbalance across demographic groups in the predicted positive outcomes that might indicate bias? | Range for normalized binary & multicategory facet labels: \([-1,+1]\) Range for continuous labels: \((-\infty, +\infty)\) Interpretation:  
  - Positive values indicate that the favored facet \( a \) has a higher proportion of predicted positive outcomes.  
  - Values near zero indicate a more equal proportion of predicted positive outcomes between facets.  
  - Negative values indicate the disfavored facet \( d \) has a higher proportion of predicted positive outcomes. |
| **Disparate Impact (DI) (p. 1096)** | Measures the ratio of proportions of the predicted labels for the favored facet \( a \) and the disfavored facet \( d \). | Has there been an imbalance across demographic groups in the predicted positive outcomes that might indicate bias? | Range for normalized binary, multicategory facet, and continuous labels: \([0,\infty)\) Interpretation:  
  - Values greater than 1 indicate the favored facet \( a \) has a higher proportion of predicted positive outcomes.  
  - A value of 1 indicates that we have demographic parity.  
  - Values less than 1 indicate the disfavored facet \( d \) has... |
<table>
<thead>
<tr>
<th>posttraining bias metric</th>
<th>Description</th>
<th>Example question</th>
<th>Interpreting metric values</th>
</tr>
</thead>
</table>
| Difference in Conditional Acceptance (DCAcc) (p. 1096) | Compares the observed labels to the labels predicted by a model and assesses whether this is the same across facets for predicted positive outcomes (acceptances). | Are there more or less acceptances for loan applications than predicted for one age group as compared to another based on qualifications? | The range for binary, multicategory facet, and continuous labels: $(-\infty, +\infty)$.  
• Positive values indicate a possible bias against the qualified applicants from the disfavored facet $d$.  
• Values near zero indicate that qualified applicants from both facets are being accepted in a similar way.  
• Negative values indicate a possible bias against the qualified applicants from the favored facet $a$. |
| Difference in Conditional Rejection (DCR) (p. 1097) | Compares the observed labels to the labels predicted by a model and assesses whether this is the same across facets for negative outcomes (rejections). | Are there more or less rejections for loan applications than predicted for one age group as compared to another based on qualifications? | The range for binary, multicategory facet, and continuous labels: $(-\infty, +\infty)$.  
• Positive values indicate a possible bias against the qualified applicants from the disfavored facet $d$.  
• Values near zero indicate that qualified applicants from both facets are being rejected in a similar way.  
• Negative values indicate a possible bias against the qualified applicants from the favored facet $a$. |
<table>
<thead>
<tr>
<th>Posttraining Bias Metric</th>
<th>Description</th>
<th>Example Question</th>
<th>Interpreting Metric Values</th>
</tr>
</thead>
</table>
| Recall Difference (RD) (p. 1099) | Compares the recall of the model for the favored and disfavored facets. | Is there an age-based bias in lending due to a model having higher recall for one age group as compared to another? | Range for binary and multicategory classification: [-1, +1].
  
  - Positive values suggest that the model finds more of the true positives for facet \( a \) and is biased against the disfavored facet \( d \).
  
  - Values near zero suggest that the model finds about the same number of true positives in both facets and is not biased.
  
  - Negative values suggest that the model finds more of the true positives for facet \( d \) and is biased against the favored facet \( a \).
<table>
<thead>
<tr>
<th>posttraining bias metric</th>
<th>Description</th>
<th>Example question</th>
<th>Interpreting metric values</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Difference in Acceptance Rates (DAR) (p. 1100)</strong></td>
<td>Measures the difference in the ratios of the observed positive outcomes (TP) to the predicted positives (TP + FP) between the favored and disfavored facets.</td>
<td>Does the model have equal precision when predicting loan acceptances for qualified applicants across all age groups?</td>
<td>The range for binary, multicategory facet, and continuous labels is [-1, +1].</td>
</tr>
</tbody>
</table>

- Positive values indicate a possible bias against facet `d` caused by the occurrence of relatively more false positives in the disfavored facet `d`.  
- Values near zero indicate the observed labels for positive outcomes (acceptances) are being predicted with equal precision for both facets by the model.  
- Negative values indicate a possible bias against facet `a` caused by the occurrence of relatively more false positives in the favored facet `a`. |
<table>
<thead>
<tr>
<th>posttraining bias metric</th>
<th>Description</th>
<th>Example question</th>
<th>Interpreting metric values</th>
</tr>
</thead>
</table>
| Difference in Rejection Rates (DRR) (p. 1100) | Measures the difference in the ratios of the observed negative outcomes (TN) to the predicted negatives (TN + FN) between the disfavored and favored facets. | Does the model have equal precision when predicting loan rejections for unqualified applicants across all age groups? | The range for binary, multicategory facet, and continuous labels is [-1, +1].
  - Positive values indicate a possible bias caused by the occurrence of relatively more false negatives in the favored facet.  
  - Values near zero indicate the observed labels for negative outcomes (rejections) are being predicted with equal precision for both facets by the model.  
  - Negative values indicate a possible bias caused by the occurrence of relatively more false negatives in the disfavored facet. |
<table>
<thead>
<tr>
<th>posttraining bias metric</th>
<th>Description</th>
<th>Example question</th>
<th>Interpreting metric values</th>
</tr>
</thead>
</table>
| **Accuracy Difference (AD) (p. 1101)** | Measures the difference between the prediction accuracy for the favored and disfavored facets. | Does the model predict labels as accurately for applications across all demographic groups? | The range for binary and multicategory facet labels is [-1, +1].  
• Positive values indicate that facet \(d\) suffers more from some combination of false positives (Type I errors) or false negatives (Type II errors). This means there is a potential bias against the disfavored facet \(d\).  
• Values near zero occur when the prediction accuracy for facet \(a\) is similar to that for facet \(d\).  
• Negative values indicate that facet \(a\) suffers more from some combination of false positives (Type I errors) or false negatives (Type II errors). This means there is a bias against the favored facet \(a\). |
| **Treatment Equality (TE) (p. 1102)** | Measures the difference in the ratio of false positives to false negatives between the favored and disfavored facets. | In loan applications, is the relative ratio of false positives to false negatives the same across all age demographics? | The range for binary and multicategory facet labels: \((-\infty, +\infty)\).  
• Positive values occur when the ratio of false positives to false negatives for facet \(a\) is greater than that for facet \(d\).  
• Values near zero occur when the ratio of false positives to false negatives for facet \(a\) is similar to that for facet \(d\).  
• Negative values occur when the ratio of false positives to false negatives for facet \(a\) is less than that for facet \(d\). |
<table>
<thead>
<tr>
<th>posttraining bias metric</th>
<th>Description</th>
<th>Example question</th>
<th>Interpreting metric values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conditional Demographic Disparity in Predicted Labels (CDDPL) (p. 1103)</td>
<td>Measures the disparity of predicted labels between the facets as a whole, but also by subgroups.</td>
<td>Do some demographic groups have a larger proportion of rejections for loan application outcomes than their proportion of acceptances?</td>
<td>The range of CDDPL values for binary, multicategory, and continuous outcomes: [-1, +1]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Positive values indicate an outcomes where facet $d$ is rejected more than accepted.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Near zero indicates no demographic disparity on average.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Negative values indicate an outcomes where facet $a$ is rejected more than accepted.</td>
</tr>
<tr>
<td>Counterfactual Fliptest (FT) (p. 1104)</td>
<td>Examines each member of facet $d$ and assesses whether similar members of facet $a$ have different model predictions.</td>
<td>Are a group of a specific age demographic, matched closely on all features with a another age group, paid on average more than that other age group?*</td>
<td>The range for binary and multicategory facet labels is [-1, +1].</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Positive values occur when the number of unfavorable counterfactual fliptest decisions for the disfavored facet $d$ exceeds the favorable ones.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Values near zero occur when the number of unfavorable and favorable counterfactual fliptest decisions balance out.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Negative values occur when the number of unfavorable counterfactual fliptest decisions for the disfavored facet $d$ is less than the favorable ones.</td>
</tr>
</tbody>
</table>

For additional information about posttraining bias metrics, see A Family of Fairness Measures for Machine Learning in Finance.
more false negatives in the disfavored facet d.

- Difference in Positive Proportions in Predicted Labels (DPPL) (p. 1095)
- Disparate Impact (DI) (p. 1096)
- Difference in Conditional Acceptance (DCAcc) (p. 1096)
- Difference in Conditional Rejection (DCR) (p. 1097)
- Recall Difference (RD) (p. 1099)
- Difference in Acceptance Rates (DAR) (p. 1100)
- Difference in Rejection Rates (DRR) (p. 1100)
- Accuracy Difference (AD) (p. 1101)
- Treatment Equality (TE) (p. 1102)
- Conditional Demographic Disparity in Predicted Labels (CDDPL) (p. 1103)
- Counterfactual Flipped Test (FT) (p. 1104)

**Difference in Positive Proportions in Predicted Labels (DPPL)**

The difference in positive proportions in predicted labels (DPPL) metric determines whether the model predicts outcomes differently for each facet. It is defined as the difference between the proportion of positive predictions \( y' = 1 \) for facet \( a \) and the proportion of positive predictions \( y' = 1 \) for facet \( d \). For example, if the model predictions grant loans to 60% of a middle-aged group (facet \( a \)) and 50% other age groups (facet \( d \)), it might be biased against facet \( d \). In this example, you need to determine whether the 10% difference is material to a case for bias. A comparison of DPL with DPPL assesses whether bias initially present in the dataset increases or decreases in the model predictions after training.

The formula for the difference in proportions of predicted labels:

\[
DPPL = q'_a - q'_d
\]

Where:

- \( q'_a = n'_a(1)/n_a \) is the predicted proportion of facet \( a \) who get a positive outcome of value 1. In our example, the proportion of a middle-aged facet predicted to get granted a loan. Here \( n'_a(1) \) represents the number of members of facet \( a \) who get a positive predicted outcome of value 1 and \( n_a \) the is number of members of facet \( a \).

- \( q'_d = n'_d(1)/n_d \) is the predicted proportion of facet \( d \) who get a positive outcome of value 1. In our example, a facet of older and younger people predicted to get granted a loan. Here \( n'_d(1) \) represents the number of members of facet \( d \) who get a positive predicted outcome and \( n_d \) the is number of members of facet \( d \).

If DPPL is close enough to 0, it means that posttraining demographic parity has been achieved.

For binary and multiclass facet labels, the normalized DPL values range over the interval \([-1, 1]\). For continuous labels, the values vary over the interval \((-\infty, +\infty)\).

- Positive DPPL values indicate that facet \( a \) has a higher proportion of predicted positive outcomes when compared with facet \( d \).
  
  This is referred to as **positive bias**.

- Values of DPPL near zero indicate a more equal proportion of predicted positive outcomes between facets \( a \) and \( d \) and a value of zero indicates perfect demographic parity.

- Negative DPPL values indicate that facet \( d \) has a higher proportion of predicted positive outcomes when compared with facet \( a \). This is referred to as **negative bias**.
Disparate Impact (DI)

The difference in positive proportions in the predicted labels metric can be assessed in the form of a ratio.

The comparison of positive proportions in predicted labels metric can be assessed in the form of a ratio instead of as a difference, as it is with the Difference in Positive Proportions in Predicted Labels (DPPL) (p. 1095). The disparate impact (DI) metric is defined as the ratio of the proportion of positive predictions (y’ = 1) for facet a over the proportion of positive predictions (y’ = 1) for facet d. For example, if the model predictions grant loans to 60% of a middle-aged group (facet a) and 50% other age groups (facet d), then DI = .5/.6 = 0.8, which indicates a positive bias and an adverse impact on facet d.

The formula for the ratio of proportions of the predicted labels:

\[ \text{DI} = \frac{q'_a}{q'_d} \]

Where:
- \( q'_a = \frac{n'_a(1)}{n_a} \) is the predicted proportion of facet a who get a positive outcome of value 1. In our example, the proportion of a middle-aged facet predicted to get granted a loan. Here \( n'_a(1) \) represents the number of members of facet a who get a positive predicted outcome and \( n_a \) the is number of members of facet a.
- \( q'_d = \frac{n'_d(1)}{n_d} \) is the predicted proportion of facet d who get a positive outcome of value 1. In our example, a facet of older and younger people predicted to get granted a loan. Here \( n'_d(1) \) represents the number of members of facet d who get a positive predicted outcome and \( n_d \) the is number of members of facet d.

For binary, multicategory facet, and continuous labels, the DI values range over the interval \([0, \infty)\).

- Values less than 1 indicate that facet a has a higher proportion of predicted positive outcomes than facet d. This is referred to as positive bias.
- A value of 1 indicates demographic parity.
- Values greater than 1 indicate that facet d has a higher proportion of predicted positive outcomes than facet a. This is referred to as negative bias.

Difference in Conditional Acceptance (DCAcc)

This metric compares the observed labels to the labels predicted by the model and assesses whether this is the same across facets for predicted positive outcomes. This metric comes close to mimicking human bias in that it quantifies how many more positive outcomes a model predicted (labels y’) for a certain facet as compared to what was observed in the training dataset (labels y). For example, if there were more acceptances (a positive outcome) for loan applications for a middle-aged group (facet a) than predicted by the model based on qualifications as compared to the facet containing other age groups (facet d), this might indicate potential bias in the way loans were approved.

The formula for the difference in conditional acceptance:

\[ \text{DCAcc} = c_a - c_d \]

Where:
- \( c_a = \frac{n_a(1)}{n'_a} \) is the ratio of the observed number of positive outcomes of value 1 (acceptances) of facet a to the predicted number of positive outcome (acceptances) for facet a.
- \( c_d = \frac{n_d(1)}{n'_d} \) is the ratio of the observed number of positive outcomes of value 1 (acceptances) of facet d to the predicted number of predicted positive outcomes (acceptances) for facet d.
The DCAcc metric can capture both positive and negative biases that reveal preferential treatment based on qualifications. Consider the following instances of age-based bias on loan acceptances.

**Example 1: Positive bias**

Suppose we have dataset of 100 middle-aged people (facet \(a\)) and 50 people from other age groups (facet \(d\)) who applied for loans, where the model recommended that 60 from facet \(a\) and 30 from facet \(d\) be given loans. So the predicted proportions are unbiased with respect to the DPPL metric, but the observed labels show that 70 from facet \(a\) and 20 from facet \(d\) were granted loans. In other words, the model granted loans to 17% more from the middle aged facet than the observed labels in the training data suggested (70/60 = 1.17) and granted loans to 33% less from other age groups than the observed labels suggested (20/30 = 0.67). The calculation of the DCAcc value quantifies this difference between +17% and -33%. The calculation of the DCAcc value gives the following:

\[
\text{DCAcc} = \frac{70}{60} - \frac{20}{30} = \frac{1}{2}
\]

**Example 2: Negative bias**

Suppose we have dataset of 100 middle-aged people (facet \(a\)) and 50 people from other age groups (facet \(d\)) who applied for loans, where the model recommended that 60 from facet \(a\) and 30 from facet \(d\) be given loans. So the predicted proportions are unbiased with respect to the DPPL metric, but the observed labels show that 50 from facet \(a\) and 40 from facet \(d\) were granted loans. In other words, the model granted loans to 17% fewer from the middle aged facet than the observed labels in the training data suggested (50/60 = 0.83), and granted loans to 33% more from other age groups than the observed labels suggested (40/30 = 1.33). The calculation of the DCAcc value quantifies this difference between -17% and +33%. The calculation of the DCAcc value gives the following:

\[
\text{DCAcc} = \frac{50}{60} - \frac{40}{30} = -\frac{1}{2}
\]

Note that you can use DCAcc to help you detect potential (unintentional) biases by humans overseeing the model predictions in a human-in-the-loop setting. Assume, for example, that the predictions \(y'\) by the model were unbiased, but the eventual decision is made by a human (possibly with access to additional features) who can alter the model predictions to generate a new and final version of \(y\). The additional processing by the human may unintentionally deny loans to a disproportionate number from one facet. DCAcc can help detect such potential biases.

The range of values for differences in conditional acceptance for binary, multicategory facet, and continuous labels is \((-\infty, +\infty)\).

- Positive values occur when the ratio of the observed number of acceptances compared to predicted acceptances for facet \(a\) is higher than the same ratio for facet \(d\). These values indicate a possible bias against the qualified applicants from facet \(d\). The larger the difference of the ratios, the more extreme the apparent bias.
- Values near zero occur when the ratio of the observed number of acceptances compared to predicted acceptances for facet \(a\) is similar to the ratio for facet \(d\). These values indicate that predicted acceptance rates are consistent with the observed values in the labeled data and that qualified applicants from both facets are being accepted in a similar way.
- Negative values occur when the ratio of the observed number of acceptances compared to predicted acceptances for facet \(a\) is less than that ratio for facet \(d\). These values indicate a possible bias against the qualified applicants from facet \(a\). The more negative the difference in the ratios, the more extreme the apparent bias.

**Difference in Conditional Rejection (DCR)**

This metric compares the observed labels to the labels predicted by the model and assesses whether this is the same across facets for negative outcomes (rejections). This metric comes close to mimicking human bias, in that it quantifies how many more negative outcomes a model granted (predicted labels \(y'\)) to a certain facet as compared to what was suggested by the labels in the training dataset (observed...
labels \( y \). For example, if there were more rejections (a negative outcome) for loan applications for a middle-aged group (facet \( a \)) than predicted by the model based on qualifications as compared to the facet containing other age groups (facet \( d \)), this might indicate potential bias in the way loans were rejected.

The formula for the difference in conditional acceptance:

\[
DCR = r_d - r_a
\]

Where:

- \( r_d = n_d^{(0)}/n_d^{(0)} \) is the ratio of the observed number of negative outcomes of value 1 (rejections) of facet \( d \) to the predicted number of negative outcomes (rejections) for facet \( d \).
- \( r_a = n_a^{(0)}/n_a^{(0)} \) is the ratio of the observed number of negative outcomes of value 0 (rejections) of facet \( a \) to the predicted number of negative outcome of value 0 (rejections) for facet \( a \).

The DCR metric can capture both positive and negative biases that reveal preferential treatment based on qualifications. Consider the following instances of age-based bias on loan rejections.

**Example 1: Positive bias**

Suppose we have dataset of 100 middle-aged people (facet \( a \)) and 50 people from other age groups (facet \( d \)) who applied for loans, where the model recommended that 60 from facet \( a \) and 30 from facet \( d \) be rejected for loans. So the predicted proportions are unbiased by the DPPL metric, but the observed labels show that 50 from facet \( a \) and 40 from facet \( d \) were rejected. In other words, the model rejected loans for 17% fewer from the middle aged facet than the observed labels in the training data suggested (50/60 = 0.83), and rejected loans for 33% more from other age groups than the observed labels suggested (40/30 = 1.33). The calculation of the DCR value quantifies this difference between -17% and +33%.

\[
DCR = 40/30 - 50/60 = 1/2
\]

**Example 2: Negative bias**

Suppose we have dataset of 100 middle-aged people (facet \( a \)) and 50 people from other age groups (facet \( d \)) who applied for loans, where the model recommended that 60 from facet \( a \) and 30 from facet \( d \) be rejected for loans. So the predicted proportions are unbiased by the DPPL metric, but the observed labels show that 70 from facet \( a \) and 20 from facet \( d \) were rejected. In other words, the model rejected loans for 17% more from the middle aged facet than the observed labels in the training data suggested (70/60 = 1.17), and rejected loans for 33% fewer from other age groups than the observed labels suggested (20/30 = 0.67). The calculation of the DCR value quantifies this difference between 17% and -33%.

\[
DCR = 20/30 - 70/60 = -1/2
\]

The range of values for differences in conditional rejection for binary, multicategory facet, and continuous labels is \((-\infty, +\infty)\).

- Positive values occur when the ratio of the observed number of rejections compared to predicted rejections for facet \( d \) is greater than that ratio for facet \( a \). These values indicate a possible bias against the qualified applicants from facet \( d \). The larger the value of DCR metric, the more extreme the apparent bias.
- Values near zero occur when the ratio of the observed number of rejections compared to predicted acceptances for facet \( a \) is similar to the ratio for facet \( d \). These values indicate that predicted rejections rates are consistent with the observed values in the labeled data and that the qualified applicants from both facets are being rejected in a similar way.
- Negative values occur when the ratio of the observed number of rejections compared to predicted rejections for facet \( d \) is less than that ratio for facet \( a \). These values indicate a possible bias against the
qualified applicants from facet $a$. The larger magnitude of the negative DCR metric, the more extreme the apparent bias.

**Recall Difference (RD)**

The recall difference (RD) metric is the difference in recall of the model between the favored facet $a$ and disfavored facet $d$. Any difference in these recalls is a potential form of bias. Recall is the true positive rate (TPR), which measures how often the model correctly predicts the cases that should receive a positive outcome. Recall is perfect for a facet if all of the $y=1$ cases are correctly predicted as $y'=1$ for that facet. Recall is greater when the model minimizes false negatives known as the Type II error. For example, how many of the people in two different groups (facets $a$ and $d$) that should qualify for loans are detected correctly by the model? If the recall rate is high for lending to facet $a$, but low for lending to facet $d$, the difference provides a measure of this bias against the group belonging to facet $d$.

The formula for difference in the recall rates for facets $a$ and $d$:

$$RD = \frac{TP_a}{TP_a + FN_a} - \frac{TP_d}{TP_d + FN_d} = TPR_a - TPR_d$$

Where:

- $TP_a$ are the true positives predicted for facet $a$.
- $FN_a$ are the false positives predicted for facet $a$.
- $TP_d$ are the true positives predicted for facet $d$.
- $FN_d$ are the false positives predicted for facet $d$.
- $TPR_a = \frac{TP_a}{TP_a + FN_a}$ is the recall for facet $a$ or its true positive rate.
- $TPR_d = \frac{TP_d}{TP_d + FN_d}$ is the recall for facet $d$ or its true positive rate.

For example, consider the following confusion matrices for facets $a$ and $d$.

**Confusion Matrix for the Favored Facet $a$**

<table>
<thead>
<tr>
<th>Class $a$ predictions</th>
<th>Actual outcome 0</th>
<th>Actual outcome 1</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>20</td>
<td>5</td>
<td>25</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>65</td>
<td>75</td>
</tr>
<tr>
<td>Total</td>
<td>30</td>
<td>70</td>
<td>100</td>
</tr>
</tbody>
</table>

**Confusion Matrix for the Disfavored Facet $d$**

<table>
<thead>
<tr>
<th>Class $d$ predictions</th>
<th>Actual outcome 0</th>
<th>Actual outcome 1</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>18</td>
<td>7</td>
<td>25</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>20</td>
<td>25</td>
</tr>
<tr>
<td>Total</td>
<td>23</td>
<td>27</td>
<td>50</td>
</tr>
</tbody>
</table>

The value of the recall difference is $RD = 65/70 - 20/27 = 0.93 - 0.74 = 0.19$ which indicates a bias against facet $d$.

The range of values for the recall difference between facets $a$ and $d$ for binary and multicategory classification is $[-1, +1]$. This metric is not available for the case of continuous labels.
Measure Posttraining Data and Model Bias

- Positive values are obtained when there is higher recall for facet $a$ than for facet $d$. This suggests that the model finds more of the true positives for facet $a$ than for facet $d$, which is a form of bias.
- Values near zero indicate that the recall for facets being compared is similar. This suggests that the model finds about the same number of true positives in both of these facets and is not biased.
- Negative values are obtained when there is higher recall for facet $d$ than for facet $a$. This suggests that the model finds more of the true positives for facet $d$ than for facet $a$, which is a form of bias.

### Difference in Acceptance Rates (DAR)

The difference in acceptance rates (DAR) metric is the difference in the ratios of the true positive (TP) predictions to the observed positives (TP + FP) for facets $a$ and $d$. This metric measures the difference in the precision of the model for predicting acceptances from these two facets. Precision measures the fraction of qualified candidates from the pool of qualified candidates that are identified as such by the model. If the model precision for predicting qualified applicants diverges between the facets, this is a bias and its magnitude is measured by the DAR.

The formula for difference in acceptance rates between facets $a$ and $d$:

$$\text{DAR} = \frac{\text{TP}_a}{\text{TP}_a + \text{FP}_a} - \frac{\text{TP}_d}{\text{TP}_d + \text{FP}_d}$$

Where:

- $\text{TP}_a$ are the true positives predicted for facet $a$.
- $\text{FP}_a$ are the false positives predicted for facet $a$.
- $\text{TP}_d$ are the true positives predicted for facet $d$.
- $\text{FP}_d$ are the false positives predicted for facet $d$.

For example, suppose the model accepts 70 middle-aged applicants (facet $a$) for a loan (predicted positive labels) of whom only 35 are actually accepted (observed positive labels). Also suppose the model accepts 100 applicants from other age demographics (facet $d$) for a loan (predicted positive labels) of whom only 40 are actually accepted (observed positive labels). Then DAR = $\frac{35}{70} - \frac{40}{100} = 0.10$, which indicates a potential bias against qualified people from the second age group (facet $d$).

The range of values for DAR for binary, multicategory facet, and continuous labels is $[-1, +1]$.

- Positive values occur when the ratio of the predicted positives (acceptances) to the observed positive outcomes (qualified applicants) for facet $a$ is larger than the same ratio for facet $d$. These values indicate a possible bias against the disfavored facet $d$ caused by the occurrence of relatively more false positives in facet $d$. The larger the difference in the ratios, the more extreme the apparent bias.
- Values near zero occur when the ratio of the predicted positives (acceptances) to the observed positive outcomes (qualified applicants) for facets $a$ and $d$ have similar values indicating the observed labels for positive outcomes are being predicted with equal precision by the model.
- Negative values occur when the ratio of the predicted positives (acceptances) to the observed positive outcomes (qualified applicants) for facet $d$ is larger than the ratio facet $a$. These values indicate a possible bias against the favored facet $a$ caused by the occurrence of relatively more false positives in facet $a$. The more negative the difference in the ratios, the more extreme the apparent bias.

### Difference in Rejection Rates (DRR)

The difference in rejection rates (DRR) metric is the difference in the ratios of the true negative (TN) predictions to the observed negatives (TN + FN) for facets $a$ and $d$. This metric measures the difference in the precision of the model for predicting rejections from these two facets. Precision measures the fraction of unqualified candidates from the pool of unqualified candidates that are identified as such by
the model. If the model precision for predicting unqualified applicants diverges between the facets, this is a bias and its magnitude is measured by the DRR.

The formula for difference in rejection rates between facets $a$ and $d$:

$$DRR = \frac{TN_d}{TN_d + FN_d} - \frac{TN_a}{TN_a + FN_a}$$

Where:
- $TN_d$ are the true negatives predicted for facet $d$.
- $FN_d$ are the false negatives predicted for facet $d$.
- $TP_a$ are the true negatives predicted for facet $a$.
- $FN_a$ are the false negatives predicted for facet $a$.

For example, suppose the model rejects 100 middle-aged applicants (facet $a$) for a loan (predicted negative labels) of whom 80 are actually unqualified (observed negative labels). Also suppose the model accepts 50 applicants from other age demographics (facet $d$) for a loan (predicted negative labels) of whom only 40 are actually unqualified (observed negative labels). Then $DRR = 40/50 - 80/100 = 0$, so no bias is indicated. When both DAR and DRR are zero, it satisfies a condition known as *equalized odds*.

The range of values for DRR for binary, multicategory facet, and continuous labels is $[-1, +1]$.

- Positive values occur when the ratio of the predicted negatives (rejections) to the observed negative outcomes (unqualified applicants) for facet $d$ is larger than the same ratio for facet $a$. These values indicate a possible bias against the favored facet $a$ caused by the occurrence of relatively more false negatives in facet $a$. The larger the difference in the ratios, the more extreme the apparent bias.
- Values near zero occur when the ratio of the predicted negatives (rejections) to the observed negative outcomes (unqualified applicants) for facets $a$ and $d$ have similar values, indicating the observed labels for negative outcomes are being predicted with equal precision by the model.
- Negative values occur when the ratio of the predicted negatives (rejections) to the observed negative outcomes (unqualified applicants) for facet $a$ is larger than the ratio facet $d$. These values indicate a possible bias against the disfavored facet $d$ caused by the occurrence of relatively more false positives in facet $d$. The more negative the difference in the ratios, the more extreme the apparent bias.

**Accuracy Difference (AD)**

Accuracy difference (AD) metric is the difference between the prediction accuracy for different facets. This metric determines whether the classification by the model is more accurate for one facet than the other. AD indicates whether one facet incurs a greater proportion of Type I and Type II errors. But it cannot differentiate between Type I and Type II errors. For example, the model may have equal accuracy for different age demographics, but the errors may be mostly false positives (Type I errors) for one age-based group and mostly false negatives (Type II errors) for the other.

Also, if loan approvals are made with much higher accuracy for a middle-aged demographic (facet $a$) than for another age-based demographic (facet $d$), either a greater proportion of qualified applicants in the second group are denied a loan (FN) or a greater proportion of unqualified applicants from that group get a loan (FP) or both. This can lead to within group unfairness for the second group, even if the proportion of loans granted is nearly the same for both age-based groups, which is indicated by a DPPL value that is close to zero.

The formula for AD metric is the difference between the prediction accuracy for facet $a$, $ACC_a$, minus that for facet $d$, $ACC_d$:

$$AD = ACC_a - ACC_d$$

Where:
- \( ACC_a = \frac{(TP_a + TN_a)}{(TP_a + TN_a + FP_a + FN_a)} \)
- \( TP_a \) are the true positives predicted for facet \( a \)
- \( TN_a \) are the true negatives predicted for facet \( a \)
- \( FP_a \) are the false positives predicted for facet \( a \)
- \( FN_a \) are the false negatives predicted for facet \( a \)

For example, suppose a model approves loans to 70 applicants from facet \( a \) of 100 and rejected the other 30. 10 should not have been offered the loan (\( FP_a \)) and 60 were approved that should have been (\( TP_a \)). 20 of the rejections should have been approved (\( FN_a \)) and 10 were correctly rejected (\( TN_a \)). The accuracy for facet \( a \) is as follows:

\[ ACC_a = \frac{60 + 10}{60 + 10 + 20 + 10} = 0.7 \]

Next, suppose a model approves loans to 50 applicants from facet \( d \) of 100 and rejected the other 50. 10 should not have been offered the loan (\( FP_d \)) and 40 were approved that should have been (\( TP_d \)). 40 of the rejections should have been approved (\( FN_d \)) and 10 were correctly rejected (\( TN_d \)). The accuracy for facet \( d \) is determined as follows:

\[ ACC_d = \frac{40 + 10}{40 + 10 + 40 + 10} = 0.5 \]

The accuracy difference is thus \( AD = ACC_a - ACC_d = 0.7 - 0.5 = 0.2 \). This indicates there is a bias against facet \( d \) as the metric is positive.

The range of values for \( AD \) for binary and multicategory facet labels is \([-1, +1]\).

- Positive values occur when the prediction accuracy for facet \( a \) is greater than that for facet \( d \). It means that facet \( d \) suffers more from some combination of false positives (Type I errors) or false negatives (Type II errors). This means there is a potential bias against the disfavored facet \( d \).
- Values near zero occur when the prediction accuracy for facet \( a \) is similar to that for facet \( d \).
- Negative values occur when the prediction accuracy for facet \( d \) is greater than that for facet \( a \). It means that facet \( a \) suffers more from some combination of false positives (Type I errors) or false negatives (Type II errors). This means the is a bias against the favored facet \( a \).

**Treatment Equality (TE)**

The treatment equality (TE) is the difference in the ratio of false positives to false negatives between facets \( a \) and \( d \). The main idea of this metric is to assess whether, even if the accuracy across groups is the same, is it the case that errors are more harmful to one group than another? Error rate comes from the total of false positives and false negatives, but the breakdown of these two maybe very different across facets. TE measures whether errors are compensating in the similar or different ways across facets.

The formula for the treatment equality:

\[ TE = \frac{FP_a}{FN_a} - \frac{FP_d}{FN_d} \]

Where:

- \( FP_a \) are the false positives predicted for facet \( a \).
- \( FN_a \) are the false negatives predicted for facet \( a \).
• FP_d are the false positives predicted for facet d.
• FN_d are the false negatives predicted for facet d.

Note the metric becomes unbounded if FN_a or FN_d is zero.

For example, suppose that there are 100 loan applicants from facet a and 50 from facet d. For facet a, 8 were wrongly denied a loan (FN_a) and another 6 were wrongly approved (FP_a). The remaining predictions were true, so TP_a + TN_a = 86. For facet d, 5 were wrongly denied (FN_d) and 2 were wrongly approved (FP_d). The remaining predictions were true, so TP_d + TN_d = 43. The ratio of false positives to false negatives equals 6/8 = 0.75 for facet a and 2/5 = 0.40 for facet d. Hence TE = 0.75 - 0.40 = 0.35, even though both facets have the same accuracy:

\[
\text{ACC}_a = \frac{86}{68 + 8 + 6} = 0.86 \\
\text{ACC}_d = \frac{43}{43 + 5 + 2} = 0.86
\]

The range of values for differences in conditional rejection for binary and multicategory facet labels is (-∞, +∞). The TE metric is not defined for continuous labels. The interpretation of this metric depends on the relative important of false positives (Type I error) and false negatives (Type II error).

• Positive values occur when the ratio of false positives to false negatives for facet a is greater than that for facet d.
• Values near zero occur when the ratio of false positives to false negatives for facet a is similar to that for facet d.
• Negative values occur when the ratio of false positives to false negatives for facet a is less than that for facet d.

**Conditional Demographic Disparity in Predicted Labels (CDDPL)**

The demographic disparity metric (DDPL) determines whether facet d has a larger proportion of the predicted rejected labels than of the predicted accepted labels. It enables a comparison of difference in predicted rejection proportion and predicted acceptance proportion across facets. This metric is exactly the same as the pretraining CDD metric except that it is computed off the predicted labels instead of the observed ones. This metric lies in the range (-1, +1).

The formula for the demographic disparity predictions for labels of facet d is as follows:

\[
\text{DDPL}_d = \frac{n'_d(0)}{n'_d(0) + n'_d(1)} - \frac{n'_d(1)}{n'_d(0) + n'_d(1)} = P_d^R(y'^0) - P_d^A(y'^1)
\]

Where:

• \(n'_d(0)\) = \(n'_a(0) + n'_d(0)\) is the number of predicted rejected labels for facets a and d.
• \(n'_d(1)\) = \(n'_a(1) + n'_d(1)\) is the number of predicted accepted labels for facets a and d.
• \(P_d^R(y'^0)\) is the proportion of predicted rejected labels (value 0) in facet d.
• \(P_d^A(y'^1)\) is the proportion of predicted accepted labels (value 1) in facet d.

A conditional demographic disparity in predicted labels (CDDPL) metric that conditions DDPL on attributes that define a strata of subgroups on the dataset is needed to rule out Simpson's paradox. The regrouping can provide insights into the cause of apparent demographic disparities for less favored facets. The classic case arose in the case of Berkeley admissions where men were accepted at a higher rate overall than women. But when departmental subgroups were examined, women were shown to have higher admission rates than men by department. The explanation was that women had applied to departments with lower acceptance rates than men had. Examining the subgroup acceptance rates revealed that women were actually accepted at a higher rate than men for the departments with lower acceptance rates.
The CDDPL metric gives a single measure for all of the disparities found in the subgroups defined by an attribute of a dataset by averaging them. It is defined as the weighted average of demographic disparities in predicted labels (DDPL\(_i\)) for each of the subgroups, with each subgroup disparity weighted in proportion to the number of observations in contains. The formula for the conditional demographic disparity in predicted labels is as follows:

\[
CDDPL = \frac{1}{n} \sum n_i \cdot DDPL_i
\]

Where:
- \(\sum n_i = n\) is the total number of observations and \(n_i\) is the number of observations for each subgroup.
- \(DDPL_i = n_i^{(0)}/n^{(0)} - n_i^{(1)}/n^{(1)} = P_i^R(y^{(0)}) - P_i^A(y^{(1)})\) is the demographic disparity in predicted labels for the subgroup.

So the demographic disparity for a subgroup in predicted labels (DDPL\(_i\)) are the difference between the proportion of predicted rejected labels and the proportion of predicted accepted labels for each subgroup.

The range of DDPL values for binary, multicategory, and continuous outcomes is [-1, +1].
- +1: when there are no predicted rejection labels for facet \(a\) or subgroup and no predicted acceptances for facet \(d\) or subgroup.
- Positive values indicate there is a demographic disparity in predicted labels as facet \(d\) or subgroup has a larger proportion of the predicted rejected labels than of the predicted accepted labels. The higher the value the greater the disparity.
- Values near zero indicate there is no demographic disparity on average.
- Negative values indicate there is a demographic disparity in predicted labels as facet \(a\) or subgroup has a larger proportion of the predicted rejected labels than of the predicted accepted labels. The lower the value the greater the disparity.
- -1: when there are no predicted rejection labels for facet \(d\) or subgroup and no predicted acceptances for facet \(a\) or subgroup.

**Counterfactual Fliptest (FT)**

The fliptest is an approach that looks at each member of facet \(d\) and assesses whether similar members of facet \(a\) have different model predictions. The members of facet \(a\) are chosen to be k-nearest neighbors of the observation from facet \(d\). We assess how many nearest neighbors of the opposite group receive a different prediction, where the flipped prediction can go from positive to negative and vice versa.

The formula for the counterfactual fliptest is the difference in the cardinality of two sets divided by the number of members of facet \(d\):

\[
FT = (F^+ - F^-)/n_d
\]

Where:
- \(F^+\) is the number of disfavored facet \(d\) members with an unfavorable outcome whose nearest neighbors in favored facet \(a\) received a favorable outcome.
- \(F^-\) is the number of disfavored facet \(d\) members with a favorable outcome whose nearest neighbors in favored facet \(a\) received an unfavorable outcome.
- \(n_d\) is the sample size of facet \(d\).

The range of values for the counterfactual fliptest for binary and multicategory facet labels is [-1, +1]. The FT metric is not defined for continuous labels.
Configure SageMaker Clarify processing jobs

- Positive values occur when the number of unfavorable counterfactual flitptest decisions for the disfavored facet \( d \) exceeds the favorable ones.
- Values near zero occur when the number of unfavorable and favorable counterfactual flitptest decisions balance out.
- Negative values occur when the number of unfavorable counterfactual flitptest decisions for the disfavored facet \( d \) is less than the favorable ones.

Configure an Amazon SageMaker Clarify Processing Jobs for Fairness and Explainability

This topic describes how to configure an Amazon SageMaker Clarify processing job capable of computing bias metrics and feature attributions for explainability. It is implemented using a specialized SageMaker Clarify container image. Instructions are provided for how to locate and download one of these container images. A brief overview of how SageMaker Clarify works is sketched. The parameters needed to configure the processing job and type of analysis are described. Prerequisites are outlined and some advice about compute resources consumed by SageMaker Clarify processing job is provided.

Topics
- Prerequisites (p. 1105)
- Getting Started with a SageMaker Clarify Container (p. 1105)
- How It Works (p. 1107)
- Configure a Processing Job Container's Input and Output Parameters (p. 1107)
- Configure the Analysis (p. 1108)

Prerequisites

Before you begin, you need to meet the following prerequisites:

- You need to provide an input dataset as tabular files in CSV or JSONLines format. The input dataset must include a label column for bias analysis. The dataset should be prepared for machine learning with any pre-processing needed, such as data cleaning or feature engineering, already completed.
- You need to provide a model artifact that supports either the CSV or JSONLines file format as one of its content type inputs. For posttraining bias metrics and explainability, we use the dataset to make inferences with the model artifact. Each row minus the label column must be ready to be used as payload for inferences.
- When creating processing jobs with the SageMaker container image, you need the following:
  - Network isolation must be disabled for the processing job.
  - If the model is in a VPC, the processing job must be in the same VPC as the model.
  - The IAM user/role of the caller must have permissions for SageMaker APIs. We recommend that you use the “arn:aws:iam::aws:policy/AmazonSageMakerFullAccess” managed policy.

Getting Started with a SageMaker Clarify Container

Amazon SageMaker provides prebuilt SageMaker Clarify container images that include the libraries and other dependencies needed to compute bias metrics and feature attributions for explainability. This image has been enabled to run SageMaker Process Data and Evaluate Models (p. 580) in your account.

The image URIs for the containers are in the following form:

\(<ACCOUNT_ID>.dkr.ecr.<REGION_NAME>.amazonaws.com/sagemaker-clarify-processing:1.0\)
For example:

```
205585389593.dkr.ecr.us-east-1.amazonaws.com/sagemaker-clarify-processing:1.0
```

The following table lists the addresses by AWS Region.

### Docker Images for Pretraining Bias Detection

<table>
<thead>
<tr>
<th>Region</th>
<th>Image address</th>
</tr>
</thead>
<tbody>
<tr>
<td>us-east-1</td>
<td>205585389593.dkr.ecr.us-east-1.amazonaws.com/sagemaker-clarify-processing:1.0</td>
</tr>
<tr>
<td>us-east-2</td>
<td>211330385671.dkr.ecr.us-east-2.amazonaws.com/sagemaker-clarify-processing:1.0</td>
</tr>
<tr>
<td>us-west-1</td>
<td>740489534195.dkr.ecr.us-west-1.amazonaws.com/sagemaker-clarify-processing:1.0</td>
</tr>
<tr>
<td>us-west-2</td>
<td>306415355426.dkr.ecr.us-west-2.amazonaws.com/sagemaker-clarify-processing:1.0</td>
</tr>
<tr>
<td>ap-east-1</td>
<td>098760798382.dkr.ecr.ap-east-1.amazonaws.com/sagemaker-clarify-processing:1.0</td>
</tr>
<tr>
<td>ap-south-1</td>
<td>452307495513.dkr.ecr.ap-south-1.amazonaws.com/sagemaker-clarify-processing:1.0</td>
</tr>
<tr>
<td>ap-northeast-2</td>
<td>263625296855.dkr.ecr.ap-northeast-2.amazonaws.com/sagemaker-clarify-processing:1.0</td>
</tr>
<tr>
<td>ap-southeast-1</td>
<td>834264404009.dkr.ecr.ap-southeast-1.amazonaws.com/sagemaker-clarify-processing:1.0</td>
</tr>
<tr>
<td>ap-southeast-2</td>
<td>007051062584.dkr.ecr.ap-southeast-2.amazonaws.com/sagemaker-clarify-processing:1.0</td>
</tr>
<tr>
<td>ap-northeast-1</td>
<td>377024640650.dkr.ecr.ap-northeast-1.amazonaws.com/sagemaker-clarify-processing:1.0</td>
</tr>
<tr>
<td>ca-central-1</td>
<td>675030665977.dkr.ecr.ca-central-1.amazonaws.com/sagemaker-clarify-processing:1.0</td>
</tr>
<tr>
<td>eu-central-1</td>
<td>017069133835.dkr.ecr.eu-central-1.amazonaws.com/sagemaker-clarify-processing:1.0</td>
</tr>
<tr>
<td>eu-west-1</td>
<td>131013547314.dkr.ecr.eu-west-1.amazonaws.com/sagemaker-clarify-processing:1.0</td>
</tr>
<tr>
<td>eu-west-2</td>
<td>440796970383.dkr.ecr.eu-west-2.amazonaws.com/sagemaker-clarify-processing:1.0</td>
</tr>
<tr>
<td>eu-west-3</td>
<td>341593696636.dkr.ecr.eu-west-3.amazonaws.com/sagemaker-clarify-processing:1.0</td>
</tr>
<tr>
<td>eu-north-1</td>
<td>763603941244.dkr.ecr.eu-north-1.amazonaws.com/sagemaker-clarify-processing:1.0</td>
</tr>
<tr>
<td>me-south-1</td>
<td>835444307964.dkr.ecr.me-south-1.amazonaws.com/sagemaker-clarify-processing:1.0</td>
</tr>
</tbody>
</table>
Configure SageMaker Clarify processing jobs

<table>
<thead>
<tr>
<th>Region</th>
<th>Image address</th>
</tr>
</thead>
<tbody>
<tr>
<td>sa-east-1</td>
<td>520018980103.dkr.ecr.sa-east-1.amazonaws.com/sagemaker-clarify-processing:1.0</td>
</tr>
<tr>
<td>af-south-1</td>
<td>811711786498.dkr.ecr.af-south-1.amazonaws.com/sagemaker-clarify-processing:1.0</td>
</tr>
<tr>
<td>eu-south-1</td>
<td>63885417683.dkr.ecr.eu-south-1.amazonaws.com/sagemaker-clarify-processing:1.0</td>
</tr>
</tbody>
</table>

How It Works

A SageMaker processing job used the SageMaker Clarify container at several stages in the lifecycle of the machine learning workflow. You can use the SageMaker Clarify container with your datasets and models to compute the following types of analysis:

- pretraining bias metrics
- posttraining bias metrics
- SHAP values for explainability

You can control which of these analyses are computed when you configure the processing job. For pretraining bias metrics, you need to provide the dataset. You can compute posttraining bias metrics and explainability after your model has been trained by providing the dataset and model name. You must configure the necessary parameters in the form of a JSON configuration file and provide this as an input to the processing job.

After the processing job completes, the result of the analyses is saved in the output location specified in the `ProcessingOutput` parameters of the processing job. You can then download it from there and view the outputs or you can view the results in Studio if you have run a notebook there.

In order to compute posttraining bias metrics and SHAP values, the computation needs to get inferences for the model name provided. To accomplish this, the processing job creates an ephemeral endpoint with the model name, known as a shadow endpoint. The processing job deletes the shadow endpoint after the computations are completed.

At a high level, the processing job completes the following steps:

1. Validate inputs and parameters.
2. Create the shadow endpoint.
3. Compute pretraining bias metrics.
5. Compute local and global feature attributions.
6. Delete shadow endpoint.
7. Generate output files.

Configure a Processing Job Container's Input and Output Parameters

The Processing Job requires that you specify the following input parameters: a dataset files with input name "dataset" as S3 object or prefix, and an analysis configuration file with input name "analysis_config" as an S3 object. The job also requires an output parameter: the output location as an S3 prefix.
You can create and run a processing job with SageMaker CreateProcessingJob API using the AWS SDK or CLI or SageMaker Python SDK.

Using SageMaker Python SDK, create a Processor using the SageMaker Clarify container image URI:

```python
from sagemaker import clarify
clarify_processor = clarify.SageMakerClarifyProcessor(role=role,
instance_count=1,
instance_type='ml.c5.xlarge',
max_runtime_in_seconds=1200,
volume_size_in_gb=100)
```

Then, you must also provide the input and output parameters for the SageMakerClarifyProcessor. If you provide the "dataset_uri", the dataset input is optional.

```python
dataset_path = "s3://my_bucket/my_folder/train.csv"
analysis_config_path = "s3://my_bucket/my_folder/analysis_config.json"
analysis_result_path = "s3://my_bucket/my_folder/output"

analysis_config_input = ProcessingInput(
    input_name="analysis_config",
    source=analysis_config_path,
    destination="/opt/ml/processing/input/config",
    s3_data_type="S3Prefix",
    s3_input_mode="File",
    s3_compression_type="None"
)
dataset_input = ProcessingInput(
    input_name="dataset",
    source=dataset_path,
    destination="/opt/ml/processing/input/data",
    s3_data_type="S3Prefix",
    s3_input_mode="File",
    s3_compression_type="None"
)
analysis_result_output = ProcessingOutput(
    source="/opt/ml/processing/output",
    destination=analysis_result_path,
    output_name="analysis_result",
    s3_upload_mode="EndOfJob"
)
```

### Configure the Analysis

Parameters for the analysis configuration must be provided in a JSON file. The file name or path must be provided in the ProcessingInput parameter named "analysis_config". Examples of analysis configuration files are provided below. In this JSON configuration file, you specify the following parameters:

- "version" – (Optional) Schema version of the configuration file. The latest supported version is used if not provided.
- "dataset_type" – Format of the dataset. Valid values are "text/csv" for CSV and "application/jsonlines" for JSON Lines.
- "dataset_uri" – (Optional) Dataset S3 prefix/object URI (if not given as ProcessingInput). If it is a prefix, then processing job recursively collects all S3 files under the prefix.
- "headers" – (Optional) A list of column names in the dataset. If the dataset_type is "application/jsonlines" and "label" is specified, then the last header shall be the header of label column.
- "label" – (Optional) Target attribute for the model to be used for bias metrics specified as a column name or index if the dataset format is CSV or as JSONPath if the dataset format is JSONLines.
- "probability_threshold" – (Optional) A float value to indicate the threshold to select the binary label in the case of binary classification. The default value is 0.5.
• "features" – (Optional) JSONPath for locating the feature columns for bias metrics if the dataset format is JSONLines.

• "label_values_or_threshold" – (Optional) List of label values or threshold to indicate positive outcome used for bias metrics.

• "facet" – (Optional) A list of features that are sensitive attributes, referred to as facets, to be used for bias metrics in the form of pairs to include the following:
  • "name_or_index" – Facet column name or index.
  • "value_or_threshold" – (Optional) List of values or threshold that the facet column can take which indicates the sensitive group, i.e. the group that is used to measure bias against. If not provided, bias metrics are computed as one group versus all for every unique value. If the facet column is numeric this threshold value is applied as the lower bound to select the sensitive group.

• "group_variable" – (Optional) A column name or index to indicate group variable to be used for the bias metric Conditional Demographic Disparity.

• "methods" – A list of methods and their parameters for the analyses and reports. If any section is omitted then it is not computed.
  • "pre_training_bias" – (Optional) Section on pretraining bias metrics.
    • "methods" – A list of pretraining metrics to be computed.
  • "post_training_bias" – (Optional) Section on posttraining bias metrics.
    • "methods" – A list of posttraining metrics to be computed.

• "shap" – (Optional) Section on SHAP value computation.
  • "baseline" – A list of rows (at least one) or S3 object URI to be used as the baseline dataset (also known as a background dataset) in the Kernel SHAP algorithm. The format should be the same as the dataset format. Each row should contain only the feature columns/values and omit the label column/values
  • "num_samples" – Number of samples to be used in the Kernel SHAP algorithm. This number determines the size of the generated synthetic dataset to compute the SHAP values.
  • "agg_method" – Aggregation method for global SHAP values. Valid values are as follows:
    • "mean_abs" – Mean of absolute SHAP values for all instances.
    • "median" – Median of SHAP values for all instances.
    • "mean_sq" – Mean of squared SHAP values for all instances.
    • "use_logit" – (Optional) Boolean value to indicate if logit function is to be applied to the model predictions. The default value is "false". If "use_logit" is "true" then the SHAP values have log-odds units.
  • "save_local_shap_values" – (Optional) Boolean value to indicate if local SHAP values are to be saved in the output location. Default is "true".

• "predictor" – (Optional) Section on model parameters, required if "shap" and "post_training_bias" sections are present.
  • "model_name" – Model name (as created by CreateModel API with container mode as SingleModel).
  • "instance_type" – Instance type for the shadow endpoint.
  • "initial_instance_count" – Instance count for the shadow endpoint.
  • "content_type" – (Optional) The model input format to be used for getting inferences with the shadow endpoint. Valid values are "text/csv" for CSV and "application/jsonlines". The default value is the same as dataset format.
  • "accept_type" – (Optional) The model output format to be used for getting inferences with the shadow endpoint. Valid values are "text/csv" for CSV and "application/jsonlines". The default value is the same as content_type.

• "label" – (Optional) Index or JSONPath location in the model output for the target attribute to be used bias metrics. In CSV accept_type case, if it is not provided then assume that the model output is a single numeric value corresponding to score or probability.
- "probability" – (Optional) Index or JSONPath location in the model output for probabilities or scores to be used for explainability. For example, if the model output is JSONLines with a list of labels and probabilities, then for bias methods, the label that corresponds to the maximum probability is selected for bias computations. For explainability method, currently all probabilities are explained.

- "label_headers" – (Optional) A list of values that the "label" takes in the dataset that describe which label value each of the scores returned by the model endpoint correspond to. It is used to extract the label value with the highest score as predicted label for bias computations.

- "content_template" – (Optional) A template string to be used to construct the model input from dataset instances. It is only used when "content_type" is "application/jsonlines". The template should have one and only one placeholder, $features, which is replaced by features list at runtime. For example, given "content_template": "{"myfeatures":$features}", if an instance (no label) is 1,2,3 then model input becomes JSONline '{"myfeatures":[1,2,3]}'.

- "report" – (Optional) Section on report parameters. A report is generated if this section is present.

- "name" – (Optional) Filename prefix for the report notebook and PDF file. The default name is "report".

- "title" – (Optional) Title string for the report notebook and PDF file. The default title is "SageMaker Analysis Report".

Example Analysis Configuration JSON File for a CSV Dataset

```
{
  "dataset_type": "text/csv",
  "headers": [
    "feature_0",
    "feature_1",
    "feature_2",
    "feature_3",
    "target"
  ],
  "label": "target",
  "label_values_or_threshold": [1],
  "probability_threshold" : 0.7,
  "facet": [
    {
      "name_or_index" : "feature_1",
      "value_or_threshold": [1]
    },
    {
      "name_or_index" : "feature_2",
      "value_or_threshold": [0.7]
    }
  ],
  "group_variable": "feature_3",
  "methods": {
    "shap": {
      "baseline": [
        ["yes", 3, 0.9, 1]
      ],
      "num_samples": 1000,
      "agg_method": "mean",
      "use_logit": true,
      "save_local_shap_values": true
    }
  }
}
```
"pre_training_bias": {  
    "methods": "all"  
},
"post_training_bias": {  
    "methods": "all"  
},
"report": {  
    "name": "report",  
    "title": "Analysis Report"  
},
"predictor": {  
    "model_name": "my_model",  
    "instance_type": "ml.m5.xlarge",  
    "initial_instance_count": 1,  
    "content_type": "text/csv",  
    "accept_type": "text/csv",  
    "label": 0,  
    "probability": 1  
}

Model output as CSV is as follows:

```
Current,[0.028986845165491104, 0.8253824710845947, 0.028993206098675728, 0.02898673340678215, 0.029557107016444206, 0.0290389321744442, 0.02905467338860035]
```

Corresponding predictor configuration is as follows:

```
"predictor": {  
    ....  
    "accept_type": "text/csv",  
    "label": 0,  
    "probability": 1,  
    ....  
}
```

Example Analysis Configuration JSON File for a JSONLines Dataset

```
{
    "dataset_type": "application/jsonlines",
    "dataset_url": "s3://my_bucket/my_folder/dataset.jsonl",
    "headers": ["Label", "Feature1", "Feature2"],
    "label": "data.label",
    "features": "data.features.values",
    "facet": [
        {  
            "name_or_index": "Feature1",  
            "value_or_threshold": [1,5]  
        },
        {  
            "name_or_index": "Feature2",  
            "value_or_threshold": [2,6]  
        }
    ],
    "methods": {  
        "shap": {  
            "baseline": [
                {  
                    "data": {  
                        "features": {  
                            "values": [9,10],  
                            "label": 0  
                        }  
                    }
                },
                {  
                    "data": {  
                        "features": {  
                            "values": [11,12],  
                            "label": 1  
                        }  
                    }
                }
            ]  
        }
    }
}
```
Dataset as S3 prefix is as follows:

```
"dataset_uri": "s3://my_bucket/my_folder"
```

Dataset as S3 object is as follows:

```
"dataset_uri": "s3://my_bucket/my_folder/train.csv"
```

Baseline as S3 object is as follows:

```
"baseline": "s3://my_bucket/my_folder/baseline.csv"
```

Model output as JSONLines is as follows:

```
{"predicted_label": "Current", "score": 
[0.028986845165491104, 0.8253824710845947, 0.028993206098675728, 0.02898673340678215, 0.029557107016444206, 0.0290389321744442, 0.02905467338860035]"}
```

Corresponding predictor configuration is as follows:

```
"predictor": {
    ...
    "accept_type": "application/jsonlines",
    "label": "predicted_label",
    "probability": "score",
    ...
}
```

Run SageMaker Clarify Processing Jobs for Bias Analysis and Explainability

You use SageMaker Clarify processing jobs to analyze potential sources of bias in your training data and to check your trained model for bias. For the procedure to analyze the data in SageMaker Studio, see Generate Reports for Bias in Pretraining Data in SageMaker Studio (p. 594). The focus here is on posttraining bias metric and SHAP values for explainability. Model predictions can be a source of bias (for example, if they make predictions that more frequently produce a negative result for one group than another). SageMaker Clarify is integrated with SageMaker Experiments so that after a model has been trained, you can identify attributes you would like to check for bias (for example, income). SageMaker runs a set of algorithms to check the trained model and provides you with a visual report on the different types of bias for each attribute, such as whether high-income earners receive more positive predictions compared to low-income earners.

Compute Resources Required for SageMaker Clarify Processing Jobs

Take the following into consideration when determining the compute resources you need to run SageMaker Clarify processing jobs:
• Processing jobs can take several minutes or more to complete.
• Computing explanations can be more time intensive than the actual inference. This includes the time to launch compute resources.
• Computing explanations can be more compute intensive than the actual inference. Review and monitor the charges you may incur from using SageMaker resources. For more information, see Amazon SageMaker Pricing.
• While creating the processing job, you can currently only specify the instance count as 1. Support for running the processing job in parallel, with instance counts greater than 1, will be supported in a future release.

Running the Processing Job

For an example notebook with instructions on how to run a SageMaker Clarify processing job in Studio to detect posttraining model bias, see Explainability and bias detection with Amazon SageMaker Clarify.

If you need instructions on how to open a notebook in Amazon SageMaker Studio, see Create or Open an Amazon SageMaker Studio Notebook (p. 87). The following code samples are taken from the example notebook listed previously.

After you have trained your model, instantiate the SageMaker Clarify processor using the following command:

```python
from sagemaker import clarify
clarify_processor = clarify.SageMakerClarifyProcessor(role=role,
            instance_count=1,
            instance_type='ml.c4.xlarge',
            sagemaker_session=session)
```

Next, configure the input dataset, where to store the output, the label column targeted with a `DataConfig` object, specify information about your trained model with `ModelConfig`, and provide information about the formats of your predictions with `ModelPredictedLabelConfig`.

```python
bias_report_output_path = 's3://{}/{}/clarify-bias'.format(bucket, prefix)
bias_data_config = clarify.DataConfig(s3_data_input_path=train_uri,
                        s3_output_path=bias_report_output_path,
                        label='Target',
                        headers=training_data.columns.to_list(),
                        dataset_type='text/csv')

model_config = clarify.ModelConfig(model_name=model_name,
                        instance_type='ml.c5.xlarge',
                        instance_count=1,
                        accept_type='text/csv')

predictions_config = clarify.ModelPredictedLabelConfig(probability_threshold=0.8)

bias_config = clarify.BiasConfig(label_values_or_threshold=[1],
                        facet_name='Sex',
                        facet_values_or_threshold=[0])
```

Use `BiasConfig` to provide information on which columns contain the facets (sensitive groups, `Sex`), what the sensitive features (`facet_values_or_threshold`) might be, and what the desirable outcomes are (`label_values_or_threshold`).

```python
bias_config = clarify.BiasConfig(label_values_or_threshold=[1],
                        facet_name='Sex',
                        facet_values_or_threshold=[0])
```

You can run both the pretraining and posttraining analysis in the processing job at the same time with `run_bias()`.
clarify_processor.run_bias(data_config=bias_data_config,  
bias_config=bias_config,  
model_config=model_config,  
model_predicted_label_config=predictions_config,  
pre_training_methods='all',  
post_training_methods='all')

View the results in Studio or download them from the bias_report_output_path S3 bucket.

Get the Analysis Results

After the processing job is finished, you can download the output files to inspect or visualize the results in Studio. The output directory contains the following files:

• analysis.json – Bias metrics and SHAP values in JSON format.
• report.ipynb – Static notebook to visualize the bias metrics and SHAP values.
• explanations_shap/out.csv – Local (per-instance) SHAP values for each row in the dataset in the same format as input dataset. On each row, the output file contains SHAP values for each feature and the predicted label.

In the analysis JSON file, the bias metrics and SHAP values are organized into three separate sections.

```json
{
    "explanations": { . . . }
    "pre_training_bias_metrics": { . . . }
    "post_training_bias_metrics": { . . . }
}
```

SHAP values are in the “explanations” section. Values correspond to the global SHAP value for each feature column.

```json
"explanations": {
    "kernel_shap": {
        "label0": {
            "global_shap_values": {
                "feature_0": 0.022486410860333206,  
                "feature_1": 0.007381025261958729,  
                "feature_2": 0.006843906804137847
            }
        }
    }
}
```

The bias metrics are in the pretraining and posttraining bias metrics sections.

```json
{
    "post_training_bias_metrics": {
        "label": "target",  
        "label_value_or_threshold": "1",  
        "facets": {
            "feature_2": [
                {  
                    "value_or_threshold": "1",  
                    "metrics": [
                        {  
                            "name": "DI",  
                            "description": "Disparate Impact (DI)",  
                            "value": 0.711340206185567
                        }
                    ]
                }
            ]
        }
    }
}
```
Model Explainability

Amazon SageMaker Clarify provides tools to help explain how machine learning (ML) models make predictions. These tools can help ML modelers and developers and other internal stakeholders understand model characteristics as a whole prior to deployment and to debug predictions provided by the model after it’s deployed. Transparency about how ML models arrive at their predictions is also critical to consumers and regulators who need to trust the model predictions if they are going to accept the decisions based on them. SageMaker Clarify uses a model-agnostic feature attribution approach, which you can used to understand why a model made a prediction after training and to provide per-instance explanation during inference. The implementation includes a scalable and efficient implementation of SHAP, based on the concept of a Shapley value from the field of cooperative game theory that assigns each feature an importance value for a particular prediction.

For more information on bias metrics and SHAP values and how to interpret them, see the Amazon AI Fairness and Explainability Whitepaper.

A bar chart of top SHAP values and tables with the bias metrics are provided in the report notebook.

Troubleshooting

If the processing job fails to finish, you can try the following:

- Inspect the job logs directly in the notebook where you ran the job in. The job logs are located in the output of the notebook cell where you initiated the run.
- Inspect the job logs in CloudWatch.
- Add the following line in your notebook to describe the last processing job and look for the failure reason and exit message:
  ```python
clarify_processor.jobs[-1].describe()
```
- Execute the following AWS CLI command to describe the processing job and look for the failure reason and exit message:
  ```bash
  aws sagemaker describe-processing-job --processing-job-name <processing-job-id>
  ```

If the processing job finishes but no results are found and a warning message is found in the CloudWatch logs that says “Signal 15 received, cleaning up”, this is an indication that the job was stopped either due to customer request using StopProcessingJob API call or the job ran out of allotted time. In this case, check the maximum runtime in the job configuration (max_runtime_in_seconds) and increase as needed.

The failure reason and exit message are intended to contain descriptive messages and exceptions, if encountered, during the run. One common reason is invalid or missing parameters. If you encounter unclear, confusing, or misleading messages or are unable to find a solution, submit feedback.
What is the function of an explanation in the machine learning context? An explanation can be thought of as the answer to a *Why question* that helps humans understand the cause of a prediction. In the context of an ML model, you might be interested in answering questions such as:

- "Why did the model predict a negative outcome such as a loan rejection for a given applicant?"
- "How does the model make predictions?"
- "Why did the model make an incorrect prediction?"
- "Which features have the largest influence on the behavior of the model?"

You can use explanations for auditing and meeting regulatory requirements, building trust in the model and supporting human decision-making, and debugging and improving model performance.

The need to satisfy the demands for human understanding about the nature and outcomes of ML inference is key to the sort of explanation needed. Research from philosophy and cognitive science disciplines has shown that people care especially about contrastive explanations, or explanations of why an event X happened instead of some other event Y that did not occur. Here, X could be an unexpected or surprising event that happened and Y corresponds to an expectation based on their existing mental model referred to as a *baseline*. Note that for the same event X, different people might seek different explanations depending on their point of view or mental model Y. In the context of explainable AI, you can think of X as the example being explained and Y as a baseline that is typically chosen to represent an uninformative or average example in the dataset. Sometimes, the baseline might be implicit as with an image with all pixels of the same color in the case of ML models for images.

**Topics**

- Feature Attributions that Use Shapley Values (p. 1116)
- SHAP Baselines for Explainability (p. 1117)
- Create Feature Attribute Baselines and Explainability Reports (p. 1117)

**Feature Attributions that Use Shapley Values**

SageMaker Clarify provides feature attributions based on the concept of *Shapley value*. You can use Shapley values to determine the contribution that each feature made to model predictions. These attributions can be provided for specific predictions and at a global level for the model as a whole. For example, if you used an ML model for college admissions, the explanations could help determine whether the GPA or the SAT score was the feature most responsible for the model's predictions, and then you can determine how responsible each feature was for determining an admission decision about a particular student.

SageMaker Clarify has taken the concept of Shapley values from game theory and deployed it in a machine learning context. The Shapley value provides a way to quantify the contribution of each player to a game, and hence the means to distribute the total gain generated by a game to its players based on their contributions. In this machine learning context, SageMaker Clarify treats the prediction of the model on a given instance as the *game* and the features included in the model as the *players*. For a first approximation, you might be tempted to determine the marginal contribution or effect of each feature by quantifying the result of either dropping that feature from the model or dropping all other features from the model. However, this approach does not take into account that features included in a model are often not independent from each other. For example, if two features are highly correlated, dropping either one of the features might not alter the model prediction significantly.

To address these potential dependencies, the Shapley value requires that the outcome of each possible combination (or coalition) of features must be considered to determine the importance of each feature. Given *d* features, there are $2^d$ such possible feature combinations, each corresponding to a potential model. To determine the attribution for a given feature *f*, consider the marginal contribution of including *f* in all feature combinations (and associated models) that do not contain *f*, and take the average. It can be shown that Shapley value is the unique way of assigning the contribution or importance of
each feature that satisfies certain desirable properties. In particular, the sum of Shapley values of each feature corresponds to the difference between the predictions of the model and a dummy model with no features. However, even for reasonable values of $d$, say 50 features, it is computationally prohibitive and impractical to train $2^d$ possible models. As a result, SageMaker Clarify needs to make use of various approximation techniques. For this purpose, SageMaker Clarify uses SHapley Additive exPlanations (SHAP), which incorporates such approximations and devised a scalable and efficient implementation of the Kernel SHAP algorithm through additional optimizations.

For additional information on Shapley values, see A Unified Approach to Interpreting Model Predictions.

**SHAP Baselines for Explainability**

As noted earlier, explanations are typically contrastive (that is, they account for deviations from a baseline). As a result, for the same model prediction, you can expect to get different explanations with respect to different baselines so your choice of a baseline is crucial. In an ML context, the baseline corresponds to a hypothetical instance that can be either uninformative or informative. During the computation of Shapley values, SageMaker Clarify generates several new instances between the baseline and the given instance, in which the absence of a feature is modeled by setting the feature value to that of the baseline and the presence of a feature is modeled by setting the feature value to that of the given instance. Thus, the absence of all features corresponds to the baseline and the presence of all features corresponds to the given instance.

How can you choose good baselines? Often it is desirable to select a baseline with very low information content. For example, you can construct an average instance from the training dataset by taking either the median or average for numerical features and the mode for categorical features. For the college admissions example, you might be interested in explaining why a particular applicant was accepted as compared to a baseline acceptances based on an average applicant.

Alternatively, you can choose to generate explanations with respect to informative baselines. For the college admissions scenario, you might want to explain why a particular applicant was rejected when compared with other applicants from similar demographic backgrounds. In this case, you can choose a baseline that represents the applicants of interest, namely those from a similar demographic background. Thus, you can use informative baselines to concentrate the analysis on the specific aspects of a particular model prediction. You can isolate the features for assessment by setting demographic attributes and other features that you can’t act on to the same value as in the given instance.

**Create Feature Attribute Baselines and Explainability Reports**

For an example notebook with instructions on how to run a SageMaker Clarify processing job in Studio that creates explanations for its predictions relative to a baseline, see Explainability and bias detection with Amazon SageMaker Clarify.

If you need instructions on how to open a notebook in Amazon SageMaker Studio, see Create or Open an Amazon SageMaker Studio Notebook (p. 87). The following code examples are taken from the example notebook listed previously. This section discusses the code related to the use of Shapley values to provide reports that compare the relative contributions each feature made the predictions.

Use `SHAPConfig` to create the baseline. In this example, the `mean_abs` is the mean of absolute SHAP values for all instances, specified as the baseline. You use `DataConfig` to configure the target variable, data input and output paths, and their formats.

```python
code
shap_config = clarify.SHAPConfig(baseline=test_features.iloc[0].values.tolist(),
                                      num_samples=15,
                                      agg_method='mean_abs')
```

1117
incremental_training = 's3://{}//{}clarity/incremental_training'.format(bucket, prefix)

incremental_data_config = clarify.DataConfig(s3_input_path=train_uri,
                                          s3_output_path=incremental_training,
                                          label='Target',
                                          headers=training_data.columns.to_list(),
                                          dataset_type='text/csv')

Then run the incremental training job.

clarify_processor.run_incremental(data_config=incremental_data_config,
                                  model_config=model_config,
                                  incremental_config=incremental_training)

View the results in Studio or download them from the incremental_training S3 bucket.

Incremental Training in Amazon SageMaker

Over time, you might find that a model generates inference that are not as good as they were in the past. With incremental training, you can use the artifacts from an existing model and use an expanded dataset to train a new model. Incremental training saves both time and resources.

Use incremental training to:

- Train a new model using an expanded dataset that contains an underlying pattern that was not accounted for in the previous training and which resulted in poor model performance.
- Use the model artifacts or a portion of the model artifacts from a popular publicly available model in a training job. You don't need to train a new model from scratch.
- Resume a training job that was stopped.
- Train several variants of a model, either with different hyperparameter settings or using different datasets.

For more information about training jobs, see Train a Model with Amazon SageMaker (p. 8).

You can train incrementally using the SageMaker console or the Amazon SageMaker Python SDK.

Important
Only three built-in algorithms currently support incremental training: Object Detection Algorithm (p. 770), Image Classification Algorithm (p. 690), and Semantic Segmentation Algorithm (p. 792).

Topics
- Perform Incremental Training (Console) (p. 1118)
- Perform Incremental Training (API) (p. 1120)

Perform Incremental Training (Console)

To complete this procedure, you need:

- The Amazon Simple Storage Service (Amazon S3) bucket URI where you've stored the training data.
- The S3 bucket URI where you want to store the output of the job.
- The Amazon Elastic Container Registry path where the training code is stored. For more information, see Docker Registry Paths for SageMaker Built-in Algorithms (p. 640).
- The URL of the S3 bucket where you've stored the model artifacts that you want to use in incremental training. To find the URL for the model artifacts, see the details page of the training job used to create
the model. To find the details page, in the SageMaker console, choose Inference, choose Models, and then choose the model.

To restart a stopped training job, use the URL to the model artifacts that are stored in the details page as you would with a model or a completed training job.

To perform incremental training (console)

1. Open the Amazon SageMaker console at https://console.aws.amazon.com/sagemaker/.
2. In the navigation pane, choose Training, then choose Training jobs.
3. Choose Create training job.
4. Provide a name for the training job. The name must be unique within an AWS Region in an AWS account. The training job name must have 1 to 63 characters. Valid characters: a-z, A-Z, 0-9, and . : + = @ _ % - (hyphen).
5. Choose the algorithm that you want to use. For information about algorithms, see Use Amazon SageMaker Built-in Algorithms (p. 635).
6. (Optional) For Resource configuration, either leave the default values or increase the resource consumption to reduce computation time.
   a. (Optional) For Instance type, choose the ML compute instance type that you want to use. In most cases, ml.m4.xlarge is sufficient.
   b. For Instance count, use the default, 1.
   c. (Optional) For Additional volume per instance (GB), choose the size of the ML storage volume that you want to provision. In most cases, you can use the default, 1. If you are using a large dataset, use a larger size.
7. Provide information about the input data for the training dataset.
   a. For Channel name, either leave the default (train) or enter a more meaningful name for the training dataset, such as expanded-training-dataset.
   b. For InputMode, choose File. For incremental training, you need to use file input mode.
   c. For S3 data distribution type, choose FullyReplicated. This causes each ML compute instance to use a full replicate of the expanded dataset when training incrementally.
   d. If the expanded dataset is uncompressed, set the Compression type to None. If the expanded dataset is compressed using Gzip, set it to Gzip.
   e. (Optional) If you are using File input mode, leave Content type empty. For Pipe input mode, specify the appropriate MIME type. Content type is the multipurpose internet mail extension (MIME) type of the data.
   f. For Record wrapper, if the dataset is saved in RecordIO format, choose RecordIO. If your dataset is not saved as a RecordIO formatted file, choose None.
   g. For S3 data type, if the dataset is stored as a single file, choose S3Prefix. If the dataset is stored as several files in a folder, choose Manifest.
   h. For S3 location, provide the URL to the path where you stored the expanded dataset.
   i. Choose Done.
8. To use model artifacts in a training job, you need to add a new channel and provide the needed information about the model artifacts.
   a. For Input data configuration, choose Add channel.
   b. For Channel name, enter model to identify this channel as the source of the model artifacts.
   c. For InputMode, choose File. Model artifacts are stored as files.
   d. For S3 data distribution type, choose FullyReplicated. This indicates that each ML compute instance should use all of the model artifacts for training.
   e. For Compression type, choose None because we are using a model for the channel.
Perform Incremental Training (API)

This example shows how to use SageMaker APIs to train a model using the SageMaker image classification algorithm and the Caltech 256 Image Dataset, then train a new model using the first one. It uses Amazon S3 for input and output sources. Please see the incremental training sample notebook for more details on using incremental training.

Note
In this example we used the original datasets in the incremental training, however you can use different datasets, such as ones that contain newly added samples. Upload the new datasets to S3 and make adjustments to the data_channels variable used to train the new model.

Get an AWS Identity and Access Management (IAM) role that grants required permissions and initialize environment variables:

```python
import sagemaker
from sagemaker import get_execution_role

role = get_execution_role()
print(role)

dess = sagemaker.Session()

bucket=dess.default_bucket()
print(bucket)

prefix = 'ic-incr-training'
```
Get the training image for the image classification algorithm:

```python
from sagemaker.amazon.amazon_estimator import get_image_uri

training_image = get_image_uri(sess.boto_region_name, 'image-classification',
repo_version="latest")
#Display the training image
print (training_image)
```

Download the training and validation datasets, then upload them to Amazon Simple Storage Service (Amazon S3):

```python
import os
import urllib.request
import boto3

# Define a download function
def download(url):
    filename = url.split('/')[-1]
    if not os.path.exists(filename):
        urllib.request.urlretrieve(url, filename)

# Download the caltech-256 training and validation datasets
download('http://data.mxnet.io/data/caltech-256/caltech-256-60-train.rec')
download('http://data.mxnet.io/data/caltech-256/caltech-256-60-val.rec')

# Create four channels: train, validation, train_lst, and validation_lst
s3train = 's3://{}/train/'.format(bucket, prefix)
s3validation = 's3://{}/validation/'.format(bucket, prefix)

# Upload the first files to the train and validation channels
!aws s3 cp caltech-256-60-train.rec $s3train --quiet
!aws s3 cp caltech-256-60-val.rec $s3validation --quiet
```

Define the training hyperparameters:

```python
# Define hyperparameters for the estimator
hyperparams = { "num_layers": "18",
    "resize": "32",
    "num_training_samples": "50000",
    "num_classes": "10",
    "image_shape": "3,28,28",
    "mini_batch_size": "128",
    "epochs": "3",
    "learning_rate": "0.1",
    "lr_scheduler_step": "2,3",
    "lr_scheduler_factor": "0.1",
    "augmentation_type": "crop_color",
    "optimizer": "sgd",
    "momentum": "0.9",
    "weight_decay": "0.0001",
    "beta_1": "0.9",
    "beta_2": "0.999",
    "gamma": "0.9",
    "eps": "1e-8",
    "top_k": "5",
    "checkpoint_frequency": "1",
    "use_pretrained_model": "0",
    "model_prefix": "" }
```

Create an estimator object and train the first model using the training and validation datasets:
# Fit the base estimator
s3_output_location = 's3://{}/{}/output'.format(bucket, prefix)
ic = sagemaker.estimator.Estimator(training_image,
    role,
    instance_count=1,
    instance_type='ml.p2.xlarge',
    volume_size=50,
    max_run=360000,
    input_mode='File',
    output_path=s3_output_location,
    sagemaker_session=sess,
    hyperparameters=hyperparams)

train_data = sagemaker.inputs.TrainingInput(s3train, distribution='FullyReplicated',
    content_type='application/x-recordio',
    s3_data_type='S3Prefix')
validation_data = sagemaker.inputs.TrainingInput(s3validation,
    distribution='FullyReplicated',
    content_type='application/x-recordio',
    s3_data_type='S3Prefix')
data_channels = {'train': train_data, 'validation': validation_data}
ic.fit(inputs=data_channels, logs=True)

To use the model to incrementally train another model, create a new estimator object and use the model artifacts (ic.model_data, in this example) for the model_uri input argument:

# Given the base estimator, create a new one for incremental training
incr_ic = sagemaker.estimator.Estimator(training_image,
    role,
    instance_count=1,
    instance_type='ml.p2.xlarge',
    volume_size=50,
    max_run=360000,
    input_mode='File',
    output_path=s3_output_location,
    sagemaker_session=sess,
    hyperparameters=hyperparams,
    model_uri=ic.model_data) # This parameter will ingest the previous job's model as a new channel
incr_ic.fit(inputs=data_channels, logs=True)

After the training job has completed, the newly trained model artifacts are stored under the S3 output path that you provided in Output_path. To deploy the model to get predictions, see Step 6: Deploy the Model to Amazon SageMaker (p. 68).

### Managed Spot Training in Amazon SageMaker

Amazon SageMaker makes it easy to train machine learning models using managed Amazon EC2 Spot instances. Managed spot training can optimize the cost of training models up to 90% over on-demand instances. SageMaker manages the Spot interruptions on your behalf.

Managed Spot Training uses Amazon EC2 Spot instance to run training jobs instead of on-demand instances. You can specify which training jobs use spot instances and a stopping condition that specifies how long SageMaker waits for a job to run using Amazon EC2 Spot instances. Metrics and logs generated during training runs are available in CloudWatch.

Spot instances can be interrupted, causing jobs to take longer to start or finish. You can configure your managed spot training job to use checkpoints. SageMaker copies checkpoint data from a local path to
Amazon S3. When the job is restarted, SageMaker copies the data from Amazon S3 back into the local path. The training can then resume from the last checkpoint instead of restarting. For more information about checkpointing, see Use Checkpoints in Amazon SageMaker (p. 1123).

**Note**
Unless your training job will complete quickly, we recommend you use checkpointing with managed spot training. SageMaker built-in algorithms and marketplace algorithms that do not checkpoint are currently limited to a MaxWaitTimeInSeconds of 3600 seconds (60 minutes).

**Using Managed Spot Training**

To use managed spot training, create a training job. Set EnableManagedSpotTraining to True and specify the MaxWaitTimeInSeconds. MaxWaitTimeInSeconds must be larger than MaxRuntimeInSeconds. For more information about creating a training job, see DescribeTrainingJob.

You can calculate the savings from using managed spot training using the formula

\[
(1 - \frac{BillableTimeInSeconds}{TrainingTimeInSeconds}) \times 100
\]

For example, if BillableTimeInSeconds is 100 and TrainingTimeInSeconds is 500, the savings is 80%.

**Managed Spot Training Lifecycle**

You can monitor a training job using TrainingJobStatus and SecondaryStatus returned by DescribeTrainingJob. The list below shows how TrainingJobStatus and SecondaryStatus values change depending on the training scenario:

- **Spot instances acquired with no interruption during training**
  1. InProgress: Starting -> Downloading -> Training -> Uploading

- **Spot instances interrupted once. Later, enough spot instances were acquired to finish the training job.**
  1. InProgress: Starting -> Downloading -> Training -> Interrupted -> Starting -> Downloading -> Training -> Uploading

- **Spot instances interrupted twice and MaxWaitTimeInSeconds exceeded.**
  1. InProgress: Starting -> Downloading -> Training -> Interrupted -> Starting -> Downloading -> Training -> Interrupted -> Downloading -> Training
  2. Stopping: Stopping
  3. Stopped: MaxWaitTimeExceeded

- **Spot instances were never launched.**
  1. InProgress: Starting
  2. Stopping: Stopping
  3. Stopped: MaxWaitTimeExceeded

**Use Checkpoints in Amazon SageMaker**

A checkpoint is a snapshot of the state of the model. They can be used with Managed Spot Training. If a training job is interrupted, a snapshot can be used to resume from a previously saved point. This can save training time.
Snapshots are saved to an Amazon S3 location you specify. You can configure the local path to use for snapshots or use the default. When a training job is interrupted, Amazon SageMaker copies the training data to Amazon S3. When the training job is restarted, the checkpoint data is copied to the local path. It can be used to resume at the checkpoint.

Built-in algorithms that support checkpointing are Semantic Segmentation, Image Classification and Object Detection.

To enable checkpoints, provide an Amazon S3 location. You can optionally provide a local path and choose to use a shared folder. The default local path is `/opt/ml/checkpoints/`. For more information, see CreateTrainingJob.

Provide Dataset Metadata to Training Jobs with an Augmented Manifest File

To include metadata with your dataset in a training job, use an augmented manifest file. When using an augmented manifest file, your dataset must be stored in Amazon Simple Storage Service (Amazon S3) and you must configure your training job to use dataset stored there. You specify the location and format of this dataset for one or more Channel. Augmented manifests can only support Pipe input mode. See the section, InputMode in Channel. Augmented manifests can only support Pipe input mode.

When specifying a channel's parameters, you specify a path to the file, called a S3Uri. Amazon SageMaker interprets this URI based on the specified S3DataType in S3DataSource. The AugmentedManifestFile option defines a manifest format that includes metadata with the input data. Using an augmented manifest file is an alternative to preprocessing when you have labeled data. For training jobs using labeled data, you typically need to preprocess the dataset to combine input data with metadata before training. If your training dataset is large, preprocessing can be time consuming and expensive.

Augmented Manifest File Format

An augmented manifest file must be formatted in JSON Lines format. In JSON Lines format, each line in the file is a complete JSON object followed by a newline separator.

During training, SageMaker parses each JSON line and sends some or all of its attributes on to the training algorithm. You specify which attribute contents to pass and the order in which to pass them with the AttributeNames parameter of the CreateTrainingJob API. The AttributeNames parameter is an ordered list of attribute names that SageMaker looks for in the JSON object to use as training input.

For example, if you list `"line", "book"` for AttributeNames, the input data must include the attribute names of line and book in the specified order. For this example, the following augmented manifest file content is valid:

```
{"author": "Herman Melville", "line": "Call me Ishmael", "book": "Moby Dick"}
{"line": "It was love at first sight.", "author": "Joseph Heller", "book": "Catch-22"}
```

SageMaker ignores unlisted attribute names even if they precede, follow, or are in between listed attributes.

When using augmented manifest files, observe the following guidelines:

- The order of the attributes listed in the AttributeNames parameter determines the order of the attributes passed to the algorithm in the training job.
- The listed AttributeNames can be a subset of all of the attributes in the JSON line. SageMaker ignores unlisted attributes in the file.
• You can specify any type of data allowed by the JSON format in AttributeNames, including text, numerical, data arrays, or objects.

• To include an S3 URI as an attribute name, add the suffix -ref to it.

If an attribute name contains the suffix -ref, the attribute's value must be an S3 URI to a data file that is accessible to the training job. For example, if AttributeNames contains ["image-ref", "is-a-cat"], a valid augmented manifest file might contain these lines:

```
{"image-ref": "s3://mybucket/sample01/image1.jpg", "is-a-cat": 1}
{"image-ref": "s3://mybucket/sample02/image2.jpg", "is-a-cat": 0}
```

For the first line of this manifest, SageMaker retrieves the contents of the S3 object s3://mybucket/foo/image1.jpg and streams it to the algorithm for training. The second line is the string representation of the is-a-cat attribute "1", which is followed by the contents of the second line.

To create an augmented manifest file, use Amazon SageMaker Ground Truth to create a labeling job. For more information, see Output Data (p. 381).

**Stream Augmented Manifest File Data**

Augmented manifest format enables you to do training in Pipe mode using files without needing to create RecordIO files. You need to specify both train and validation channels as values for the InputDataConfig parameter of the CreateTrainingJob request. Augmented manifest files are supported only for channels using Pipe input mode. For each channel, the data is extracted from its augmented manifest file and streamed (in order) to the algorithm through the channel's named pipe. Pipe mode uses the first in first out (FIFO) method, so records are processed in the order in which they are queued. For information about Pipe input mode, see Input Mode.

Attribute names with a "-ref" suffix point to preformatted binary data. In some cases, the algorithm knows how to parse the data. In other cases, you might need to wrap the data so that records are delimited for the algorithm. If the algorithm is compatible with RecordIO-formatted data, specifying RecordIO for RecordWrapperType solves this issue. If the algorithm is not compatible with RecordIO format, specify None for RecordWrapperType and make sure that your data is parsed correctly for your algorithm.

Using the ["image-ref", "is-a-cat"] example, if you use RecordIO wrapping, the following stream of data is sent to the queue:

```
recordio_formatted(s3://mybucket/foo/image1.jpg)recordio_formatted("1")recordio_formatted(s3://mybucket/bar/image2.jpg)recordio_formatted("0")
```

Images that are not wrapped with RecordIO format, are streamed with the corresponding is-a-cat attribute value as one record. This can cause a problem because the algorithm might not delimit the images and attributes correctly. For more information about using augmented manifest files for image classification, see Train with Augmented Manifest Image Format.

With augmented manifest files and Pipe mode in general, size limits of the EBS volume do not apply. This includes settings that otherwise must be within the EBS volume size limit such as S3DataDistributionType. For more information about Pipe mode and how to use it, see Using Your Own Training Algorithms - Input Data Configuration.

**Use an Augmented Manifest File (Console)**

To complete this procedure, you need:

• The URL of the S3 bucket where you've stored the augmented manifest file.
To store the data that is listed in the augmented manifest file in an S3 bucket.
- The URL of the S3 bucket where you want to store the output of the job.

**To use an augmented manifest file in a training job (console)**

1. Open the Amazon SageMaker console at https://console.aws.amazon.com/sagemaker/.
2. In the navigation pane, choose **Training**, then choose **Training jobs**.
3. Choose **Create training job**.
4. Provide a name for the training job. The name must be unique within an AWS Region in an AWS account. It can have 1 to 63 characters. Valid characters: a-z, A-Z, 0-9, and . : + = @ _ % - (hyphen).
5. Choose the algorithm that you want to use. For information about supported built-in algorithms, see **Use Amazon SageMaker Built-in Algorithms**. If you want to use a custom algorithm, make sure that it is compatible with Pipe mode.
6. (Optional) For **Resource configuration**, either accept the default values or, to reduce computation time, increase the resource consumption.
   a. (Optional) For **Instance type**, choose the ML compute instance type that you want to use. In most cases, **ml.m4.xlarge** is sufficient.
   b. For **Instance count**, use the default, 1.
   c. (Optional) For **Additional volume per instance (GB)**, choose the size of the ML storage volume that you want to provision. In most cases, you can use the default, 1. If you are using a large dataset, use a larger size.
7. Provide information about the input data for the training dataset.
   a. For **Channel name**, either accept the default (**train**) or enter a more meaningful name, such as **training-augmented-manifest-file**.
   b. For **InputMode**, choose **Pipe**.
   c. For **S3 data distribution type**, choose **FullyReplicated**. When training incrementally, fully replicating causes each ML compute instance to use a complete copy of the expanded dataset. For neural-based algorithms, such as **Neural Topic Model (NTM) Algorithm** (p. 748), choose **ShardedByS3Key**.
   d. If the data specified in the augmented manifest file is uncompressed, set the **Compression type** to **None**. If the data is compressed using gzip, set it to **Gzip**.
   e. (Optional) For **Content type**, specify the appropriate MIME type. Content type is the multipurpose internet mail extension (MIME) type of the data.
   f. For **Record wrapper**, if the dataset specified in the augmented manifest file is saved in RecordIO format, choose **RecordIO**. If your dataset is not saved as a RecordIO-formatted file, choose **None**.
   g. For **S3 data type**, choose **AugmentedManifestFile**.
   h. For **S3 location**, provide the path to the bucket where you stored the augmented manifest file.
   i. For **AugmentedManifestFile attribute names**, specify the name of an attribute that you want to use. The attribute name must be present within the augmented manifest file, and is case-sensitive.
   j. (Optional) To add more attribute names, choose **Add row** and specify another attribute name for each attribute.
   k. (Optional) To adjust the order of attribute names, choose the up or down buttons next to the names. When using an augmented manifest file, the order of the specified attribute names is important.
   l. Choose **Done**.
8. For **Output data configuration**, provide the following information:
   a. For **S3 location**, type the path to the S3 bucket where you want to store the output data.
b. (Optional) You can use your AWS Key Management Service (AWS KMS) encryption key to encrypt the output data at rest. For Encryption key, provide the key ID or its Amazon Resource Number (ARN). For more information, see KMS-Managed Encryption Keys.

9. (Optional) For Tags, add one or more tags to the training job. A tag is metadata that you can define and assign to AWS resources. In this case, you can use tags to help you manage your training jobs. A tag consists of a key and a value, which you define. For example, you might want to create a tag with Project as a key and a value that refers to a project that is related to the training job, such as Home value forecasts.

10. Choose Create training job. SageMaker creates and runs the training job.

After the training job has finished, SageMaker stores the model artifacts in the bucket whose path you provided for S3 output path in the Output data configuration field. To deploy the model to get predictions, see Step 6: Deploy the Model to Amazon SageMaker (p. 68).

Use an Augmented Manifest File (API)

The following shows how to train a model with an augmented manifest file using the SageMaker high-level Python library:

```python
# Create a model object set to using "Pipe" mode.
model = sagemaker.estimator.Estimator(training_image,
    role,
    train_instance_count=1,
    train_instance_type='ml.p3.2xlarge',
    train_volume_size = 50,
    train_max_run = 360000,
    input_mode = 'Pipe',
    output_path=s3_output_location,
    sagemaker_session=session)

# Create a train data channel with S3_data_type as 'AugmentedManifestFile' and attribute names.
train_data = sagemaker.session.s3_input(your_augmented_manifest_file,
    distribution='FullyReplicated',
    content_type='application/x-recordio',
    s3_data_type='AugmentedManifestFile',
    attribute_names=['source-ref', 'annotations'],
    input_mode='Pipe',
    record_wrapping='RecordIO')

data_channels = {'train': train_data}

# Train a model.
model.fit(inputs=data_channels, logs=True)
```

After the training job has finished, SageMaker stores the model artifacts in the bucket whose path you provided for S3 output path in the Output data configuration field. To deploy the model to get predictions, see Step 6: Deploy the Model to Amazon SageMaker (p. 68).

Monitor and Analyze Training Jobs Using Metrics

An Amazon SageMaker training job is an iterative process that teaches a model to make predictions by presenting examples from a training dataset. Typically, a training algorithm computes several metrics, such as training error and prediction accuracy. These metrics help diagnose whether the model is learning well and will generalize well for making predictions on unseen data. The training algorithm writes the values of these metrics to logs, which SageMaker monitors and sends to Amazon CloudWatch in real time. To analyze the performance of your training job, you can view graphs of these metrics.
in CloudWatch. When a training job has completed, you can also get a list of the metric values that it computes in its final iteration by calling the `DescribeTrainingJob` operation.

**Topics**
- Training Metrics Sample Notebooks (p. 1128)
- Defining Training Metrics (p. 1128)
- Monitoring Training Job Metrics (Console) (p. 1130)
- Monitoring Training Job Metrics (SageMaker Console) (p. 1130)
- Example: Viewing a Training and Validation Curve (p. 1133)

### Training Metrics Sample Notebooks

The following sample notebooks show how to view and plot training metrics:

- An Introduction to the Amazon SageMaker ObjectToVec Model for Sequence-to-sequence Embedding (`object2vec_sentence_similarity.ipynb`)
- Regression with the Amazon SageMaker XGBoost Algorithm (`xgboost_abalone.ipynb`)

For instructions how to create and access Jupyter notebook instances that you can use to run the examples in SageMaker, see Example Notebooks (p. 134). To see a list of all the SageMaker samples, after creating and opening a notebook instance, choose the SageMaker Examples tab. To access the example notebooks that show how to use training metrics, `object2vec_sentence_similarity.ipynb` and `xgboost_abalone.ipynb`, from the Introduction to Amazon algorithms section. To open a notebook, choose its Use tab, then choose Create copy.

### Defining Training Metrics

SageMaker automatically parses the logs for metrics that built-in algorithms emit and sends those metrics to CloudWatch. If you want SageMaker to parse logs from a custom algorithm and send metrics that the algorithm emits to CloudWatch, you have to specify the metrics that you want SageMaker to send to CloudWatch when you configure the training job. You specify the name of the metrics that you want to send and the regular expressions that SageMaker uses to parse the logs that your algorithm emits to find those metrics.

You can specify the metrics that you want to track with the SageMaker console, the SageMaker Python SDK (https://github.com/aws/sagemaker-python-sdk), or the low-level SageMaker API.

**Topics**
- Defining Regular Expressions for Metrics (p. 1128)
- Defining Training Metrics (Low-level SageMaker API) (p. 1129)
- Defining Training Metrics (SageMaker Python SDK) (p. 1129)
- Define Training Metrics (Console) (p. 1130)

### Defining Regular Expressions for Metrics

To find a metric, SageMaker searches the logs that your algorithm emits and finds logs that match the regular expression that you specify for that metric. If you are using your own algorithm, do the following:

- Make sure that the algorithm writes the metrics that you want to capture to logs
- Define a regular expression that accurately searches the logs to capture the values of the metrics that you want to send to CloudWatch metrics.
For example, suppose your algorithm emits metrics for training error and validation error by writing logs similar to the following to stdout or stderr:

```
Train_error=0.138318;  Valid_error = 0.324557;
```

If you want to monitor both of those metrics in CloudWatch, your AlgorithmSpecification would look like the following:

```
"AlgorithmSpecification": {
  "TrainingImage": ContainerName,
  "TrainingInputMode": "File",
  "MetricDefinitions" : [
    {
      "Name": "train:error",
      "Regex": "Train_error=(.*?);",
    },
    {
      "Name": "validation:error",
      "Regex": "Valid_error=(.*?);"
    }
  ]
}
```

In the regex for the train:error metric defined above, the first part of the regex finds the exact text "Train_error=", and the expression (.*)?; captures zero or more of any character until the first semicolon character. In this expression, the parenthesis tell the regex to capture what is inside them, . means any character, * means zero or more, and ? means capture only until the first instance of the ; character.

**Defining Training Metrics (Low-level SageMaker API)**

Define the metrics that you want to send to CloudWatch by specifying a list of metric names and regular expressions in the MetricDefinitions field of the AlgorithmSpecification input parameter that you pass to the CreateTrainingJob operation. For example, if you want to monitor both the train:error and validation:error metrics in CloudWatch, your AlgorithmSpecification would look like the following:

```
"AlgorithmSpecification": {
  "TrainingImage": ContainerName,
  "TrainingInputMode": "File",
  "MetricDefinitions" : [
    {
      "Name": "train:error",
      "Regex": "Train_error=(.*?);",
    },
    {
      "Name": "validation:error",
      "Regex": "Valid_error=(.*?);"
    }
  ]
}
```

For more information about defining and running a training job by using the low-level SageMaker API, see Create and Run a Training Job (AWS SDK for Python (Boto3)) (p. 66).

**Defining Training Metrics (SageMaker Python SDK)**

Define the metrics that you want to send to CloudWatch by specifying a list of metric names and regular expressions as the metric_definitions argument when you initialize an Estimator object.
For example, if you want to monitor both the **train:error** and **validation:error** metrics in CloudWatch, your **Estimator** initialization would look like the following:

```python
estimator = Estimator(image_name=ImageName,
                      role='SageMakerRole',
                      train_instance_count=1,
                      train_instance_type='ml.c4.xlarge',
                      k=10,
                      sagemaker_session=sagemaker_session,
                      metric_definitions=[
                          {'Name': 'train:error', 'Regex': 'Train_error=(.*?);'},
                          {'Name': 'validation:error', 'Regex': 'Valid_error=(.*?);'}
                      ])
```

For more information about training by using Amazon SageMaker Python SDK estimators, see https://github.com/aws/sagemaker-python-sdk#sagemaker-python-sdk-overview.

## Define Training Metrics (Console)

You can define metrics for a custom algorithm in the console when you create a training job by providing the name and regular expression (regex) for **Metrics**.

For example, if you want to monitor both the **train:error** and **validation:error** metrics in CloudWatch, your metric definitions would look like the following:

```json
[
    {
        "Name": "train:error",
        "Regex": "Train_error=(.*?);"
    },
    {
        "Name": "validation:error",
        "Regex": "Valid_error=(.*?);"
    }
]
```

## Monitoring Training Job Metrics (Console)

You can monitor the metrics that a training job emits in real time in the CloudWatch console.

**To monitor training job metrics (CloudWatch console)**

2. Choose **Metrics**, then choose /aws/sagemaker/TrainingJobs.
3. Choose **TrainingJobName**.
4. On the **All metrics** tab, choose the names of the training metrics that you want to monitor.
5. On the **Graphed metrics** tab, configure the graph options. For more information about using CloudWatch graphs, see Graph Metrics in the Amazon CloudWatch User Guide.

## Monitoring Training Job Metrics (SageMaker Console)

You can monitor the metrics that a training job emits in real time by using the SageMaker console.
To monitor training job metrics (SageMaker console)

2. Choose Training jobs, then choose the training job whose metrics you want to see.
3. Choose TrainingJobName.
4. In the Monitor section, you can review the graphs of instance utilization and algorithm metrics.
Monitor

Access logs for debugging and progress reporting. View metrics to set alarms, send notifications, or take actions. Learn more

- MemoryUtilization
- CPUUtilization
- DiskUtilization
- GPUUtilization
- GPMemoryUtilization
- Trainthroughput
- Trainaccuracy
- Validationaccuracy
- Validationcross_entropy
- Traincross_entropy
Example: Viewing a Training and Validation Curve

Typically, you split the data that you train your model on into training and validation datasets. You use the training set to train the model parameters that are used to make predictions on the training dataset. Then you test how well the model makes predictions by calculating predictions for the validation set. To analyze the performance of a training job, you commonly plot a training curve against a validation curve.

Viewing a graph that shows the accuracy for both the training and validation sets over time can help you to improve the performance of your model. For example, if training accuracy continues to increase over time, but, at some point, validation accuracy starts to decrease, you are likely overfitting your model. To address this, you can make adjustments to your model, such as increasing regularization.

For this example, you can use the `Image-classification-full-training` example that is in the Example notebooks section of your SageMaker notebook instance. If you don't have a SageMaker notebook instance, create one by following the instructions at Step 2: Create an Amazon SageMaker Notebook Instance (p. 60). If you prefer, you can follow along with the End-to-End Multiclass Image Classification Example in the example notebook on GitHub. You also need an Amazon S3 bucket to store the training data and for the model output. If you haven't created a bucket to use with SageMaker, create one by following the instructions at Step 1: Create an Amazon S3 Bucket (p. 59).

To view training and validation error curves

2. Choose Notebooks, and then choose Notebook instances.
3. Choose the notebook instance that you want to use, and then choose Open.
4. On the dashboard for your notebook instance, choose SageMaker Examples.
5. Expand the Introduction to Amazon Algorithms section, and then choose Use next to Image-classification-full-training.ipynb.
7. In the first code cell of the notebook, replace `<bucket-name>` with the name of your S3 bucket.
8. Run all of the cells in the notebook up to the Deploy section. You don't need to deploy an endpoint or get inference for this example.
10. Choose Metrics, then choose `/aws/sagemaker/TrainingJobs`.
11. Choose TrainingJobName.
12. On the All metrics tab, choose the train:accuracy and validation:accuracy metrics for the training job that you created in the notebook.
13. On the graph, choose an area that the metric's values to zoom in. You should see something like the following:
Example: Viewing a Training and Validation Curve
Deploy Models for Inference

After you build and train your models, you can deploy them to get predictions in one of two ways:

• To set up a persistent endpoint to get predictions from your models, use Amazon SageMaker hosting services. For an overview on deploying a single model or multiple models with SageMaker hosting services, see Deploy a Model on SageMaker Hosting Services (p. 11).

For an example of how to deploy a model to the SageMaker hosting service, see Step 6.1: Deploy the Model to SageMaker Hosting Services (p. 68).

Or, if you prefer, watch the following video tutorial:

Deploy Your ML Models to Production at Scale with Amazon SageMaker

• To get predictions for an entire dataset, use SageMaker batch transform. For an overview on deploying a model with SageMaker batch transform, see Get Inferences for an Entire Dataset with Batch Transform (p. 14).

For an example of how to deploy a model with batch transform, see Step 6.2: Deploy the Model with Batch Transform (p. 70).

Or, if you prefer, watch the following video tutorial:

Deploy Your ML Models to Production at Scale with Amazon SageMaker

Prerequisites

These topics assume that you have built and trained one or more machine learning models and are ready to deploy them. If you are new to SageMaker and have not completed these prerequisite tasks, work through the steps in the Get Started with Amazon SageMaker (p. 33) tutorial to familiarize yourself with an example of how SageMaker manages the data science process and how it handles model deployment. For more information about training a model, see Train Models (p. 630).

What do you want to do?

SageMaker provides features to manage resources and optimize inference performance when deploying machine learning models. For guidance on using inference pipelines, compiling and deploying models with Neo, Elastic Inference, and automatic model scaling, see the following topics.

• To manage data processing and real-time predictions or to process batch transforms in a pipeline, see Deploy an Inference Pipeline (p. 1220).

• If you want to deploy a model on inf1 instances, see Compile and Deploy Models with Neo (p. 1234).

• To train TensorFlow, Apache MXNet, PyTorch, ONNX, and XGBoost models once and optimize them to deploy on ARM, Intel, and Nvidia processors, see Compile and Deploy Models with Neo (p. 1234).

• To preprocess entire datasets quickly or to get inferences from a trained model for large datasets when you don't need a persistent endpoint, see Use Batch Transform (p. 1323).

• To speed up the throughput and decrease the latency of getting real-time inferences from your deep learning models that are deployed as SageMaker hosted models using a GPU instance for your endpoint, see Use Amazon SageMaker Elastic Inference (EI) (p. 1312).

• To dynamically adjust the number of instances provisioned in response to changes in your workload, see Automatically Scale Amazon SageMaker Models (p. 1330).
• To create an endpoint that can host multiple models using a shared serving container, see Host Multiple Models with Multi-Model Endpoints (p. 1160).
• To test multiple models in production, see Test models in production (p. 1346).

Manage Model Deployments

For guidance on managing model deployments, including monitoring, troubleshooting, and best practices, and for information on storage associated with inference hosting instances:

• For tools that can be used to monitor model deployments, see Monitor Amazon SageMaker (p. 1704).
• For troubleshooting model deployments, see Troubleshoot Amazon SageMaker Model Deployments (p. 1354).
• For model deployment best practices, see Deployment Best Practices (p. 1355).
• For information about the size of storage volumes provided for different sizes of hosting instances, see Host Instance Storage Volumes (p. 1355).

Deploy Your Own Inference Code

For developers that need more advanced guidance on how to run your own inference code:

• To run your own inference code hosting services, see Use Your Own Inference Code with Hosting Services (p. 1387).
• To run your own inference code for batch transforms, see Use Your Own Inference Code with Batch Transform (p. 1391).

Guide to SageMaker

What Is Amazon SageMaker? (p. 1)

Register and Deploy Models with Model Registry

With the SageMaker model registry you can do the following:

• Catalog models for production.
• Manage model versions.
• Associate metadata, such as training metrics, with a model.
• Manage the approval status of a model.
• Deploy models to production.
• Automate model deployment with CI/CD.

Catalog models by creating model package groups that contain different versions of a model. You can create a model group that tracks all of the models that you train to solve a particular problem. You can then register each model you train and the model registry adds it to the model group as a new model version. A typical workflow might look like the following:

• Create a model group.
• Create an ML pipeline that trains a model. For information about SageMaker pipelines, see Create and Manage SageMaker Pipelines (p. 1410).
For each run of the ML pipeline, create a model version that you register in the model group you created in the first step.

The following topics show how to use the model registry.

Topics
- Create a Model Group (p. 1137)
- Register a Model Version (p. 1140)
- View Model Groups and Versions (p. 1141)
- View the Details of a Model Version (p. 1143)
- Update the Approval Status of a Model (p. 1148)
- Deploy a Model in the Registry (p. 1153)
- Deploy a Model Version from a Different Account (p. 1154)
- View the Deployment History of a Model (p. 1156)

Create a Model Group

A model group contains a group of versioned models. Create a model group by using either the AWS SDK for Python (Boto3) or in SageMaker Studio.

Create a Model Package Group (Boto3)

To create a model group by using Boto3, call the create_model_package_group method, and specify a name and description as parameters. The following example shows how to create a model group. The response from the create_model_package_group call is the Amazon Resource Name (ARN) of the new model package group.

First, import the required packages and set up the SageMaker Boto3 client.

```python
import time
import os
from sagemaker import get_execution_role, session
import boto3

region = boto3.Session().region_name
role = get_execution_role()
sm_client = boto3.client('sagemaker', region_name=region)

model_package_group_name = "scikit-iris-detector-" + str(round(time.time()))
model_package_group_input_dict = {
    "ModelPackageGroupName" : model_package_group_name,
    "ModelPackageGroupDescription" : "Sample model package group"
}

create_model_package_group_response = sm_client.create_model_package_group(**model_package_group_input_dict)
print('ModelPackageGroup Arn : {}'.format(create_model_package_group_response['ModelPackageGroupArn']))
```
Create a Model Package Group (SageMaker Studio)

To create a model group in SageMaker Studio, complete the following steps.

1. Sign in to Studio. For more information, see Onboard to Amazon SageMaker Studio (p. 34).
2. In the left navigation pane, choose the Components and registries icon (🔧).
3. Choose Model registry.
4. Choose Create model group.
5. In the **Create new model group** dialog box, enter the following information:

   - For **Name**, enter the name of the new model group.
   - (Optional) For **Description**, enter a description for the model group.
   - (Optional) For **Tags**, enter any key-value pairs you want to associate with the model group. For information about using tags, see Tagging AWS resources in the *AWS General Reference*.
   - (Optional) For **Project**, choose a project with which to associate the model group. For information about projects, see Automate MLOps with SageMaker Projects (p. 1439).

6. Choose **Create model group**.
Register a Model Version

You register a model by creating a model version that specifies the model group to which it belongs. A model version must include both the model artifacts (the trained weights of a model), and the inference code for the model. Create a model version by using either the AWS SDK for Python (Boto3) or by creating a step in a SageMaker model building pipeline.

Register a Model Version (SageMaker Pipelines)

To register a model version by using a SageMaker model building pipeline, create a RegisterModel step in your pipeline. For information about creating RegisterModel step as part of a pipeline, see Define a Pipeline (p. 1411).

Register a Model Version (Boto3)

To register a model version by using Boto3, call the create_model_package method.

First, you set up the parameter dictionary to pass to the create_model_package method.
```python
modelpackage_inference_specification = {
    "InferenceSpecification": {
        "Containers": [
            {
                "Image": '257758044811.dkr.ecr.us-east-2.amazonaws.com/sagemaker-xgboost:1.2-1',
            },
            "SupportedContentTypes": [ "text/csv" ],
            "SupportedResponseMIMETypes": [ "text/csv" ],
        }
    },

    # Specify the model data
    modelpackage_inference_specification["InferenceSpecification"]['Containers'][0]['ModelDataUrl']=model_url

    create_model_package_input_dict = {
        "ModelPackageGroupName" : model_package_group_name,
        "ModelPackageDescription" : "Model to detect 3 different types of irises (Setosa, Versicolour, and Virginica)",
        "ModelApprovalStatus" : "PendingManualApproval"
    }

    create_model_package_input_dict.update(modelpackage_inference_specification)

    create_model_package_response = sm_client.create_model_package(**create_model_package_input_dict)

    model_package_arn = create_model_package_response["ModelPackageArn"]

    print('ModelPackage Version ARN : {}'.format(model_package_arn))

View Model Groups and Versions

Model groups and versions help you organize your models. You can view a list of the model versions in a model group.

View a List of Model Versions in a Group

You can view all of the model versions that are associated with a model group. If a model group represents all models that you train to address a specific ML problem, you can view all of those related models.

View a List of Model Versions in a Group (Boto3)

To view model versions associated with a model group by using Boto3, call the list_model_packages method, and pass the name of the model group as the value of the ModelPackageGroupName parameter. The following code lists the model versions associated with the model group you created in Create a Model Package Group (Boto3) (p. 1137).

```
1. Sign in to Studio. For more information, see Onboard to Amazon SageMaker Studio (p. 34).
2. In the left navigation pane, choose the **Components and registries** icon (🔍).
3. Choose **Model registry**.

4. From the model groups list, choose the model group you want to view.
5. A new tab appears with a list of the model versions in the model group, as shown following.
View the Details of a Model Version

You can view details of a specific model version by using either the AWS SDK for Python (Boto3) or by using SageMaker Studio.

View the Details of a Model Version (Boto3)

To view the details of a model version by using Boto3, complete the following steps.

1. Call the `list_model_packages` method to view the model versions in a model group.

   ```python
   sm_client.list_model_packages(ModelPackageName="ModelGroup1")
   ```

   The response is a list of model package summaries. You can get the Amazon Resource Name (ARN) of the model versions from this list.
2. Call `describe_model_package` to see the details of the model version. You pass in the ARN of a model version that you got in the output of the call to `list_model_packages`.

```python
sm_client.describe_model_package(ModelPackageName="arn:aws:sagemaker:us-east-2:123456789012:model-package/ModelGroup1/1")
```

The output of this call is a JSON object with the model version details.

```json
{
    "ModelPackageGroupName": "ModelGroup1",
    "ModelPackageVersion": 1,
    "ModelPackageDescription": "Test Model",
    "CreationTime": datetime.datetime(2020, 10, 29, 1, 27, 46, 46000, tzinfo=tzlocal()),
    "InferenceSpecification": {
        "Containers": [{
            "Image": "257758044811.dkr.ecr.us-east-2.amazonaws.com/sagemaker-xgboost:1.0-1-cpu-py3",
            "ImageDigest": "sha256:99fa602c0f19aee33297a5926f8497ca7bcd2a391b7d6000300204eef803bca66",
            "ModelDataUrl": "s3://sagemaker-us-east-2:123456789012/ModelGroup1/pipelines-0gdonccek7o9-AbaloneTrain-stmiylhtIR/output/model.tar.gz"}
        ],
        "SupportedTransformInstanceTypes": ["ml.m5.xlarge"],
        "SupportedTransformInstanceTypes": ["ml.t2.medium", "ml.m5.xlarge"],
        "SupportedContentTypes": ["text/csv"],
        "SupportedResponseMIMETypes": ["text/csv"],
        "ModelPackageStatus": "Completed",
        "ModelPackageStatusDetails": {
            "ValidationStatuses": [],
            "ImageScanStatuses": [],
            "CertifyForMarketplace": False,
            "ModelApprovalStatus": "PendingManualApproval",
            "LastModifiedTime": datetime.datetime(2020, 10, 29, 1, 28, 0, 438000, tzinfo=tzlocal()),
            "ResponseMetadata": {
                "RequestId": "12345678-abcd-1234-abcd-aabbcdddeefef",
                "HTTPStatus": 200,
                "HTTPHeaders": {
                    "x-amzn-requestid": "12345678-abcd-1234-abcd-aabbcdddeefef",
                    "content-type": "application/x-amz-json-1.1",
                    "date": "Mon, 23 Nov 2020 04:57:18 GMT"}
            },
            "RetryAttempts": 0}
        }
    }
}
```

**View the Details of a Model Version (SageMaker Studio)**

To view the details of a model version in SageMaker Studio, complete the following steps.
1. Sign in to Studio. For more information, see Onboard to Amazon SageMaker Studio (p. 34).
2. In the left navigation pane, choose the Components and registries icon (🔗).
3. Choose Model registry.
4. From the model groups list, choose the model group you want to view.
5. A new tab appears with a list of the model versions in the model group.
6. In the list of model versions, double-click the model version for which you want to view details.
7. On the model version tab that opens, choose one of the following to see details about the model version:

- **Activity** - Shows events for the model version, such as approval status updates.
- **Metrics** - Shows quality metrics for the model. For metrics to appear, you must enable data capture for your model by using &SM; Model Monitor. For information about capturing data, see Capture Data (p. 1195).
- **Settings** - Shows information such as the project that the model version is associated with, the pipeline that generated the model, the model group, and the model's location in Amazon S3.
Update the Approval Status of a Model

After you create a model version, you typically want to evaluate its performance before you deploy it to a production endpoint. If it performs to your requirements, you can update the approval status of the model version to Approved. If it does not perform to your requirements, you update the approval status to Rejected.

Setting the approval status of a model version to Approved can trigger CI/CD deployment for the model. Similarly, you can block promotion of a model by setting its approval status to Rejected. You can manually set the status approval of a model version after you register it, or you can create a conditional step to evaluate the model when you create a SageMaker pipeline. For information about creating a conditional step in a SageMaker pipeline, see Create and Manage SageMaker Pipelines (p. 1410).

**Note**

SageMaker does not support changing the approval status from Approved to Rejected.

You can update the approval status of a model version by using the AWS SDK for Python (Boto3) or by using SageMaker Studio. You can also update the approval status of a model version as part of a conditional step in a SageMaker pipeline. For information about using a model approval step in a SageMaker pipeline, see SageMaker Pipelines Overview (p. 1398).

**Update the Approval Status of a Model (Boto3)**

When you created the model version in Register a Model Version (p. 1140), you set the `ModelApprovalStatus` to PendingManualApproval. You must update the approval status for the model to Approved by calling `update_model_package`. Note that you can also automate this process by writing code that, for example, sets the approval status of a model depending on the result of an evaluation of some measure of the model's performance. You can also create a step in a pipeline that
automatically deploys a new model version when it is approved. The following code snippet shows how to manually change the approval status to Approved.

```python
model_package_update_input_dict = {
    "ModelPackageArn" : model_package_arn,
    "ModelApprovalStatus" : "Approved"
}
model_package_update_response = sm_client.update_model_package(**model_package_update_input_dict)
```

### Update the Approval Status of a Model (SageMaker Studio)

To update the approval status of a model version in SageMaker Studio, complete the following steps.

1. Sign in to Studio. For more information, see [Onboard to Amazon SageMaker Studio](p. 34).
2. In the left navigation pane, choose the **Components and registries** icon.
3. Choose **Model registry**.
4. From the model groups list, choose the model group you want to view.
5. A new tab appears with a list of the model versions in the model group.
6. In the list of model versions, double-click the model version for which you want to update the approval status.
7. On the model version tab that opens, choose **Update status**.
8. In the **Update model version status** dialog box, for **Status** choose either **Approved** or **Rejected**, and then choose **Update status**.
Deploy a Model in the Registry

After you register a model version and approve it for deployment, deploy it to a SageMaker endpoint for real-time inference.

When you create a MLOps project and choose an MLOps project template that includes model deployment, approved model versions in the model registry are automatically deployed to production. For information about using SageMaker MLOps projects, see Automate MLOps with SageMaker Projects (p. 1439).

Deploy a Model in the Registry (Boto3)

To deploy a model version, complete the following steps:

1. Create a model object from the model version by calling create_model, passing the Amazon Resource Name (ARN) of the model version as the primary container for the model object. The following code snippet assumes you have already created the SageMaker Boto3 client sm_client,
and that you have already created a model version with an ARN that you have stored in a variable named `model_version_arn`.

```python
def model_name = 'DEMO-modelregistry-model-' + strftime("%Y-%m-%d-%H-%M-%S", gmtime())
print("Model name : ".format(model_name))
primary_container = {
  'ModelPackageName': model_version_arn,
}
create_model_respose = sm_client.create_model(
  ModelName = model_name,
  ExecutionRoleArn = role,
  PrimaryContainer = primary_container
)
print("Model arn : ".format(create_model_respose['ModelArn']))
```

2. Create an endpoint configuration by calling `create_endpoint_config`. The endpoint configuration specifies the number and type of Amazon EC2 instances to use for the endpoint.

```python
def endpoint_config_name = 'DEMO-modelregistry-EndpointConfig-' + strftime("%Y-%m-%d-%H-%M-%S", gmtime())
print(endpoint_config_name)
create_endpoint_config_response = sm_client.create_endpoint_config(
  EndpointConfigName = endpoint_config_name,
  ProductionVariants=[[
    'InstanceType':'ml.m4.xlarge',
    'InitialVariantWeight':1,
    'InitialInstanceCount':1,
    'ModelName':model_name,
    'VariantName':'AllTraffic']])
```

3. Create the endpoint by calling `create_endpoint`.

```python
def endpoint_name = 'DEMO-modelregistry-endpoint-' + strftime("%Y-%m-%d-%H-%M-%S", gmtime())
print("EndpointName={}".format(endpoint_name))
create_endpoint_response = sm_client.create_endpoint(
  EndpointName=endpoint_name,
  EndpointConfigName=endpoint_config_name)
print(create_endpoint_response['EndpointArn'])
```

**Deploy a Model Version from a Different Account**

Enable an AWS account to deploy model versions that were created in a different account by adding a cross-account resource policy. For example, one team in your organization might be responsible for training models, and a different team is responsible for deploying and updating models. When you create resource policies, you apply the policy to the resource to which you want to grant access. For more information about cross-account resource policies in AWS, see [Cross-account policy evaluation logic](https://docs.aws.amazon.com/IdentityAndAccessManagement/latest/UserGuide/cross-account-policy-evaluation-logic.html) in the *AWS Identity and Access Management User Guide*.

To enable cross-account model deployment in SageMaker, you have to provide a cross-account resource policy for the model group that contains the model versions you want to deploy, the Amazon ECR repository where the inference image for the model group resides, and the Amazon S3 bucket where the model versions are stored. The following example creates cross-account policies for all three of these resources, and applies the policies to the resources.

```python
import json
```
# cross account id to grant access to
cross_account_id = "123456789012"

# 1. Create policy for access to the ECR repository
ecr_repository_policy = {
    'Version': '2012-10-17',
    'Statement': [{
        'Sid': 'AddPerm',
        'Effect': 'Allow',
        'Principal': {
            'AWS': f'arn:aws:iam::{cross_account_id}:root'
        },
        'Action': ['ecr:*']
    }]
}
# Convert the ECR policy from JSON dict to string
ecr_repository_policy = json.dumps(ecr_repository_policy)
# Set the new ECR policy
ecr = boto3.client('ecr')
response = ecr.set_repository_policy(
    registryId = account,
    repositoryName = 'decision-trees-sample',
    policyText = ecr_repository_policy
)

# 2. Create policy for access to the S3 bucket
bucket_policy = {
    'Version': '2012-10-17',
    'Statement': [{
        'Sid': 'AddPerm',
        'Effect': 'Allow',
        'Principal': {
            'AWS': f'arn:aws:iam::{cross_account_id}:root'
        },
        'Action': 's3:*',
        'Resource': f'arn:aws:s3:::{bucket}/*' 
    }]
}
# Convert the policy from JSON dict to string
bucket_policy = json.dumps(bucket_policy)
# Set the new policy
s3 = boto3.client('s3')
response = s3.put_bucket_policy(
    Bucket = bucket,
    Policy = bucket_policy
)

# 3. Create policy for access to the ModelPackageGroup
model_package_group_policy = {
    'Version': '2012-10-17',
    'Statement': [{
        'Sid': 'AddPermModelPackageGroup',
        'Effect': 'Allow',
        'Principal': {
            'AWS': f'arn:aws:iam::{cross_account_id}:root'
        },
        'Action': ['sagemaker:DescribeModelPackageGroup'],
        'Resource': f'arn:aws:sagemaker:{region}:{account}:model-package-group/{model_package_group_name}'
    },
    {'Sid': 'AddPermModelPackageVersion',
        'Effect': 'Allow',
        'Principal': {
            'AWS': f'arn:aws:iam::{cross_account_id}:root'
        },
        'Action': ['sagemaker:DescribeModelPackageVersion'],
        'Resource': f'arn:aws:sagemaker:{region}:{account}:model-package-group/{model_package_group_name}/model-package-version/*'
    }
}
'AWS': f'arn:aws:iam::{cross_account_id}:root'

    'Action': ['sagemaker:DescribeModelPackage",
               "sagemaker:ListModelPackages",
               "sagemaker:UpdateModelPackage",
               "sagemaker:CreateModel"],
    'Resource': f'arn:aws:sagemaker:{region}:{account}:model-package/{model_package_group_name}/*'

"
]

# Convert the policy from JSON dict to string
model_package_group_policy = json.dumps(model_package_group_policy)

# Set the new policy
response = sm_client.put_model_package_group_policy(
    ModelPackageGroupName = model_package_group_name,
    ResourcePolicy = model_package_group_policy)

print('ModelPackageGroupArn :
    {').format(create_model_package_group_response['ModelPackageGroupArn']))
print("First Versioned ModelPackageArn: " + model_package_arn)
print("Second Versioned ModelPackageArn: " + model_package_arn2)
print("Success! You are all set to proceed for cross account deployment.")

The example assumes that you previously defined the following variables:

- account - The account of the authenticated caller.
- bucket - The S3 bucket where the model versions are stored.
- model_package_group_name - The model group to which you want to grant access.

To be able to deploy a model that was created in a different account, the user must have a role that has
access to SageMaker actions, such as a role with the AmazonSageMakerFullAccess managed policy.
For information about SageMaker managed policies, see AWS Managed (Predefined) Policies for Amazon
SageMaker (p. 1663).

**View the Deployment History of a Model**

View the deployments for a model version SageMaker Studio by opening the tab for that model version.

**View the deployment history for a model version**

1. Sign in to Studio. For more information, see Onboard to Amazon SageMaker Studio (p. 34).
2. In the left navigation pane, choose the Components and registries icon (ظرف).
3. Choose Model registry.
4. From the model groups list, choose the model group you want to view.
5. A new tab appears with a list of the model versions in the model group.
6. In the list of model versions, double-click the model version for which you want to view details.
7. On the model version tab that opens, choose **Activity**. Deployments for the model version appear as events in the activity list with an **Event type** of **ModelDeployment**.
Host Multiple Models with Multi-Model Endpoints

To create an endpoint that can host multiple models, use multi-model endpoints. Multi-model endpoints provide a scalable and cost-effective solution to deploying large numbers of models. They use a shared serving container that is enabled to host multiple models. This reduces hosting costs by improving endpoint utilization compared with using single-model endpoints. It also reduces deployment overhead because Amazon SageMaker manages loading models in memory and scaling them based on the traffic patterns to them.

Multi-model endpoints also enable time-sharing of memory resources across your models. This works best when the models are fairly similar in size and invocation latency. When this is the case, multi-model endpoints can effectively use instances across all models. If you have models that have significantly higher transactions per second (TPS) or latency requirements, we recommend hosting them on dedicated endpoints. Multi-model endpoints are also well suited to scenarios that can tolerate occasional cold-start-related latency penalties that occur when invoking infrequently used models.
Multi-model endpoints support A/B testing. They work with Auto Scaling and AWS PrivateLink. You can use multi-model-enabled containers with serial inference pipelines, but only one multi-model-enabled container can be included in an inference pipeline. You can’t use multi-model-enabled containers with Amazon Elastic Inference.

You can use the AWS SDK for Python (Boto) or the SageMaker console to create a multi-model endpoint. You can use multi-model endpoints with custom-built containers by integrating the Multi Model Server library.

**Topics**
- Supported Algorithms and Frameworks (p. 1161)
- Sample Notebooks for Multi-Model Endpoints (p. 1161)
- Instance Recommendations for Multi-Model Endpoint Deployments (p. 1162)
- Create a Multi-Model Endpoint (p. 1163)
- Invoke a Multi-Model Endpoint (p. 1166)
- Add or Remove Models (p. 1167)
- Build Your Own Container with Multi Model Server (p. 1167)
- How Multi-Model Endpoints Work (p. 1172)
- Multi-Model Endpoint Security (p. 1173)
- CloudWatch Metrics for Multi-Model Endpoint Deployments (p. 1173)

**Supported Algorithms and Frameworks**

The inference containers for the following algorithms and frameworks support multi-model endpoints:
- XGBoost Algorithm (p. 813)
- K-Nearest Neighbors (k-NN) Algorithm (p. 719)
- Linear Learner Algorithm (p. 732)
- Use Scikit-learn with Amazon SageMaker (p. 30)
- Use Apache MXNet with Amazon SageMaker (p. 17)
- Use PyTorch with Amazon SageMaker (p. 27)

To use any other framework or algorithm, use the SageMaker inference toolkit to build a container that supports multi-model endpoints. For information, see Build Your Own Container with Multi Model Server (p. 1167).

**Sample Notebooks for Multi-Model Endpoints**

For a sample notebook that uses SageMaker to deploy multiple XGBoost models to an endpoint, see the Multi-Model Endpoint XGBoost Sample Notebook. For a sample notebook that shows how to set up and deploy a custom container that supports multi-model endpoints in SageMaker, see the Multi-Model Endpoint BYOC Sample Notebook. For instructions how to create and access Jupyter notebook instances that you can use to run the example in SageMaker, see Use Amazon SageMaker Notebook Instances (p. 124). After you've created a notebook instance and opened it, choose the SageMaker Examples tab to see a list of all the SageMaker samples. The Multi-Model Endpoint notebook is located in the ADVANCED FUNCTIONALITY section. To open a notebook, choose its Use tab and choose Create copy.
Instance Recommendations for Multi-Model Endpoint Deployments

There are several items to consider when selecting a SageMaker ML instance type for a multi-model endpoint. Provision sufficient Amazon Elastic Block Store (Amazon EBS) capacity for all of the models that need to be served. Balance performance (minimize cold starts) and cost (don't over-provision instance capacity). For information about the size of the storage volume that SageMaker attaches for each instance type for an endpoint and for a multi-model endpoint, see Host Instance Storage Volumes (p. 1355). For a container configured to run in MultiModel mode, the storage volume provisioned for its instances has more memory. This allows more models to be cached on the instance storage volume.

When choosing a SageMaker ML instance type, consider the following:

- Multi-model endpoints are not supported on GPU instance types.
- The traffic distribution (access patterns) to the models that you want to host behind the multi-model endpoint, along with the model size (how many models could be loaded in memory on the instance):
  - Think of the amount of memory on an instance as the cache space for models to be loaded. Think of the number of vCPUs as the concurrency limit to perform inference on the loaded models (assuming that invoking a model is bound to CPU).
  - A higher amount of instance memory enables you to have more models loaded and ready to serve inference requests. You don't need to waste time loading the model.
  - A higher amount of vCPUs enables you to invoke more unique models concurrently (again assuming that inference is bound to CPU).
  - Have some "slack" memory available so that unused models can be unloaded, and especially for multi-model endpoints with multiple instances. If an instance or an Availability Zone fails, the models on those instances will be rerouted to other instances behind the endpoint.
- Tolerance to loading/downloading times:
  - d instance type families (for example, m5d, c5d, or r5d) come with an NVMe (non-volatile memory express) SSD, which offers high I/O performance and might reduce the time it takes to download models to the storage volume and for the container to load the model from the storage volume.
  - Because d instance types come with an NVMe SSD storage, SageMaker does not attach an Amazon EBS storage volume to these ML compute instances that hosts the multi-model endpoint. Auto scaling works best when the models are similar sized and homogenous, that is when they have similar inference latency and resource requirements.

In some cases, you might opt to reduce costs by choosing an instance type that can't hold all of the targeted models in memory at once. SageMaker dynamically unloads models when it runs out of memory to make room for a newly targeted model. For infrequently requested models, you are going to pay a price with the dynamic load latency. In cases with more stringent latency needs, you might opt for larger instance types or more instances. Investing time up front for proper performance testing and analysis will pay great dividends in successful production deployments.

You can use the Average statistic of the ModelCacheHit metric to monitor the ratio of requests where the model is already loaded. You can use the SampleCount statistic for the ModelUnloadingTime metric to monitor the number of unload requests sent to the container during a time period. If models are unloaded too frequently (an indicator of thrashing, where models are being unloaded and loaded again because there is insufficient cache space for the working set of models), consider using a larger instance type with more memory or increasing the number of instances behind the multi-model endpoint. For multi-model endpoints with multiple instances, be aware that a model might be loaded on more than 1 instance.

SageMaker multi-model endpoints fully supports Auto Scaling, which manages replicas of models to ensure models scale based on traffic patterns. We recommend that you configure your multi-model
You can use AWS SDK for Python (Boto) or Amazon SageMaker to create a multi-model endpoint.

**Create a Multi-Model Endpoint (AWS SDK for Python (Boto))**

You create a multi-model endpoint using the Amazon SageMaker `create_model`, `create_endpoint_config`, and `create_endpoint` APIs just as you would create a single model endpoint, but with two changes. When defining the model container, you need to pass a new `Mode` parameter value, `MultiModel`. You also need to pass the `ModelDataUrl` field that specifies the prefix in Amazon S3 where the model artifacts are located, instead of the path to a single model artifact, as you would when deploying a single model.

For a sample notebook that uses SageMaker to deploy multiple XGBoost models to an endpoint, see Multi-Model Endpoint XGBoost Sample Notebook.

The following procedure outlines the key steps used in that sample to create a multi-model endpoint.

**To deploy the model (AWS SDK for Python (Boto 3))**

1. Get a container with an image that supports deploying multi-model endpoints. For a list of built-in algorithms and framework containers that support multi-model endpoints, see Supported Algorithms and Frameworks (p. 1161). For this example, we use the K-Nearest Neighbors (K-NN) Algorithm (p. 719) built-in algorithm. We call the SageMaker Python SDK utility function `image_uris.retrieve()` to get the address for the K-Nearest Neighbors built-in algorithm image.

   ```python
   import sagemaker
   image = sagemaker.image_uris.retrieve("knn")
   container = {
       'Image': image,
       'ModelDataUrl': 's3://my-bucket/path/to/artifacts/',
       'Mode': 'MultiModel'
   }
   ```

2. Create the model that uses this container.

   ```python
   response = sm_client.create_model(
       ModelName = 'my-multi-model-name',
       ExecutionRoleArn = role,
       Containers = [container])
   ```

3. (Optional) If you are using a serial inference pipeline, get the additional container(s) to include in the pipeline, and include it in the `Containers` argument of `CreateModel`:

   ```python
   preprocessor_container = {
       'Image': '123456789012.dkr.ecr.us-east-1.amazonaws.com/mypreprocessorimage:mytag'
   }
   ```
Create a Multi-Model Endpoint

4. Configure the multi-model endpoint for the model. We recommend configuring your endpoints with at least two instances. This allows SageMaker to provide a highly available set of predictions across multiple Availability Zones for the models.

```python
response = sm_client.create_endpoint_config(
    EndpointConfigName = 'my-epc',
    ProductionVariants=[[
        'InstanceType': 'ml.m4.xlarge',
        'InitialInstanceCount': 2,
        'InitialVariantWeight': 1,
        'ModelName': 'my-multi-model-name',
        'VariantName': 'AllTraffic'
    ]]
)
```

**Note**
You can use only one multi-model-enabled endpoint in a serial inference pipeline.

5. Create the multi-model endpoint using the EndpointName and EndpointConfigName parameters.

```python
response = sm_client.create_endpoint(
    EndpointName      = 'my-endpoint',
    EndpointConfigName = 'my-epc'
)
```

Create a Multi-Model Endpoint (Console)

**To create a multi-model endpoint (console)**

1. Open the Amazon SageMaker console at https://console.aws.amazon.com/sagemaker/.
2. Choose Model, and then from the Inference group, choose Create model.
3. For Model name, enter a name.
4. For IAM role, choose or create an IAM role that has the AmazonSageMakerFullAccess IAM policy attached.
5. In the Container definition section, for Provide model artifacts and inference image options, choose Use multiple models.
Create a Multi-Model Endpoint

Model settings

Model name

mml-test-model

Maximum of 63 alphanumeric characters. Can include hyphens (-), but not spaces. Must be unique for your account in an AWS Region.

IAM role

Amazon SageMaker requires permissions to call other services on your behalf. Choose a role or let Amazon SageMaker create a role that has the AmazonSageMakerFullAccess IAM policy attached.

AmazonSageMaker-ExecutionRole- xxxxxxxxxxxxxxxxxxxx

Container definition 1

Container input options

Provide model artifacts and inference image location

Provide model artifacts and inference image options

Use a single model

Use this to host a single model in this container.

Use multiple models

Use this to host multiple models in this container.

Location of inference code image

Type the registry path where the inference code image is stored in Amazon ECR.

123456789012.dkr.ecr.us-east-1.amazonaws.com/myimage:mytag

Location of model artifacts

Type the URL where model artifacts are stored in S3.

s3://my-bucket/path/to/artifacts/
6. Choose **Create model**.
7. Deploy your multi-model endpoint as you would a single model endpoint. For instructions, see [Step 6.1: Deploy the Model to SageMaker Hosting Services (p. 68)](#).

---

## Invoke a Multi-Model Endpoint

To invoke a multi-model endpoint, use the `invoke_endpoint` from the SageMaker Runtime just as you would invoke a single model endpoint, with one change. Pass a new `TargetModel` parameter that specifies which of the models at the endpoint to target. The SageMaker Runtime `InvokeEndpoint` request supports `X-Amzn-SageMaker-Target-Model` as a new header that takes the relative path of the model specified for invocation. The SageMaker system constructs the absolute path of the model by combining the prefix that is provided as part of the `CreateModel` API call with the relative path of the model.

The following example prediction request uses the AWS SDK for Python (Boto 3) in the sample notebook.

```python
response = runtime_sm_client.invoke_endpoint(
    EndpointName = 'my-endpoint',
    ContentType = 'text/csv',
    TargetModel = 'Houston_TX.tar.gz',
    Body         = body)
```

The multi-model endpoint dynamically loads target models as needed. You can observe this when running the MME Sample Notebook as it iterates through random invocations against multiple target models hosted behind a single endpoint. The first request against a given model takes longer because the model has to be downloaded from Amazon Simple Storage Service (Amazon S3) and loaded into memory. (This is called a cold start.) Subsequent calls finish faster because there's no additional overhead after the model has loaded.

## Retry Requests on ModelNotReadyException Errors

The first time you call `invoke_endpoint` for a model, the model is downloaded from Amazon Simple Storage Service and loaded into the inference container. This makes the first call take longer to return. Subsequent calls to the same model finish faster, because the model is already loaded.

SageMaker returns a response for a call to `invoke_endpoint` within 60 seconds. Some models are too large to download within 60 seconds. If the model does not finish loading before the 60 second timeout limit, the request to `invoke_endpoint` returns with the error code `ModelNotReadyException`, and the model continues to download and load into the inference container for up to 360 seconds. If you get a `ModelNotReadyException` error code for an `invoke_endpoint` request, retry the request. By default, the AWS SDKs for Python (Boto 3) (using [Legacy retry mode](#)) and Java retry `invoke_endpoint` requests that result in `ModelNotReadyException` errors. You can configure the retry strategy to continue retrying the request for up to 360 seconds. If you expect your model to take longer than 60 seconds to download and load into the container, set the SDK socket timeout to 70 seconds. For more information about configuring the retry strategy for Boto 3, see [Configuring a retry mode](#). The following code shows an example that configures the retry strategy to retry calls to `invoke_endpoint` for up to 180 seconds.

```python
import boto3
from botocore.config import Config

# This example retry strategy sets the retry attempts to 2.
# With this setting, the request can attempt to download and/or load the model
# for up to 180 seconds: 1 original request (60 seconds) + 2 retries (120 seconds)
config = Config(
    read_timeout=70,
    ...,  # other settings
)
```
Add or Remove Models

You can deploy additional models to a multi-model endpoint and invoke them through that endpoint immediately. When adding a new model, you don’t need to update or bring down the endpoint, so you avoid the cost of creating and running a separate endpoint for each new model.

SageMaker unloads unused models from the container when the instance is reaching memory capacity and more models need to be downloaded into the container. SageMaker also deletes unused model artifacts from the instance storage volume when the volume is reaching capacity and new models need to be downloaded. The first invocation to a newly added model takes longer because the endpoint takes time to download the model from S3 to the container's memory in instance hosting the endpoint.

With the endpoint already running, copy a new set of model artifacts to the Amazon S3 location where you store your models.

```
aws s3 cp AdditionalModel.tar.gz s3://my-bucket/path/to/artifacts/
```

Important

To update a model, proceed as you would when adding a new model. Use a new and unique name. Don’t overwrite model artifacts in Amazon S3 because the old version of the model might still be loaded in the containers or on the storage volume of the instances on the endpoint. Invocations to the new model could then invoke the old version of the model.

Client applications can request predictions from the additional target model as soon as it is stored in S3.

```
response = runtime_sm_client.invoke_endpoint(
    EndpointName='endpoint_name',
    ContentType='text/csv',
    TargetModel='AdditionalModel.tar.gz',
    Body=body
)
```

To delete a model from a multi-model endpoint, stop invoking the model from the clients and remove it from the S3 location where model artifacts are stored.

Build Your Own Container with Multi Model Server

Custom Elastic Container Registry (ECR) images deployed in Amazon SageMaker are expected to adhere to the basic contract described in Use Your Own Inference Code with Hosting Services (p. 1387) that govern how SageMaker interacts with a Docker container that runs your own inference code. For a container to be capable of loading and serving multiple models concurrently, there are additional APIs and behaviors that must be followed. This additional contract includes new APIs to load, list, get, and unload models, and a different API to invoke models. There are also different behaviors for error scenarios that the APIs need to abide by. To indicate that the container complies with the additional requirements, you can add the following command to your Docker file:

```
LABEL com.amazonaws.sagemaker.capabilities.multi-models=true
```

SageMaker also injects an environment variable into the container.
If you are creating a multi-model endpoint for a serial inference pipeline, your Docker file must have the required labels for both multi-models and serial inference pipelines. For more information about serial information pipelines, see Run Real-time Predictions with an Inference Pipeline (p. 1224).

To help you implement these requirements for a custom container, two libraries are available:

- **Multi Model Server** is an open source framework for serving machine learning models that can be installed in containers to provide the front end that fulfills the requirements for the new multi-model endpoint container APIs. It provides the HTTP front end and model management capabilities required by multi-model endpoints to host multiple models within a single container, load models into and unload models out of the container dynamically, and performs inference on a specified loaded model. It also provides a pluggable backend that supports a pluggable custom backend handler where you can implement your own algorithm.

- **SageMaker Inference Toolkit** is a library that bootstraps Multi Model Server with a configuration and settings that make it compatible with SageMaker multi-model endpoints. It also allows you to tweak important performance parameters, such as the number of workers per model, depending on the needs of your scenario.

## Use the SageMaker Inference Toolkit

Currently, the only pre-built containers that support multi-model endpoints are the MXNet inference container and the PyTorch inference container. If you want to use any other framework or algorithm, you need to build a container. The easiest way to do this is to use the SageMaker Inference Toolkit to extend an existing pre-built container. The SageMaker inference toolkit is an implementation for the multi-model server (MMS) that creates endpoints that can be deployed in SageMaker. For a sample notebook that shows how to set up and deploy a custom container that supports multi-model endpoints in SageMaker, see the Multi-Model Endpoint BYOC Sample Notebook.

**Note**
The SageMaker inference toolkit supports only Python model handlers. If you want to implement your handler in any other language, you must build your own container that implements the additional multi-model endpoint APIs. For information, see Contract for Custom Containers to Serve Multiple Model (p. 1170).

### To extend a container by using the SageMaker inference toolkit

1. Create a model handler. MMS expects a model handler, which is a Python file that implements functions to pre-process, get predictions from the model, and process the output in a model handler. For an example of a model handler, see `model_handler.py` from the sample notebook.

2. Import the inference toolkit and use its `model_server.start_model_server` function to start MMS. The following example is from the `dockerd-entrypoint.py` file from the sample notebook. Notice that the call to `model_server.start_model_server` passes the model handler described in the previous step:

```python
import subprocess
import sys
import shlex
import os
from retrying import retry
from subprocess import CalledProcessError
from sagemaker_inference import model_server

def _retry_if_error(exception):
    return isinstance(exception, CalledProcessError or OSError)
```

```python
import subprocess
import sys
import shlex
import os
from retrying import retry
from subprocess import CalledProcessError
from sagemaker_inference import model_server

def _retry_if_error(exception):
    return isinstance(exception, CalledProcessError or OSError)
```
@retry(stop_max_delay=1000 * 50,
    retry_on_exception=_retry_if_error)
def _start_mms():
    # by default the number of workers per model is 1, but we can configure it through
    # environment variable below if desired.
    # os.environ['SAGEMAKER_MODEL_SERVER_WORKERS'] = '2'
    model_server.start_model_server(handler_service="/home/model-server/model_handler.py:handle")

def main():
    if sys.argv[1] == 'serve':
        _start_mms()
    else:
        subprocess.check_call(shlex.split(' '.join(sys.argv[1:])))

        # prevent docker exit
        subprocess.call(['tail', '-f', '/dev/null'])

main()

3. In your Dockerfile, copy the model handler from the first step and specify the Python file from
the previous step as the entrypoint in your Dockerfile. The following lines are from the Dockerfile
used in the sample notebook:

```bash
# Copy the default custom service file to handle incoming data and inference requests
COPY model_handler.py /home/model-server/model_handler.py

# Define an entrypoint script for the docker image
ENTRYPOINT ["python", "/usr/local/bin/dockerd-entrypoint.py"]
```

4. Build and register your container. The following shell script from the sample notebook builds the
container and uploads it to an Amazon Elastic Container Registry repository in your AWS account:

```bash
%%sh

# The name of our algorithm
algorithm_name=demo-sagemaker-multimodel

cd container

account=$(aws sts get-caller-identity --query Account --output text)

# Get the region defined in the current configuration (default to us-west-2 if none
defined)
region=$(aws configure get region)
region=${region:-us-west-2}

fullname="$(account).dkr.ecr.$(region).amazonaws.com/$algorithm_name:latest"

# If the repository doesn't exist in ECR, create it.
aws ecr describe-repositories --repository-names "$algorithm_name" > /dev/null 2>&1
if [ "$algorithm_name" != "$algorithm_name" ]
then
    aws ecr create-repository --repository-name "$algorithm_name" > /dev/null
fi

# Get the login command from ECR and execute it directly
$(aws ecr get-login --region $(region) --no-include-email)

# Build the docker image locally with the image name and then push it to ECR
# with the full name.
```
Bring Your Own Container

```
docker build -q -t ${algorithm_name} .
docker tag ${algorithm_name} ${fullname}
docker push ${fullname}
```

You can now use this container to deploy multi-model endpoints in SageMaker.

**Topics**
- Contract for Custom Containers to Serve Multiple Model (p. 1170)

**Contract for Custom Containers to Serve Multiple Model**

To handle multiple models, your container must support a set of APIs that enable Amazon SageMaker to communicate with the container for loading, listing, getting, and unloading models as required. The `model_name` is used in the new set of APIs as the key input parameter. The customer container is expected to keep track of the loaded models using `model_name` as the mapping key. Also, the `model_name` is an opaque identifier and is not necessarily the value of the TargetModel parameter passed into the InvokeEndpoint API. The original TargetModel value in the InvokeEndpoint request is passed to container in the APIs as a `X-Amzn-SageMaker-Target-Model` header that can be used for logging purposes.

**Topics**
- LOAD MODEL API (p. 1170)
- LIST MODEL API (p. 1170)
- GET MODEL API (p. 1171)
- UNLOAD MODEL API (p. 1171)
- INVOKE MODEL API (p. 1172)

**LOAD MODEL API**

Instructs the container to load a particular model present in the `url` field of the body into the memory of the customer container and to keep track of it with the assigned `model_name`. After a model is loaded, the container should be ready to serve inference requests using this `model_name`.

```
POST /models HTTP/1.1
Content-Type: application/json
Accept: application/json

{
    "model_name" : "{model_name}"
    "url" : "/opt/ml/models/{model_name}/model",
}
```

**Note**

If `model_name` is already loaded, this API should return 409. Any time a model cannot be loaded due to lack of memory or to any other resource, this API should return a 507 HTTP status code to SageMaker, which then initiates unloading unused models to reclaim.

**LIST MODEL API**

Returns the list of models loaded into the memory of the customer container.

```
GET /models HTTP/1.1
Accept: application/json
```
Response =
{
    "models": [
        {
            "modelName" : "{model_name}",
            "modelUrl" : "/opt/ml/models/{model_name}/model",
        },
        {
            "modelName" : "{model_name}",
            "modelUrl" : "/opt/ml/models/{model_name}/model",
        },
        ....
    ]
}

This API also supports pagination.

GET /models HTTP/1.1
Accept: application/json
Response =
{
    "models": [
        {
            "modelName" : "{model_name}",
            "modelUrl" : "/opt/ml/models/{model_name}/model",
        },
        {
            "modelName" : "{model_name}",
            "modelUrl" : "/opt/ml/models/{model_name}/model",
        },
        ....
    ]
}

SageMaker can initially call the List Models API without providing a value for next_page_token. If a nextPageToken field is returned as part of the response, it will be provided as the value for next_page_token in a subsequent List Models call. If a nextPageToken is not returned, it means that there are no more models to return.

**GET MODEL API**

This is a simple read API on the model_name entity.

GET /models/{model_name} HTTP/1.1
Accept: application/json

{
    "modelName" : "{model_name}",
    "modelUrl" : "/opt/ml/models/{model_name}/model",
}

**Note**

If model_name is not loaded, this API should return 404.

**UNLOAD MODEL API**

Instructs the SageMaker platform to instruct the customer container to unload a model from memory. This initiates the eviction of a candidate model as determined by the platform when starting the process.
of loading a new model. The resources provisioned to `model_name` should be reclaimed by the container when this API returns a response.

DELETE /models/{model_name}

**Note**
If `model_name` is not loaded, this API should return 404.

**INVOKE MODEL API**

Makes a prediction request from the particular `model_name` supplied. The SageMaker Runtime InvokeEndpoint request supports `X-Amzn-SageMaker-Target-Model` as a new header that takes the relative path of the model specified for invocation. The SageMaker system constructs the absolute path of the model by combining the prefix that is provided as part of the `CreateModel` API call with the relative path of the model.

POST /models/{model_name}/invoke HTTP/1.1
Content-Type: Content-Type
Accept: Accept
X-Amzn-SageMaker-Custom-Attributes: CustomAttributes
X-Amzn-SageMaker-Target-Model: [relativePath]/{artifactName}.tar.gz

**Note**
If `model_name` is not loaded, this API should return 404.

**How Multi-Model Endpoints Work**

SageMaker manages the lifecycle of models hosted on multi-model endpoints in the container's memory. Instead of downloading all of the models from an Amazon S3 bucket to the container when you create the endpoint, SageMaker dynamically loads them when you invoke them. When SageMaker receives an invocation request for a particular model, it does the following:

1. Routes the request to an instance behind the endpoint.
2. Downloads the model from the S3 bucket to that instance's storage volume.
3. Loads the model to the container's memory on that instance. If the model is already loaded in the container's memory, invocation is faster because SageMaker doesn't need to download and load it.

SageMaker continues to route requests for a model to the instance where the model is already loaded. However, if the model receives many invocation requests, and there are additional instances for the multi-model endpoint, SageMaker routes some requests to another instance to accommodate the traffic. If the model isn't already loaded on the second instance, the model is downloaded to that instance's storage volume and loaded into the container's memory.

When an instance's memory utilization is high and SageMaker needs to load another model into memory, it unloads unused models from that instance's container to ensure that there is enough memory to load the model. Models that are unloaded remain on the instance's storage volume and can be loaded into the container's memory later without being downloaded again from the S3 bucket. If the instance's storage volume reaches its capacity, SageMaker deletes any unused models from the storage volume.

To delete a model, stop sending requests and delete it from the S3 bucket. SageMaker provides multi-model endpoint capability in a serving container. Adding models to, and deleting them from, a multi-model endpoint doesn't require updating the endpoint itself. To add a model, you upload it to the S3 bucket and invoke it. You don't need code changes to use it.

When you update a multi-model endpoint, invocation requests on the endpoint might experience higher latencies as traffic is directed to the instances in the updated endpoint.
Multi-Model Endpoint Security

Models and data in a multi-model endpoint are co-located on instance storage volume and in container memory. All instances for Amazon SageMaker endpoints run on a single tenant container that you own. Only your models can run on your multi-model endpoint. It’s your responsibility to manage the mapping of requests to models and to provide access for users to the correct target models. SageMaker uses IAM roles to provide IAM identity-based policies that you use to specify allowed or denied actions and resources and the conditions under which actions are allowed or denied.

By default, an IAM principal with `InvokeEndpoint` permissions on a multi-model endpoint can invoke any model at the address of the S3 prefix defined in the `CreateModel` operation, provided that the IAM Execution Role defined in operation has permissions to download the model. If you need to restrict `InvokeEndpoint` access to a limited set of models in S3, you can do one of the following:

- **Restrict `InvokeEndpoint` calls to specific models hosted at the endpoint by using the `sagemaker:TargetModel` IAM condition key.** For example, the following policy allows `InvokeEndpoint` requests only when the value of the `TargetModel` field matches one of the specified regular expressions:

```json
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Action": ["sagemaker:InvokeEndpoint"],
            "Effect": "Allow",
            "Condition": {
                // TargetModel provided must be from this set of values
                "StringLike": {
                    "sagemaker:TargetModel": ["company_a/*", "common/*"]
                }
            }
        }
    ]
}
```


- **Create multi-model endpoints with more restrictive S3 prefixes.**

For more information about how SageMaker uses roles to manage access to endpoints and perform operations on your behalf, see [SageMaker Roles](https://docs.aws.amazon.com/sagemaker/latest/dg/sagemaker-roles.html) (p. 1647). Your customers might also have certain data isolation requirements dictated by their own compliance requirements that can be satisfied using IAM identities.

CloudWatch Metrics for Multi-Model Endpoint Deployments

Amazon SageMaker provides metrics for endpoints so you can monitor the cache hit rate, the number of models loaded, and the model wait times for loading, downloading, and uploading at a multi-model endpoint. For information, see [Multi-Model Endpoint Model Loading Metrics](https://docs.aws.amazon.com/sagemaker/latest/dg/metrics-loading.html) and [Multi-Model Endpoint Model Instance Metrics](https://docs.aws.amazon.com/sagemaker/latest/dg/metrics-instance.html) in Monitor Amazon SageMaker with Amazon CloudWatch (p. 1704). Per-model metrics aren’t supported.
Amazon SageMaker Model Monitor

Amazon SageMaker Model Monitor continuously monitors the quality of Amazon SageMaker machine learning models in production. With Model Monitor, you can set alerts that notify you when there are deviations in the model quality. Early and proactive detection of these deviations enables you to take corrective actions, such as retraining models, auditing upstream systems, or fixing quality issues without having to monitor models manually or build additional tooling. You can use Model Monitor prebuilt monitoring capabilities that do not require coding. You also have the flexibility to monitor models by coding to provide custom analysis.

Model Monitor provides the following types of monitoring:

- **Monitor Data Quality** (p. 1176) - Monitor drift in data quality.
- **Monitor Model Quality** (p. 1181) - Monitor drift in model quality metrics, such as accuracy.
- **Monitor Bias Drift for Models in Production** (p. 1187) - Monitor bias in your model's predictions.
- **Monitor Feature Attribution Drift for Models in Production** (p. 1191) - Monitor drift in feature attribution.

**Topics**

- How Model Monitor Works (p. 1174)
- Monitor Data Quality (p. 1176)
- Monitor Model Quality (p. 1181)
- Monitor Bias Drift for Models in Production (p. 1187)
- Monitor Feature Attribution Drift for Models in Production (p. 1191)
- Capture Data (p. 1195)
- Schedule Monitoring Jobs (p. 1196)
- Amazon SageMaker Model Monitor Prebuilt Container (p. 1198)
- Interpret Results (p. 1199)
- Visualize Results in Amazon SageMaker Studio (p. 1201)
- Advanced Topics (p. 1207)

**How Model Monitor Works**

Amazon SageMaker Model Monitor automatically monitors machine learning (ML) models in production and notifies you when quality issues arise. Model Monitor uses rules to detect drift in your models and alerts you when it happens. The following figure shows how this process works.
To enable model monitoring, you take the following steps, which follow the path of the data through the various data collection, monitoring, and analysis processes:

- Enable the endpoint to capture data from incoming requests to a trained ML model and the resulting model predictions.
- Create a baseline from the dataset that was used to train the model. The baseline computes metrics and suggests constraints for the metrics. Real-time predictions from your model are compared to the constraints, and are reported as violations if they are outside the constrained values.
- Create a monitoring schedule specifying what data to collect, how often to collect it, how to analyze it, and which reports to produce.
- Inspect the reports, which compare the latest data with the baseline, and watch for any violations reported and for metrics and notifications from Amazon CloudWatch.

Notes

- Model Monitor currently supports only endpoints that host a single model and does not support monitoring multi-model endpoints. For information on using multi-model endpoints, see Host Multiple Models with Multi-Model Endpoints (p. 1160).
- Model Monitor supports monitoring inference pipelines, but capturing and analyzing data is done for the entire pipeline, not for individual containers in the pipeline.
- If you launch SageMaker Studio in a custom Amazon VPC, you need to create VPC endpoints to enable Model Monitor to communicate with Amazon S3 and CloudWatch. For information about VPC endpoints, see VPC endpoints in the Amazon Virtual Private Cloud User Guide. For information about launching SageMaker Studio in a custom VPC, see Connect SageMaker Studio Notebooks to Resources in a VPC (p. 1679).

Model Monitor Sample Notebooks

For a sample notebook that takes you through the full end-to-end workflow for Model Monitor, see the Introduction to Amazon SageMaker Model Monitor.

For a sample notebook that enables the model monitoring experience for an existing endpoint, see the Enable Model Monitoring.
For a sample notebook that visualizes the statistics.json file for a selected execution in a monitoring schedule, see the Model Monitor Visualization.

For instructions that show you how to create and access Jupyter notebook instances that you can use to run the example in SageMaker, see Use Amazon SageMaker Notebook Instances (p. 124). After you have created a notebook instance and opened it, choose the SageMaker Examples tab to see a list of all the SageMaker samples. To open a notebook, choose the notebook’s Use tab and choose Create copy.

Monitor Data Quality

Data quality monitoring automatically monitors machine learning (ML) models in production and notifies you when data quality issues arise. ML models in production have to make predictions on real-life data that is not carefully curated like most training datasets. If the statistical nature of the data that your model receives while in production drifts away from the nature of the baseline data it was trained on, the model begins to lose accuracy in its predictions. Amazon SageMaker Model Monitor uses rules to detect data drift and alerts you when it happens. To monitor data quality, follow these steps:

• Enable data capture. This captures inference input and output from a real-time inference endpoint and stores the data in Amazon S3. For more information, see Capture Data (p. 1195).
• Create a baseline. In this step, you run a baseline job that analyzes an input dataset that you provide. The baseline computes baseline schema constraints and statistics for each feature using Deequ, an open source library built on Apache Spark, which is used to measure data quality in large datasets. For more information, see Create a Baseline (p. 1176).
• Define and schedule data quality monitoring jobs. For more information, see Schedule Monitoring Jobs (p. 1196).
• View data quality metrics. For more information, see Schema for Statistics (statistics.json file) (p. 1178).
• Integrate data quality monitoring with Amazon CloudWatch. For more information, see CloudWatch Metrics (p. 1179).
• Interpret the results of a monitoring job. For more information, see Interpret Results (p. 1199).
• Use SageMaker Studio to enable data quality monitoring and visualize results. For more information, see Visualize Results in Amazon SageMaker Studio (p. 1201).

Topics
• Create a Baseline (p. 1176)
• Schema for Statistics (statistics.json file) (p. 1178)
• CloudWatch Metrics (p. 1179)
• Schema for Violations (constraint_violations.json file) (p. 1179)

Create a Baseline

The baseline calculations of statistics and constraints are needed as a standard against which data drift and other data quality issues can be detected. Model Monitor provides a built-in container that provides the ability to suggest the constraints automatically for CSV and flat JSON input. This sagemaker-model-monitor-analyzer container also provides you with a range of model monitoring capabilities, including constraint validation against a baseline, and emitting Amazon CloudWatch metrics. This container is based on Spark and is built with Deequ. All column names in your baseline dataset must be compliant with Spark. For column names, use only lowercase characters, and _ as the only special character.

The training dataset that you used to trained the model is usually a good baseline dataset. The training dataset data schema and the inference dataset schema should exactly match (the number and order of the features). Note that the prediction/output columns are assumed to be the first columns in the training dataset. From the training dataset, you can ask SageMaker to suggest a set of baseline
Monitor Data Quality

To Create a baseline from a training dataset

When you have your training data ready and stored in Amazon S3, start a baseline processing job with `DefaultModelMonitor.suggest_baseline(..)` using the Amazon SageMaker Python SDK. This uses an Amazon SageMaker Model Monitor Prebuilt Container (p. 1198) that generates baseline statistics and suggests baseline constraints for the dataset and writes them to the `output_s3_uri` location that you specify.

```python
from sagemaker.model_monitor import DefaultModelMonitor
from sagemaker.model_monitor.dataset_format import DatasetFormat

my_default_monitor = DefaultModelMonitor(
    role=role,
    instance_count=1,
    instance_type='ml.m5.xlarge',
    volume_size_in_gb=20,
    max_runtime_in_seconds=3600,
)

my_default_monitor.suggest_baseline(
    baseline_dataset=baseline_data_uri+'/training-dataset-with-header.csv',
    dataset_format=DatasetFormat.csv(header=True),
    output_s3_uri=baseline_results_uri,
    wait=True
)
```

**Note**

If you provide the feature/column names in the training dataset as the first row and set the `header=True` option as shown in the previous code sample, SageMaker uses the feature name in the constraints and statistics file.

The baseline statistics for the dataset are contained in the `statistics.json` file and the suggested baseline constraints are contained in the `constraints.json` file in the location you specify with `output_s3_uri`.

### Output Files for Tabular Dataset Statistics and Constraints

<table>
<thead>
<tr>
<th>File Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>statistics.json</strong></td>
<td>This file is expected to have columnar statistics for each feature in the dataset that is analyzed. For more information about the schema for this file, see <a href="#">Schema for Statistics (statistics.json file) (p. 1212)</a>.</td>
</tr>
<tr>
<td><strong>constraints.json</strong></td>
<td>This file is expected to have the constraints on the features observed. For more information about the schema for this file, see <a href="#">Schema for Constraints (constraints.json file) (p. 1214)</a>.</td>
</tr>
</tbody>
</table>

The Amazon SageMaker Python SDK provides convenience functions described to generate the baseline statistics and constraints. But if you want to call processing job directly for this purpose instead, you need to set the `Environment` map as shown in the following example:

```json
"Environment": {
   "dataset_format": "$\"csv\": \{"header\": true\} 
   "dataset_source": "/opt/ml/processing/sm_input",
```
Schema for Statistics (statistics.json file)

Amazon SageMaker Model Monitor prebuilt container computes per column/feature statistics. The statistics are calculated for the baseline dataset and also for the current dataset that is being analyzed.

```json
{
    "version": 0,
    # dataset level stats
    "dataset": {
        "item_count": number
    },
    # feature level stats
    "features": [
        {
            "name": "feature-name",
            "inferred_type": "Fractional" | "Integral",
            "numerical_statistics": {
                "common": {
                    "num_present": number,
                    "num_missing": number
                },
                "mean": number,
                "sum": number,
                "std_dev": number,
                "min": number,
                "max": number,
                "distribution": {
                    "kll": {
                        "buckets": [
                            {
                                "lower_bound": number,
                                "upper_bound": number,
                                "count": number
                            }
                        ],
                        "sketch": {
                            "parameters": {
                                "c": number,
                                "k": number
                            },
                            "data": [
                                [num, num, num, num],
                                [num, num]
                            ]
                        }
                    }
                }
            }
        }
    ]
}
```

"output_path": "/opt/ml/processing/sm_output",
"publish_cloudwatch_metrics": "Disabled",
}
Note the following:

- The prebuilt containers compute KLL sketch, which is a compact quantiles sketch.
- By default, we materialize the distribution in 10 buckets. This is not currently configurable.

## CloudWatch Metrics

You can use the built-in Amazon SageMaker Model Monitor container for CloudWatch metrics. When the `emit_metrics` option is Enabled in the baseline constraints file, SageMaker emits these metrics for each feature/column observed in the dataset in the `/aws/sagemaker/Endpoints/data-metric` namespace with `EndpointName` and `ScheduleName` dimensions.

For numerical fields, the built-in container emits the following CloudWatch metrics:

- Metric: Max → query for MetricName: `feature_data_{feature_name}`, Stat: Max
- Metric: Min → query for MetricName: `feature_data_{feature_name}`, Stat: Min
- Metric: Sum → query for MetricName: `feature_data_{feature_name}`, Stat: Sum
- Metric: SampleCount → query for MetricName: `feature_data_{feature_name}`, Stat: SampleCount
- Metric: Average → query for MetricName: `feature_data_{feature_name}`, Stat: Average

For both numerical and string fields, the built-in container emits the following CloudWatch metrics:

- Metric: Completeness → query for MetricName: `feature_non_null_{feature_name}`, Stat: Sum
- Metric: Baseline Drift → query for MetricName: `feature_baseline_drift_{feature_name}`, Stat: Sum

## Schema for Violations (constraint_violations.json file)

The violations file is generated as the output of a MonitoringExecution, which lists the results of evaluating the constraints (specified in the constraints.json file) against the current dataset that was
analyzed. The Amazon SageMaker Model Monitor prebuilt container provides the following violation checks.

```
{
  "violations": [{
    "feature_name": "string",
    "constraint_check_type": "data_type_check",
    | "completeness_check",
    | "baseline_drift_check",
    | "missing_column_check",
    | "extra_column_check",
    | "categorical_values_check"
    "description": "string"
  }]
}
```

### Types of Violations Monitored

<table>
<thead>
<tr>
<th>Violation Check Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>data_type_check</strong></td>
<td>If the data types in the current execution are not the same as in the baseline dataset, this violation is flagged. During the baseline step, the generated constraints suggest the inferred data type for each column. The monitoring_config.datatype_check_threshold parameter can be tuned to adjust the threshold on when it is flagged as a violation.</td>
</tr>
<tr>
<td><strong>completeness_check</strong></td>
<td>If the completeness (% of non-null items) observed in the current execution exceeds the threshold specified in completeness threshold specified per feature, this violation is flagged. During the baseline step, the generated constraints suggest a completeness value.</td>
</tr>
<tr>
<td><strong>baseline_drift_check</strong></td>
<td>If the calculated distribution distance between the current and the baseline datasets is more than the threshold specified in monitoring_config.comparison_threshold, this violation is flagged.</td>
</tr>
<tr>
<td><strong>missing_column_check</strong></td>
<td>If the number of columns in the current dataset is less than the number in the baseline dataset, this violation is flagged.</td>
</tr>
<tr>
<td><strong>extra_column_check</strong></td>
<td>If the number of columns in the current dataset is more than the number in the baseline, this violation is flagged.</td>
</tr>
<tr>
<td><strong>categorical_values_check</strong></td>
<td>If there are more unknown values in the current dataset than in the baseline dataset, this violation is flagged. This value is dictated by the threshold in monitoring_config.domain_content_threshold.</td>
</tr>
</tbody>
</table>
Monitor Model Quality

Model quality monitoring jobs monitor the performance of a model by comparing the predictions that the model makes with the actual ground truth labels that the model attempts to predict. To do this, model quality monitoring merges data that is captured from real-time inference with actual labels that you store in an Amazon S3 bucket, and compares the predictions with the actual labels.

To measure model quality, model monitor uses metrics that depend on the ML problem type. For example, if your model is for a regression problem, one of the metrics evaluated is mean square error (mse). For information about all of the metrics used for the different ML problem types, see Model Quality Metrics (p. 1184).

Model quality monitoring follows the same steps as data quality monitoring, but adds the additional step of merging the actual labels from Amazon S3 with the predictions captured from the real-time inference endpoint. To monitor model quality, follow these steps:

• Enable data capture. This captures inference input and output from a real-time inference endpoint and stores the data in Amazon S3. For more information, see Capture Data (p. 1195).
• Create a baseline. In this step, you run a baseline job that compares predictions from the model with ground truth labels in a baseline dataset. The baseline job automatically creates baseline statistical rules and constraints that define thresholds against which the model performance is evaluated. For more information, see Create a Model Quality Baseline (p. 1181).
• Define and schedule model quality monitoring jobs. For more information, see Schedule Model Quality Monitoring Jobs (p. 1183).
• Ingest ground truth labels that model monitor merges with captured prediction data from real-time inference endpoint. For more information, see Ingest Ground Truth Labels and Merge Them With Predictions (p. 1183).
• Integrate model quality monitoring with Amazon CloudWatch. For more information, see Model Quality CloudWatch Metrics (p. 1187).
• Interpret the results of a monitoring job. For more information, see Interpret Results (p. 1199).
• Use SageMaker Studio to enable model quality monitoring and visualize results. For more information, see Visualize Results in Amazon SageMaker Studio (p. 1201).

Topics
• Create a Model Quality Baseline (p. 1181)
• Schedule Model Quality Monitoring Jobs (p. 1183)
• Ingest Ground Truth Labels and Merge Them With Predictions (p. 1183)
• Model Quality Metrics (p. 1184)
• Model Quality CloudWatch Metrics (p. 1187)

Create a Model Quality Baseline

Create a baseline job that compares your model predictions with ground truth labels in a baseline dataset that you have stored in Amazon S3. Typically, you use a training dataset as the baseline dataset. The baseline job calculates metrics for the model and suggests constraints to use to monitor model quality drift.

To create a baseline job, you need to have a dataset that contains predictions from your model along with labels that represent the ground truth for your data.

To create a baseline job use the ModelQualityMonitor class provided by the SageMaker Python SDK, and complete the following steps.

1181
To create a model quality baseline job

1. First, create an instance of the `ModelQualityMonitor` class. The following code snippet shows how to do this.

```python
from sagemaker import get_execution_role, session, Session
from sagemaker.model_monitor import ModelQualityMonitor

role = get_execution_role()
session = Session()

model_quality_monitor = ModelQualityMonitor(
    role=role,
    instance_count=1,
    instance_type='ml.m5.xlarge',
    volume_size_in_gb=20,
    max_runtime_in_seconds=1800,
    sagemaker_session=session
)
```

2. Now call the `suggest_baseline` method of the `ModelQualityMonitor` object to run a baseline job. The following code snippet assumes that you have a baseline dataset that contains both predictions and labels stored in Amazon S3.

```python
baseline_job_name = "MyBaselineJob"
job = model_quality_monitor.suggest_baseline(
    job_name=baseline_job_name,
    baseline_dataset=baseline_dataset_uri, # The S3 location of the validation dataset.
    dataset_format=DatasetFormat.csv(header=True),
    output_s3_uri = baseline_results_uri, # The S3 location to store the results.
    problem_type='BinaryClassification',
    inference_attribute= "prediction", # The column in the dataset that contains predictions.
    probability_attribute= "probability", # The column in the dataset that contains probabilities.
    ground_truth_attribute= "label" # The column in the dataset that contains ground truth labels.
)
job.wait(logs=False)
```

3. After the baseline job finishes, you can see the constraints that the job generated. First, get the results of the baseline job by calling the `latest_baselining_job` method of the `ModelQualityMonitor` object.

```python
baseline_job = model_quality_monitor.latest_baselining_job
```

4. The baseline job suggests constraints, which are thresholds for metrics that model monitor measures. If a metric goes beyond the suggested threshold, Model Monitor reports a violation. To view the constraints that the baseline job generated, call the `suggested_constraints` method of the baseline job. The following code snippet loads the constraints for a binary classification model into a Pandas dataframe.

```python
import pandas as pd

pd.DataFrame(baseline_job.suggested_constraints().body_dict["binary_classification_constraints"]).T
```

We recommend that you view the generated constraints and modify them as necessary before using them for monitoring. For example, if a constraint is too aggressive, you might get more alerts for violations than you want.
5. When you are satisfied with the constraints, pass them as the `constraints` parameter when you create a monitoring schedule. For more information, see `Schedule Model Quality Monitoring Jobs` (p. 1183).

The suggested baseline constraints are contained in the `constraints.json` file in the location you specify with `output_s3_uri`. For information about the schema for this file in the `Schema for Constraints (constraints.json file)` (p. 1214).

**Schedule Model Quality Monitoring Jobs**

For general information about scheduling monitoring jobs, see the section called “Schedule Monitoring Jobs” (p. 1196). For model quality monitoring, you also have to consider the fact that the availability of ground truth labels might be delayed.

To address this, use offsets. Model quality jobs include `StartOffset` and `EndOffset`, which are fields of the `ModelQualityJobInput` parameter of the `create_model_quality_job_definition` method that work as follows:

- **StartOffset** - If specified, jobs subtract this time from the start time.
- **EndOffset** - If specified, jobs subtract this time from the end time.

The format of the offsets are, for example, `-P7H`, where 7H is 7 hours. You can use `-P#H` or `-P#D`, where H=hours, D=days, and M=minutes, and # is the number.

For example, if your ground truth starts coming in after 1 day, but is not complete for a week, set `StartOffset` to `-P8D` and `EndOffset` to `-P1D`. Then, if you schedule a job to run at 2020-01-09T13:00, it analyzes data from between 2020-01-01T13:00 and 2020-01-08T13:00.

**Important**

The schedule cadence should be such that one execution finishes before the next execution starts, which allows the ground truth merge job and monitoring job from the execution to complete. The maximum runtime of an execution is divided between the two jobs, so for an hourly model quality monitoring job, the value of `MaxRuntimeInSeconds` specified as part of `StoppingCondition` should be no more than 1800.

**Ingest Ground Truth Labels and Merge Them With Predictions**

Model quality monitoring compares the predictions your model makes with ground truth labels to measure the quality of the model. For this to work, you periodically label data captured by your endpoint and upload it to Amazon S3.

To match ground truth labels with captured prediction data, there must be a unique identifier for each record in the dataset. The structure of each record for ground truth data is as follows:

```json
{
   "groundTruthData": {
      "data": "1",
      "encoding": "CSV" # only CSV supported at launch, we assume "data" only consists of label
   },
   "eventMetadata": {
      "eventId": "aaaa-bbbb-cccc"
   },
   "eventVersion": "0"
}
```

In the `groundTruthData` structure, `eventId` can be one of the following:
Monitor Model Quality

- **eventId** – This ID is automatically generated when a user invokes the endpoint.
- **inferenceId** – The caller supplies this ID when they invoke the endpoint.

If `inferenceId` is present in captured data records, Model Monitor uses it to merge captured data with ground truth records. You are responsible for making sure that the `inferenceId` in the ground truth records match the `inferenceId` in the captured records. If `inferenceId` is not present in captured data, model monitor uses `eventId` from the captured data records to match them with a ground truth record.

You must upload ground truth data to an Amazon S3 bucket that has the same path format as captured data, which is of the following form:

```
s3://bucket/prefixyyyy/mm/dd/hh
```

The date in this path is the date when the ground truth label is collected, and does not have to match the date when the inference was generated.

After you create and upload the ground truth labels, include the location of the labels as a parameter when you create the monitoring job. If you are using AWS SDK for Python (Boto3), do this by specifying the location of ground truth labels as the `S3Uri` field of the `GroundTruthS3Input` parameter in a call to the `create_model_quality_job_definition` method. If you are using the SageMaker Python SDK, specify the location of the ground truth labels as the `ground_truth_input` parameter in the call to the `create_monitoring_schedule` method of the `ModelQualityMonitor` object.

**Model Quality Metrics**

Model quality monitoring jobs compute different metrics depending on the ML problem type. The following sections list the metrics analyzed for each ML problem type.

**Note**

Standard deviation for metrics are provided only when at least 200 samples are available. Model Monitor computes standard deviation by randomly sampling 80% of the data 5 times, computing the metric, and taking the standard deviation for those results.

**Regression Metrics**

The following shows an example of the metrics that model quality monitor computes for a regression problem.

```json
"regression_metrics" : { 
  "mae" : { 
    "value" : 0.3711832061068702,
    "standard_deviation" : 0.0037566388129940394
  },
  "mse" : { 
    "value" : 0.3711832061068702,
    "standard_deviation" : 0.0037566388129940524
  },
  "rmse" : { 
    "value" : 0.609248066149471,
    "standard_deviation" : 0.003079253267651125
  },
  "r2" : { 
    "value" : -1.3766111872212665,
    "standard_deviation" : 0.022653980022771227
  }
}
```
Binary Classification Metrics

The following shows an example of the metrics that model quality monitor computes for a binary classification problem.

```
"binary_classification_metrics" : {
    "confusion_matrix" : {
        "0" : {
            "0" : 1,
            "1" : 2
        },
        "1" : {
            "0" : 0,
            "1" : 1
        }
    },
    "recall" : {
        "value" : 1.0,
        "standard_deviation" : "NaN"
    },
    "precision" : {
        "value" : 0.3333333333333333,
        "standard_deviation" : "NaN"
    },
    "accuracy" : {
        "value" : 0.5,
        "standard_deviation" : "NaN"
    },
    "recall_best_constant_classifier" : {
        "value" : 1.0,
        "standard_deviation" : "NaN"
    },
    "precision_best_constant_classifier" : {
        "value" : 0.25,
        "standard_deviation" : "NaN"
    },
    "accuracy_best_constant_classifier" : {
        "value" : 0.25,
        "standard_deviation" : "NaN"
    },
    "true_positive_rate" : {
        "value" : 1.0,
        "standard_deviation" : "NaN"
    },
    "true_negative_rate" : {
        "value" : 0.33333333333333337,
        "standard_deviation" : "NaN"
    },
    "false_positive_rate" : {
        "value" : 0.6666666666666666,
        "standard_deviation" : "NaN"
    },
    "false_negative_rate" : {
        "value" : 0.0,
        "standard_deviation" : "NaN"
    },
    "receiver_operating_characteristic_curve" : {
        "false_positive_rates" : [ 0.0, 0.0, 0.0, 0.0, 0.0, 1.0 ],
        "true_positive_rates" : [ 0.0, 0.25, 0.5, 0.75, 1.0, 1.0 ]
    },
    "precision_recall_curve" : {
        "precisions" : [ 1.0, 1.0, 1.0, 1.0, 1.0 ],
        "recalls" : [ 0.0, 0.25, 0.5, 0.75, 1.0 ]
    },
    "auc" : {
        "value" : 1185
    }
}
```
Multiclass Metrics

The following shows an example of the metrics that model quality monitor computes for a multiclass classification problem.

```
"multiclass_classification_metrics" : {
    "confusion_matrix" : {
        "0" : {
            "0" : 1180,
            "1" : 510
        },
        "1" : {
            "0" : 268,
            "1" : 138
        }
    },
    "accuracy" : {
        "value" : 0.6288167938931297,
        "standard_deviation" : 0.003756638812994008
    },
    "weighted_recall" : {
        "value" : 0.6288167938931297,
        "standard_deviation" : 0.003756638812994008
    },
    "weighted_precision" : {
        "value" : 0.6983172269629505,
        "standard_deviation" : 0.006195912915307507
    },
    "weighted_f0_5" : {
        "value" : 0.6803947317178771,
        "standard_deviation" : 0.005328406973561699
    },
    "weighted_f1" : {
        "value" : 0.6571162346664904,
        "standard_deviation" : 0.004385008075019733
    }
}
```
Model Quality CloudWatch Metrics

If you set the value of the enable_cloudwatch_metrics to True when you create the monitoring schedule, model quality monitoring jobs send all metrics to Amazon CloudWatch.

Model quality metrics appear in the `aws/sagemaker/Endpoints/model-metrics` namespace. For a list of the metrics that are emitted, see Model Quality Metrics (p. 1184).

You can use CloudWatch metrics to create an alarm when a specific metric doesn't meet the threshold you specify. For instructions about how to create CloudWatch alarms, see Create a CloudWatch Alarm Based on a Static Threshold in the Amazon CloudWatch User Guide.

Monitor Bias Drift for Models in Production

Amazon SageMaker Clarify bias monitoring helps data scientists and ML engineers monitor predictions for bias on a regular basis. As the model is monitored, customers can view exportable reports and graphs detailing bias in SageMaker Studio and configure alerts in Amazon CloudWatch to receive notifications if bias beyond a certain threshold is detected. Bias can be introduced or exacerbated in deployed ML models when the training data differs from the data that the model sees during deployment (that is, the live data). These kinds of changes in the live data distribution might be temporary (for example, due to some short-lived, real-world events) or permanent. In either case, it might be important to detect these changes. For example, the outputs of a model for predicting home prices can become biased if the mortgage rates used to train the model differ from current, real-world mortgage rates. With bias detection capabilities in Model Monitor, when SageMaker detects bias beyond a certain threshold, it automatically generates metrics that you can view in SageMaker Studio and through Amazon CloudWatch alerts.

In general, measuring bias only during the train-and-deploy phase might not be sufficient. It is possible that after the model has been deployed, the distribution of the data that the deployed model sees (that
is, the live data) is different from data distribution in the training dataset. This change might introduce bias in a model over time. The change in the live data distribution might be temporary (for example, due to some short-lived behavior like the holiday season) or permanent. In either case, it might be important to detect these changes and take steps to reduce the bias when appropriate.

To detect these changes, SageMaker Clarify provides functionality to monitor the bias metrics of a deployed model continuously and raise automated alerts if the metrics exceed a threshold. For example, consider the DPPL bias metric. Specify an allowed range of values $\mathbf{A} = (a_{\text{min}}, a_{\text{max}})$, for instance an interval of $(-0.1, 0.1)$, that DPPL should belong to during deployment. Any deviation from this range should raise a bias detected alert. With SageMaker Clarify, you can perform these checks at regular intervals.

For example, you can set the frequency of the checks to 2 days. This means that SageMaker Clarify computes the DPPL metric on data collected during a 2-day window. In this example, $D_{\text{win}}$ is the data that the model processed during last 2-day window. An alert is issued if the DPPL value $b_{\text{win}}$ computed on $D_{\text{win}}$ falls outside of an allowed range $\mathbf{A}$. This approach to checking if $b_{\text{win}}$ is outside of $\mathbf{A}$ can be somewhat noisy. $D_{\text{win}}$ might consist of very few samples and might not be representative of the live data distribution. The small sample size means that the value of bias $b_{\text{win}}$ computed over $D_{\text{win}}$ might not be a very robust estimate. In fact, very high (or low) values of $b_{\text{win}}$ may be observed purely due to chance. To ensure that the conclusions drawn from the observed data $D_{\text{win}}$ are statistically significant, SageMaker Clarify makes use of confidence intervals. Specifically, it uses the Normal Bootstrap Interval method to construct an interval $\mathbf{C} = (c_{\text{min}}, c_{\text{max}})$ such that SageMaker Clarify is confident that the true bias value computed over the full live data is contained in $\mathbf{C}$ with high probability. Now, if the confidence interval $\mathbf{C}$ overlaps with the allowed range $\mathbf{A}$, SageMaker Clarify interprets it as “it is likely that the bias metric value of the live data distribution falls within the allowed range”. If $\mathbf{C}$ and $\mathbf{A}$ are disjoint, SageMaker Clarify is confident that the bias metric does not lie in $\mathbf{A}$ and raises an alert.

**Model Monitor Sample Notebook**

Amazon SageMaker Clarify provides the following sample notebook that shows how to capture real-time inference data, create a baseline to monitor evolving bias against, and inspect the results:

- **Monitoring bias drift and feature attribution drift** Amazon SageMaker Clarify – Use Amazon SageMaker Model Monitor to monitor bias drift and feature attribution drift over time.

This notebook has been verified to run in Amazon SageMaker Studio only. If you need instructions on how to open a notebook in Amazon SageMaker Studio, see Create or Open an Amazon SageMaker Studio Notebook (p. 87). If you’re prompted to choose a kernel, choose Python 3 (Data Science). The following topics contain the highlights from the last two steps, and they contain code examples from the example notebook.

**Topics**

- Create a Bias Drift Baseline (p. 1188)
- Schedule Feature Attribute Drift Monitoring Jobs (p. 1190)
- Inspect Reports for Data Bias Drift (p. 1190)

**Create a Bias Drift Baseline**

After you have configured your application to capture real-time inference data, the first task to monitor for bias drift is to create a baseline. This involves configuring the data inputs, which groups are sensitive, how the predictions are captured, and the model and its posttraining bias metrics. Then you need to start the baselining job.

Model bias monitor can detect bias drift of ML models on a regular basis. Similar to the other monitoring types, the standard procedure of creating a model bias monitor is first baselining and then establishing a monitoring schedule.
```python
define ModelBiasMonitor:
    role=role,
    sagemaker_session=sagemaker_session,
    max_runtime_in_seconds=1800,
)
```

DataConfig stores information about the dataset to be analyzed (for example, the dataset file), its format (that is, CSV or JSONLines), headers (if any) and label.

```python
model_bias_baselining_job_result_uri = f"{baseline_results_uri}/model_bias"
model_bias_data_config = DataConfig(
    s3_data_input_path=validation_dataset,
    s3_output_path=model_bias_baselining_job_result_uri,
    label=label_header,
    headers=all_headers,
    dataset_type=dataset_type,
)
```

BiasConfig is the configuration of the sensitive groups in the dataset. Typically, bias is measured by computing a metric and comparing it across groups. The group of interest is called the facet. For posttraining bias, you should also take the positive label into account.

```python
model_bias_config = BiasConfig(
    label_values_or_threshold=[1],
    facet_name="Account Length",
    facet_values_or_threshold=[100],
)
```

ModelPredictedLabelConfig specifies how to extract a predicted label from the model output. In this example, the 0.8 cutoff has been chosen in anticipation that customers will turn over frequently. For more complicated outputs, there are a few more options, like "label" is the index, name, or JSONPath to locate predicted label in endpoint response payload.

```python
model_predicted_label_config = ModelPredictedLabelConfig(
    probability_threshold=0.8,
)
```

ModelConfig is the configuration related to the model to be used for inferencing. In order to compute posttraining bias metrics, the computation needs to get inferences for the model name provided. To accomplish this, the processing job uses the model to create an ephemeral endpoint (also known as shadow endpoint). The processing job deletes the shadow endpoint after the computations are completed. This configuration is also used by the explainability monitor.

```python
model_config = ModelConfig(
    model_name=model_name,
    instance_count=endpoint_instance_count,
    instance_type=endpoint_instance_type,
    content_type=dataset_type,
    accept_type=dataset_type,
)
```

Now you can start the baselining job.

```python
model_bias_monitor.suggest_baseline(
    model_config=model_config,
)```
Monitor Bias Drift

The scheduled monitor automatically picks up baselining job name and waits for it before monitoring begins.

Schedule Feature Attribute Drift Monitoring Jobs

Now that you have a baseline, you can call the `create_monitoring_schedule()` method to schedule an hourly monitor to analyze the data with a monitoring schedule. If you have submitted a baselining job, the monitor automatically picks up analysis configuration from the baselining job. If you skip the baselining step or the capture dataset has a different nature from the training dataset, you must provide the analysis configuration.

```python
model_bias_analysis_config = None
if not model_bias_monitor.latest_baselining_job:
    model_bias_analysis_config = BiasAnalysisConfig(
        model_bias_config,
        headers=all_headers,
        label=label_header,
    )
model_bias_monitor.create_monitoring_schedule(
    analysis_config=model_bias_analysis_config,
    output_s3_uri=s3_report_path,
    endpoint_input=EndpointInput(
        endpoint_name=endpoint_name,
        destination="/opt/ml/processing/input/endpoint",
        start_time_offset="-PT1H",
        end_time_offset="-PT0H",
        probability_threshold_attribute=0.8,
    ),
    ground_truth_input=ground_truth_upload_path,
    schedule_cron_expression=schedule_expression,
)
```

Inspect Reports for Data Bias Drift

If you are not able to inspect the results of the monitoring in the generated reports in SageMaker Studio, you can print them out as follows:

```python
schedule_desc = model_bias_monitor.describe_schedule()
execution_summary = schedule_desc.get("LastMonitoringExecutionSummary")
if execution_summary and execution_summary["MonitoringExecutionStatus"] in ["Completed",
    "CompletedWithViolations"]:
    last_model_bias_monitor_execution = model_bias_monitor.list_executions()[\-1]
l_model_bias_monitor_execution_report_uri = last_model_bias_monitor_execution.output.destination
    print(f'Report URI: {last_model_bias_monitor_execution_report_uri}')
    last_model_bias_monitor_execution_report_files = sorted(S3Downloader.list(last_model_bias_monitor_execution_report_uri))
    print("Found Report Files:")
    print("\n ".join(last_model_bias_monitor_execution_report_files))
else:
    last_model_bias_monitor_execution = None
    print("****STOP****  \nNo completed executions to inspect further. Please wait till an execution completes or investigate previously reported failures.")
```
If there are violations compared to the baseline, they are listed here:

```python
if last_model_bias_monitor_execution:
    model_bias_violations = last_model_bias_monitor_execution.constraint_violations()
    if model_bias_violations:
        print(model_bias_violations.body_dict)
```

In SageMaker Studio, you can see visualizations of the analysis results and CloudWatch metrics by choosing the **Endpoints** tab, and then double-clicking the endpoint.

### Monitor Feature Attribution Drift for Models in Production

A drift in the distribution of live data for models in production can result in a corresponding drift in the feature attribution values, just as it could cause a drift in bias when monitoring bias metrics. Amazon SageMaker Clarify feature attribution monitoring helps data scientists and ML engineers monitor predictions for feature attribution drift on a regular basis. As the model is monitored, customers can view exportable reports and graphs detailing feature attributions in SageMaker Studio and configure alerts in Amazon CloudWatch to receive notifications if it is detected that the attribution values drift beyond a certain threshold.

To illustrate this with a specific situation, consider a hypothetical scenario for college admissions. Assume that we observe the following (aggregated) feature attribution values in the training data and in the live data:

**College Admission Hypothetical Scenario**

<table>
<thead>
<tr>
<th>Feature</th>
<th>Attribution in training data</th>
<th>Attribution in live data</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAT score</td>
<td>0.70</td>
<td>0.10</td>
</tr>
<tr>
<td>GPA</td>
<td>0.50</td>
<td>0.20</td>
</tr>
<tr>
<td>Class rank</td>
<td>0.05</td>
<td>0.70</td>
</tr>
</tbody>
</table>

The change from training data to live data appears significant. The feature ranking has completely reversed. Similar to the bias drift, the feature attribution drifts might be caused by a change in the live data distribution and warrant a closer look into the model behavior on the live data. Again, the first step in these scenarios is to raise an alarm that a drift has happened.

We can detect the drift by comparing how the ranking of the individual features changed from training data to live data. In addition to being sensitive to changes in ranking order, we also want to be sensitive to the raw attribution score of the features. For instance, given two features that fall in the ranking by the same number of positions going from training to live data, we want to be more sensitive to the feature that had a higher attribution score in the training data. With these properties in mind, we use the Normalized Discounted Cumulative Gain (NDCG) score for comparing the feature attributions rankings of training and live data.

Specifically, assume we have the following:

- \( F = [f_1, \ldots, f_m] \) is the list of features sorted with respect to their attribution scores in the training data where \( m \) is the total number of features. For instance, in our case, \( F = [\text{SAT Score}, \text{GPA}, \text{Class Rank}] \).
- \( a(f) \) is a function that returns the feature attribution score on the training data given a feature \( f \). For example, \( a(\text{SAT Score}) = 0.70 \).
- \( F' = [f'_1, \ldots, f'_m] \) is the list of features sorted with respect to their attribution scores in the live data. For example, \( F' = [\text{Class Rank}, \text{GPA}, \text{SAT Score}] \).
Then, we can compute the NDCG as:

\[
\text{NDCG} = \frac{\text{DCG}}{\text{iDCG}}
\]

with

- \( \text{DCG} = \sum_{i=1}^{m} a(f_i)/\log_2(i+1) \)
- \( \text{iDCG} = \sum_{i=1}^{m} a(f_i)/\log_2(i+1) \)

The quantity DCG measures whether features with high attribution in the training data are also ranked higher in the feature attribution computed on the live data. The quantity iDCG measures the ideal score and it’s just a normalizing factor to ensure that the final quantity resides in the range \([0, 1]\), with 1 being the best possible value. A NDCG value of 1 means that the feature attribution ranking in the live data is the same as the one in the training data. In this particular example, because the ranking changed by quite a bit, the NDCG value is 0.69.

In SageMaker Clarify, if the NDCG value is below 0.90, we automatically raise an alert.

**Model Monitor Example Notebook**

SageMaker Clarify provides the following example notebook that shows how to capture real-time inference data, create a baseline to monitor evolving bias against, and inspect the results:

- Monitoring bias drift and feature attribution drift Amazon SageMaker Clarify – Use Amazon SageMaker Model Monitor to monitor bias drift and feature attribution drift over time.

This notebook has been verified to run in SageMaker Studio only. If you need instructions on how to open a notebook in SageMaker Studio, see Create or Open an Amazon SageMaker Studio Notebook (p. 87). If you’re prompted to choose a kernel, choose Python 3 (Data Science). The following topics contain the highlights from the last two steps, and they contain code examples from the example notebook.

**Topics**

- Create a SHAP Baseline for Models in Production (p. 1192)
- Schedule Feature Attribute Drift Monitoring Jobs (p. 1194)
- Inspect Reports for Feature Attribute Drift in Production Models (p. 1194)

**Create a SHAP Baseline for Models in Production**

Explanations are typically contrastive, that is, they account for deviations from a baseline. For information on explainability baselines, see SHAP Baselines for Explainability (p. 1117).

In addition to providing explanations for per-instance inferences, SageMaker Clarify also supports global explanation for ML models that helps you understand the behavior of a model as a whole in terms of its features. SageMaker Clarify generates a global explanation of an ML model by aggregating the Shapley values over multiple instances. SageMaker Clarify supports the following different ways of aggregation, which you can use to define baselines:

- \( \text{mean}_\text{abs} \) – Mean of absolute SHAP values for all instances.
- \( \text{median} \) – Median of SHAP values for all instances.
- \( \text{mean}_\text{sq} \) – Mean of squared SHAP values for all instances.

After you have configured your application to capture real-time inference data, the first task to monitor for drift in feature attribution is to create a baseline to compare against. This involves configuring
the data inputs, which groups are sensitive, how the predictions are captured, and the model and its posttraining bias metrics. Then you need to start the baselining job. Model explainability monitor can explain the predictions of a deployed model that's producing inferences and detect feature attribution drift on a regular basis.

```python
model_explainability_monitor = ModelExplainabilityMonitor(
    role=role,
    sagemaker_session=sagemaker_session,
    max_runtime_in_seconds=1800,
)
```

In this example, the explainability baselining job shares the test dataset with the bias baselining job, so it uses the same `DataConfig`, and the only difference is the job output URI.

```python
model_explainability_baselining_job_result_uri = f"{baseline_results_uri}/model_explainability"
model_explainability_data_config = DataConfig(
    s3_data_input_path=validation_dataset,
    s3_output_path=model_explainability_baselining_job_result_uri,
    label=label_header,
    headers=all_headers,
    dataset_type=dataset_type,
)
```

Currently the SageMaker Clarify explainer offers a scalable and efficient implementation of SHAP, so the explainability config is `SHAPConfig`, including the following:

- **baseline** – A list of rows (at least one) or S3 object URI to be used as the baseline dataset in the Kernel SHAP algorithm. The format should be the same as the dataset format. Each row should contain only the feature columns/values and omit the label column/values.
- **num_samples** – Number of samples to be used in the Kernel SHAP algorithm. This number determines the size of the generated synthetic dataset to compute the SHAP values.
- **agg_method** – Aggregation method for global SHAP values. Following are valid values:
  - **mean_abs** – Mean of absolute SHAP values for all instances.
  - **median** – Median of SHAP values for all instances.
  - **mean_sq** – Mean of squared SHAP values for all instances.
- **use_logit** – Indicator of whether the logit function is to be applied to the model predictions. Default is `False`. If `use_logit` is `True`, the SHAP values will have log-odds units.
- **save_local_shap_values** (bool) – Indicator of whether to save the local SHAP values in the output location. Default is `True`.

```python
# Here use the mean value of test dataset as SHAP baseline
test_dataframe = pd.read_csv(test_dataset, header=None)
shap_baseline = [list(test_dataframe.mean())]
shap_config = SHAPConfig(
    baseline=shap_baseline,
    num_samples=100,
    agg_method="mean_abs",
    save_local_shap_values=False,
)
```

Start a baselining job. The same `model_config` is required because the explainability baselining job needs to create a shadow endpoint to get predictions for the generated synthetic dataset.
model_explainability_monitor.suggest_baseline(
    data_config=model_explainability_data_config,
    model_config=model_config,
    explainability_config=shap_config,
)
print(f"ModelExplainabilityMonitor baselining job:
{model_explainability_monitor.latest_baselining_job_name}")

Schedule Feature Attribute Drift Monitoring Jobs

Model explainability monitoring helps you understand and interpret the predictions made by your ML models. When Model Monitor is configured to monitor model explainability, SageMaker automatically detects any drift in relative importance of features and creates reports explaining feature attributions.

Call the create_monitoring_schedule() method to schedule an hourly monitor to analyze the data with a monitoring schedule. If a baselining job has been submitted, the monitor automatically picks up analysis configuration from the baselining job. However, if you skip the baselining step or the capture dataset has a different nature from the training dataset, you have to provide the analysis configuration. ModelConfig is required by ExplainabilityAnalysisConfig for the same reason that it's required for the baselining job. Note that only features are required for computing feature attribution, so you should exclude ground truth labeling.

Inspect Reports for Feature Attribute Drift in Production Models

After the schedule that you set up is started by default, you need to wait for the its first execution to start, and then stop the schedule to avoid incurring charges.

To inspect the reports, use the following code:

```python
schedule_desc = model_explainability_monitor.describe_schedule()
execution_summary = schedule_desc.get("LastMonitoringExecutionSummary")
if execution_summary and execution_summary["MonitoringExecutionStatus"] in ["Completed", "CompletedWithViolations"]:
    last_model_explainability_monitor_execution =
last_model_explainability_monitor_execution_report_uri =
last_model_explainability_monitor_execution.output.destination
print(f'Report URI: {last_model_explainability_monitor_execution_report_uri}')
last_model_explainability_monitor_execution_report_files =
sorted(S3Downloader.list(last_model_explainability_monitor_execution_report_uri))
print("Found Report Files:")
print("\n ".join(last_model_explainability_monitor_execution_report_files))
else:
    last_model_explainability_monitor_execution = None
    print("****STOP**** \n No completed executions to inspect further. Please wait till an execution completes or investigate previously reported failures.")
```

If there are any violations compared to the baseline, they are listed here:

```python
if last_model_explainability_monitor_execution:
    model_explainability_violations =
last_model_explainability_monitor_execution.constraint_violations()
if model_explainability_violations:
    print(model_explainability_violations.body_dict)
```

In SageMaker Studio, you can see visualizations of the analysis results and CloudWatch metrics by choosing the Endpoints tab, and then double-clicking the endpoint.
Capture Data

To use Model Monitor, configure your real-time inference endpoint to capture data from requests and responses and store the captured data in Amazon S3. Model monitor compares metrics from this data with a baseline that you create for the model. For information about creating a baseline, see Create a Baseline (p. 1176).

To set up data capture

1. First, configure the Amazon S3 buckets Model Monitor uses to store the captured data.

   ```python
   import boto3
   import re
   import json
   from sagemaker import get_execution_role, session

   region = boto3.Session().region_name
   role = get_execution_role()
   print("RoleArn: {}".format(role))

   # You can use a different bucket, but make sure the role you chose for this notebook has s3:PutObject permissions. This is the bucket into which the data is captured
   bucket = session.Session(boto3.Session()).default_bucket()
   print("Demo Bucket: {}".format(bucket))
   prefix = 'sagemaker/DEMO-ModelMonitor'
   data_capture_prefix = '{}/datacapture'.format(prefix)
   s3_capture_upload_path = 's3://{}/{}'.format(bucket, data_capture_prefix)
   reports_prefix = '{}/reports'.format(prefix)
   s3_report_path = 's3://{}/{}'.format(bucket, reports_prefix)
   code_prefix = '{}/code'.format(prefix)
   s3_code_preprocessor_uri = 's3://{}/{}/preprocessor.py'.format(bucket, code_prefix)
   s3_code_postprocessor_uri = 's3://{}/{}/postprocessor.py'.format(bucket, code_prefix)

   print("Capture path: {}".format(s3_capture_upload_path))
   print("Report path: {}".format(s3_report_path))
   print("Preproc Code path: {}".format(s3_code_preprocessor_uri))
   print("Postproc Code path: {}".format(s3_code_postprocessor_uri))
   ```

2. Next, upload the pre-trained model to Amazon S3.

   ```python
   model_file = open("model/your-prediction-model.tar.gz", 'rb')
   s3_key = os.path.join(prefix, 'your-prediction-model.tar.gz')
   boto3.Session().resource('s3').Bucket(bucket).Object(s3_key).upload_fileobj(model_file)
   ```

3. Configure the data you want to capture by configuring the data you want to capture in a `DataCaptureConfig` structure. You can capture the request payload, the response payload, or both with this configuration.

   ```python
   from sagemaker.model_monitor import DataCaptureConfig

   endpoint_name = 'your-pred-model-monitor-' + strftime("%Y-%m-%d-%H-%M-%S", gmtime())
   print("EndpointName:{}".format(endpoint_name))

   data_capture_config = DataCaptureConfig(
       enable_capture=True,
       sampling_percentage=100,
       destination_s3_uri=s3_capture_upload_path)
   ```
4. Enable data capture by passing the configuration you created in the previous step when you deploy the model.

```python
predictor = model.deploy(initial_instance_count=1,
                          instance_type='ml.m4.xlarge',
                          endpoint_name=endpoint_name,
                          data_capture_config=data_capture_config)
```

5. Invoke the endpoint to send data to the endpoint to get inferences in real time. Because you enabled the data capture in the previous steps, the request and response payload, along with some additional metadata, is saved in the Amazon S3 location that you specified in `DataCaptureConfig`.

```python
from sagemaker.predictor import RealTimePredictor
import time

predictor = RealTimePredictor(endpoint=endpoint_name, content_type='text/csv')

# get a subset of test data for a quick test
!head -120 test_data/test-dataset-input-cols.csv > test_data/test_sample.csv
print("Sending test traffic to the endpoint {}. Please wait...".format(endpoint_name))

with open('test_data/test_sample.csv', 'r') as f:
    for row in f:
        payload = row.rstrip('
')
        response = predictor.predict(data=payload)
        time.sleep(0.5)
print("Done!")
```

6. View captured data by listing the data capture files stored in Amazon S3. Expect to see different files from different time periods, organized based on the hour when the invocation occurred.

```python
s3_client = boto3.Session().client('s3')
current_endpoint_capture_prefix = '{}/{}'.format(data_capture_prefix, endpoint_name)
result = s3_client.list_objects(Bucket=bucket, Prefix=current_endpoint_capture_prefix)
capture_files = [capture_file.get("Key") for capture_file in result.get('Contents')]

print("Found Capture Files:")
print(" 
 ".join(capture_files))
```

The format of the Amazon S3 path is as follows:

```
s3://{destination-bucket-prefix}/{endpoint-name}/{variant-name}/yyyy/mm/dd/hh/filename.jsonl
```

### Schedule Monitoring Jobs

Amazon SageMaker Model Monitor provides you the ability to continuously monitor the data collected from the endpoints on a schedule. You can create a monitoring schedule with the `CreateMonitoringSchedule` API with a predefined periodic interval. For example, every `x` hours (`x` can range from 1 to 23).

With a monitoring schedule, SageMaker can kick off processing jobs at a specified frequency to analyze the data collected during a given period. SageMaker provides a prebuilt container for performing analysis on tabular datasets. In the processing job, SageMaker compares the dataset for the current analysis with the baseline statistics, constraints provided and generate a violations report. In addition,
CloudWatch metrics are emitted for each feature under analysis. Alternatively, you could choose to bring your own container as outlined in the Bring Your Own Containers (p. 1210) topic.

You can create a model monitoring schedule for the endpoint created earlier. Use the baseline resources (constraints and statistics) to compare against the real-time traffic. For this example, upload the training dataset that was used to train the pretrained model included in this example. If you already have it in Amazon S3, you can point to it directly.

```python
# copy over the training dataset to Amazon S3 (if you already have it in Amazon S3, you could reuse it)
baseline_prefix = prefix + '/baselining'
baseline_data_prefix = baseline_prefix + '/data'
baseline_results_prefix = baseline_prefix + '/results'

baseline_data_uri = 's3://{}/{}'.format(bucket, baseline_data_prefix)
baseline_results_uri = 's3://{}/{}'.format(bucket, baseline_results_prefix)
print('Baseline data uri: {}'.format(baseline_data_uri))
print('Baseline results uri: {}'.format(baseline_results_uri))

training_data_file = open("test_data/training-dataset-with-header.csv", 'rb')
s3_key = os.path.join(baseline_prefix, 'data', 'training-dataset-with-header.csv')
boto3.Session().resource('s3').Bucket(bucket).Object(s3_key).upload_fileobj(training_data_file)
```

Create a model monitoring schedule for the endpoint using the baseline constraints and statistics to compare against real-time traffic.

```python
from sagemaker.model_monitor import CronExpressionGenerator
from time import gmtime, strftime

mon_schedule_name = 'DEMO-xgb-churn-pred-model-monitor-schedule-' + strftime("%Y-%m-%d-%H-%M-%S", gmtime())
my_default_monitor.create_monitoring_schedule(
    schedule_name=mon_schedule_name,
    endpoint_input=predictor.endpoint,
    post_analytics_processor_script=s3_code_postprocessor_uri,
    output_s3_uri=s3_report_path,
    statistics=my_default_monitor.baseline_statistics(),
    constraints=my_default_monitor.suggested_constraints(),
    schedule_cron_expression=CronExpressionGenerator.hourly(),
    enable_cloudwatch_metrics=True,
)
```

Describe and inspect the schedule: After you describe it, observe that the MonitoringScheduleStatus in MonitoringScheduleSummary returned by the ListMonitoringSchedules API changes to Scheduled.

```python
desc_schedule_result = my_default_monitor.describe_schedule()
print('Schedule status: {}'.format(desc_schedule_result['MonitoringScheduleStatus']))
```

### The cron Expression for Monitoring Schedule

To provide details for the monitoring schedule, use ScheduleConfig, which is a cron expression that describes details about the monitoring schedule.

Amazon SageMaker Model Monitor supports the following cron expressions:

- To set the job to start every hour, use the following:
  
  Hourly: `cron(0 * ? * *)`
To run the job daily, use the following:

cron(0 [00-23] ? * * *)

For example, the following are valid cron expressions:

- Daily at 12 PM UTC: cron(0 12 ? * * *)
- Daily at 12 AM UTC: cron(0 0 ? * * *)

To support running every 6, 12 hours, Model Monitor supports the following expression:

cron(0 [00-23]/[01-24] ? * * *)

For example, the following are valid cron expressions:

- Every 12 hours, starting at 5 PM UTC: cron(0 17/12 ? * * *)
- Every two hours, starting at 12 AM UTC: cron(0 0/2 ? * * *)

Notes

- Although the cron expression is set to start at 5 PM UTC, note that there could be a delay of 0-20 minutes from the actual requested time to run the execution.
- If you want to run on a daily schedule, don’t provide this parameter. SageMaker picks a time to run every day.
- Currently, SageMaker only supports hourly integer rates between 1 hour and 24 hours.

Amazon SageMaker Model Monitor Prebuilt Container

SageMaker provides a built-in container sagemaker-model-monitor-analyzer that provides you with a range of model monitoring capabilities, including constraint suggestion, statistics generation, constraint validation against a baseline, and emitting Amazon CloudWatch metrics. This container is based on Spark and is built with Deequ. The prebuilt container for SageMaker Model Monitor can be accessed as follows:

```
<ACCOUNT_ID>.dkr.ecr.<REGION_NAME>.amazonaws.com/sagemaker-model-monitor-analyzer
```

For example: 159807026194.dkr.ecr.us-west-2.amazonaws.com/sagemaker-model-monitor-analyzer

The following table lists the supported values for account IDs and corresponding AWS Region names.

<table>
<thead>
<tr>
<th>ACCOUNT_ID</th>
<th>REGION_NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>156813124566</td>
<td>us-east-1</td>
</tr>
<tr>
<td>777275614652</td>
<td>us-east-2</td>
</tr>
<tr>
<td>890145073186</td>
<td>us-west-1</td>
</tr>
<tr>
<td>159807026194</td>
<td>us-west-2</td>
</tr>
<tr>
<td>875698925577</td>
<td>af-south-1</td>
</tr>
</tbody>
</table>
To write your own analysis container, see the container contract described in Customize Monitoring (p. 1207).

### Interpret Results

After you run a baseline processing job and obtained statistics and constraint for your dataset, you can execute monitoring jobs that calculate statistics and list any violations encountered relative to the baseline constraints. Amazon CloudWatch metrics are also reported in your account by default. For information on viewing the results of monitoring in Amazon SageMaker Studio, see Visualize Results in Amazon SageMaker Studio (p. 1201).

### List Executions

The schedule starts monitoring jobs at the specified intervals. The following code lists the latest five executions. If you are running this code after creating the hourly schedule, the executions might be empty, and you might have to wait until you cross the hour boundary (in UTC) to see the executions start. The following code includes the logic for waiting.

```python
mon_executions = my_default_monitor.list_executions()
print("We created a hourly schedule above and it will kick off executions ON the hour (plus 0 - 20 min buffer.\nWe will have to wait till we hit the hour...")
```
Interpret Results

while len(mon_executions) == 0:
    print("Waiting for the 1st execution to happen...")
    time.sleep(60)
    mon_executions = my_default_monitor.list_executions()

Inspect a Specific Execution

In the previous step, you picked up the latest completed or failed scheduled execution. You can explore what went right or wrong. The terminal states are:

- **Completed** – The monitoring execution completed and no issues were found in the violations report.
- **CompletedWithViolations** – The execution completed, but constraint violations were detected.
- **Failed** – The monitoring execution failed, possibly due to client error (for example, a role issues) or infrastructure issues. To identify the cause, see the `FailureReason` and `ExitMessage`.

```
latest_execution = mon_executions[-1] # latest execution's index is -1, previous is -2 and so on..
time.sleep(60)
latest_execution.wait(logs=False)
print("Latest execution status: {}").format(latest_execution.describe()['ProcessingJobStatus'])
print("Latest execution result: {}").format(latest_execution.describe()]['ExitMessage'])
```

```
report_uri=latest_execution.output.destination
print('Report Uri: {}'.format(report_uri))
```

List Generated Reports

List the generated reports. Use the following code to list the generated reports.

```
report_uri=latest_execution.output.destination
print('Report Uri: {}'.format(report_uri))
```

```
from urlparse import urlparse
s3uri = urlparse(report_uri)
report_bucket = s3uri.netloc
report_key = s3uri.path.lstrip('/
print('Report bucket: {}').format(report_bucket))
print('Report key: {}').format(report_key))
```

```
s3_client = boto3.Session().client('s3')
result = s3_client.list_objects(Bucket=report_bucket, Prefix=report_key)
report_files = [report_file.get("Key") for report_file in result.get("Contents")]
print("Found Report Files:")
print("\n ",join(report_files))
```

Violations Report

If there are violations compared to the baseline, they are generated in the violations report. Use the following code to list the violations.

```
```
violations = my_default_monitor.latest_monitoring_constraint_violations()
pd.set_option('display.max_colwidth', -1)
constraints_df = pd.io.json.json_normalize(violations.body_dict['violations'])
constraints_df.head(10)

This applies only to datasets that contain tabular data. The following schema files specify the statistics calculated and the violations monitored for.

### Output Files for Tabular Datasets

<table>
<thead>
<tr>
<th>File Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>statistics.json</td>
<td>Contains columnar statistics for each feature in the dataset that is analyzed. See the schema of this file in the next topic.</td>
</tr>
<tr>
<td></td>
<td><strong>Note</strong></td>
</tr>
<tr>
<td></td>
<td>This file is created only for data quality monitoring.</td>
</tr>
<tr>
<td>constraints_violations.json</td>
<td>Contains a list of violations found in this current set of data as compared to the baseline statistics and constraints file specified in the baseline_constraints and baseline_statistics paths.</td>
</tr>
</tbody>
</table>

The Amazon SageMaker Model Monitor Prebuilt Container (p. 1198) saves a set of Amazon CloudWatch metrics for each feature by default.

The container code can emit CloudWatch metrics in this location: `/opt/ml/output/metrics/cloudwatch`.

### Visualize Results in Amazon SageMaker Studio

You can also visualize the results of monitoring in Amazon SageMaker Studio. You can view the details of any monitoring job run, and you can create charts that show the baseline and captured values for any metric that the monitoring job calculates.

**To view the detailed results of a monitoring job**

1. Sign in to Studio. For more information, see Onboard to Amazon SageMaker Studio (p. 34).
2. In the left navigation pane, choose the **Components and registries** icon (🔍).
3. Choose **Endpoints** in the drop-down menu.
4. On the endpoint tab, choose the monitoring type for which you want to see job details.

5. Choose the name of the monitoring job run for which you want to view details from the list of monitoring jobs.
6. The **MONITORING JOB DETAILS** tab opens with a detailed report of the monitoring job.

You can create a chart that displays the baseline and captured metrics for a time period.
To create a chart in SageMaker Studio to visualize monitoring results

1. Sign in to Studio. For more information, see Onboard to Amazon SageMaker Studio (p. 34).
2. In the left navigation pane, choose the Components and registries icon ( ).
3. Choose Endpoints in the drop-down menu.
4. On the Endpoint tab, choose the monitoring type you want to create a chart for. This example shows a chart for the Model quality monitoring type.
5. Choose Add chart.

6. On the **CHART PROPERTIES** tab, choose the time period, statistic, and metric that you want to chart. This example shows a chart for a **Timeline** of **1 week**, the **Average Statistic** of, and the **F1 Metric**.
7. The chart that shows the baseline and current metric statistic you chose in the previous step shows up in the **Endpoint** tab.
Advanced Topics

The following sections contain more advanced tasks that explain how to customize monitoring using preprocessing and postprocessing scripts, how to build your own container, and how to use AWS CloudFormation to create a monitoring schedule.

Topics
- Customize Monitoring (p. 1207)
- Create a Monitoring Schedule with an AWS CloudFormation Custom Resource (p. 1217)

Customize Monitoring

In addition to using the built-in monitoring mechanisms, you can create your own custom monitoring schedules and procedures using preprocessing and postprocessing scripts or by using or building your own container.

Topics
- Preprocessing and Postprocessing (p. 1207)
- Bring Your Own Containers (p. 1210)

Preprocessing and Postprocessing

In addition to using the built-in mechanisms, you can extend the code with the preprocessing and postprocessing scripts.

Topics
- Postprocessing Script (p. 1207)
- Preprocessing Script (p. 1208)

Postprocessing Script

You can extend the code with the postprocessing script by following this contract:
def postprocess_handler():
    print("Hello from post-proc script!")

Specify it as a path in Amazon Simple Storage Service (Amazon S3) in the CreateMonitoringSchedule request, as shown following:


Preprocessing Script

The Amazon SageMaker Model Monitor container works only with tabular or flattened JSON structures. We provide a per-record preprocessor for some small changes required to transform the dataset. For example, if your output is an array [1.0, 2.1], you need to convert this into a flattened JSON, like {"prediction0": 1.0, "prediction1": 2.1"}. A sample implementation might look like the following:

```python
def preprocess_handler(inference_record):
    input_data = inference_record.endpoint_input.data
    output_data = inference_record.endpoint_output.data

    input_data['feature0'] = random.randint(1, 3)
    input_data['feature1'] = random.uniform(0, 1.6)
    input_data['feature2'] = random.uniform(0, 1.6)

    output_data['prediction0'] = random.uniform(1, 30)

    return {**input_data, **output_data}
```

Specify it as a path in Amazon S3 in the CreateMonitoringSchedule request:


The structure of the inference_record is defined as follows:

```python
KEY_EVENT_METADATA = "eventMetadata"
KEY_EVENT_METADATA_EVENT_ID = "eventId"
KEY_EVENT_METADATA_EVENT_TIME = "inferenceTime"
KEY_EVENT_METADATA_CUSTOM_ATTR = "customAttributes"
KEY_EVENTDATA = "captureData"
KEY_EVENTDATA_INPUT = "endpointInput"
KEY_EVENTDATA_OUTPUT = "endpointOutput"
KEY_EVENTDATA_ENCODING = "encoding"
KEY_EVENTDATA_DATA = "data"
KEY_EVENTDATA_OBSERVED_CONTENT_TYPE = "observedContentType"
KEY_EVENTDATA_MODE = "mode"
KEY_EVENT_VERSION = "eventVersion"
"
```
class EventConfig:
    def __init__(self, endpoint, variant, start_time, end_time):
        self.endpoint = endpoint
        self.variant = variant
        self.start_time = start_time
        self.end_time = end_time

class EventMetadata:
    def __init__(self, event_metadata_dict):
        self.event_id = event_metadata_dict.get(KEY_EVENT_METADATA_EVENT_ID, None)
        self.event_time = event_metadata_dict.get(KEY_EVENT_METADATA_EVENT_TIME, None)
        self.custom_attribute = event_metadata_dict.get(KEY_EVENTDATA_OBSERVED_CONTENT_TYPE, None)

class EventData:
    def __init__(self, data_dict):
        self.encoding = data_dict.get(KEY_EVENTDATA_ENCODING, None)
        self.data = data_dict.get(KEY_EVENTDATA_DATA, None)
        self.observedContentType = data_dict.get(KEY_EVENTDATA_OBSERVED_CONTENT_TYPE, None)
        self.mode = data_dict.get(KEY_EVENTDATA_MODE, None)

    def as_dict(self):
        ret = {
            KEY_EVENTDATA_ENCODING: self.encoding,
            KEY_EVENTDATA_DATA: self.data,
            KEY_EVENTDATA_OBSERVED_CONTENT_TYPE: self.observedContentType,
        }
        return ret

class CapturedData:
    def __init__(self, event_dict):
        self.event_metadata = None
        self.endpoint_input = None
        self.endpoint_output = None
        self.event_version = None
        self.event_dict = event_dict
        self._event_dict_postprocessed = False

        if KEY_EVENT_METADATA in event_dict:
            self.event_metadata = EventMetadata(event_dict[KEY_EVENT_METADATA])
        if KEY_EVENTDATA in event_dict:
            if KEY_EVENTDATA_INPUT in event_dict[KEY_EVENTDATA]:
                self.endpoint_input = EventData(event_dict[KEY_EVENTDATA][KEY_EVENTDATA_INPUT])
            if KEY_EVENTDATA_OUTPUT in event_dict[KEY_EVENTDATA]:
                self.endpoint_output = EventData(event_dict[KEY_EVENTDATA][KEY_EVENTDATA_OUTPUT])
        if KEY_EVENT_VERSION in event_dict:
            self.event_version = event_dict[KEY_EVENT_VERSION]

    def as_dict(self):
if self._event_dict_postprocessed is True:
    return self.event_dict
if KEY_EVENTDATA in self.event_dict:
    if KEY_EVENTDATA_INPUT in self.event_dict[KEY_EVENTDATA]:
        self.event_dict[KEY_EVENTDATA][KEY_EVENTDATA_INPUT] =
        self.endpoint_input.as_dict()
    if KEY_EVENTDATA_OUTPUT in self.event_dict[KEY_EVENTDATA]:
        self.event_dict[KEY_EVENTDATA][
            KEY_EVENTDATA_OUTPUT
        ] = self.endpoint_output.as_dict()
self._event_dict_postprocessed = True
return self.event_dict

Bring Your Own Containers

Amazon SageMaker Model Monitor provides a prebuilt container with ability to analyze the data captured from endpoints for tabular datasets. If you would like to bring your own container, Model Monitor provides extension points which you can leverage.

Under the hood, when you create a MonitoringSchedule, Model Monitor ultimately kicks off processing jobs. Hence the container needs to be aware of the processing job contract documented in the Build Your Own Processing Container (Advanced Scenario) (p. 600) topic. Note that Model Monitor kicks off the processing job on your behalf per the schedule. While invoking, Model Monitor sets up additional environment variables for you so that your container has enough context to process the data for that particular execution of the scheduled monitoring. For additional information on container inputs, see the Container Contract Inputs (p. 1210).

In the container, using the above environment variables/context, you can now analyze the dataset for the current period in your custom code. After this analysis is complete, you can choose to emit your reports to be uploaded to an S3 bucket. The reports that the prebuilt container generates are documented in Container Contract Outputs (p. 1212). If you would like the visualization of the reports to work in SageMaker Studio, you should follow the same format. You can also choose to emit completely custom reports.

You also emit CloudWatch metrics from the container by following the instructions in CloudWatch Metrics for Bring Your Own Containers (p. 1216).

Topics

- Container Contract Inputs (p. 1210)
- Container Contract Outputs (p. 1212)
- CloudWatch Metrics for Bring Your Own Containers (p. 1216)

Container Contract Inputs

The Amazon SageMaker Model Monitor platform invokes your container code according to a specified schedule. If you chose to write your own container code, the following environment variables are available for your container code. In this context, you can analyze the current dataset or evaluate the constraints if you chose to and emit metrics, if applicable.

"Environment": {
    "dataset_format": "{"sagemakerCaptureJson": {"captureIndexNames": ["endpointInput", "endpointOutput"]}}",
    "dataset_source": "/opt/ml/processing/endpointdata",
    "end_time": "2019-12-01T16:00:00Z",
    "output_path": "/opt/ml/processing/resultdata",
    "publish_cloudwatch_metrics": "Disabled",
    "sagemaker_endpoint_name": "endpoint-name",
    "sagemaker_monitoring_schedule_name": "schedule-name",
    "start_time": "2019-12-01T15:20:00Z"}
### Parameters

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dataset_format</td>
<td>For a job started from a MonitoringSchedule backed by an Endpoint, this is sageMakerCaptureJson with the capture indices endpointInput, or endpointOutput, or both.</td>
</tr>
<tr>
<td>dataset_source</td>
<td>The local path in which the data corresponding to the monitoring period, as specified by start_time and end_time, are available. At this path, the data is available in /{endpoint-name}/variant-name/yyyy/mm/dd/hh.</td>
</tr>
<tr>
<td></td>
<td>We sometimes download more than what is specified by the start and end times. It is up to the container code to parse the data as required.</td>
</tr>
<tr>
<td>output_path</td>
<td>The local path to write output reports and other files. You specify this parameter in the CreateMonitoringSchedule request as MonitoringOutputConfig.MonitoringOutput[0].LocalPath. It is uploaded to the S3Uri path specified in MonitoringOutputConfig.MonitoringOutput[0].S3Uri.</td>
</tr>
<tr>
<td>publish_cloudwatch_metrics</td>
<td>For a job launched by CreateMonitoringSchedule, this parameter is set to Enabled. The container can choose to write the Amazon CloudWatch output file at [filepath].</td>
</tr>
<tr>
<td>sagemaker_endpoint_name</td>
<td>The name of the Endpoint that this scheduled job was launched for.</td>
</tr>
<tr>
<td>sagemaker_monitoring_schedule_name</td>
<td>The name of the MonitoringSchedule that launched this job.</td>
</tr>
<tr>
<td>sagemaker_endpoint_datacapture_prefix</td>
<td>The prefix specified in the DataCaptureConfig parameter of the Endpoint. The container can use this if it needs to directly access more data than already downloaded by SageMaker at the dataset_source path.</td>
</tr>
<tr>
<td>start_time, end_time</td>
<td>The time window for this analysis run. For example, for a job scheduled to run at 05:00 UTC and a job that runs on 20/02/2020, start_time: is 2020-02-19T06:00:00Z and end_time: is 2020-02-20T05:00:00Z</td>
</tr>
<tr>
<td>baseline_constraints:</td>
<td>The local path of the baseline constraint file specified in BaselineConfig.ConstraintResource.S3Uri. This is available only if this parameter was specified in the CreateMonitoringSchedule request.</td>
</tr>
</tbody>
</table>
Parameter Name | Description
---|---
baseline_statistics | The local path to the baseline statistics file specified in BaselineConfig.StatisticsResource.S3Uri. This is available only if this parameter was specified in the CreateMonitoringSchedule request.

Container Contract Outputs

The container can analyze the data available in the *dataset_source* path and write reports to the path in *output_path*. The container code can write any reports that suit your needs.

If you use the following structure and contract, certain output files are treated specially by SageMaker in the visualization and API. This applies only to tabular datasets.

Output Files for Tabular Datasets

<table>
<thead>
<tr>
<th>File Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>statistics.json</td>
<td>This file is expected to have columnar statistics for each feature in the dataset that is analyzed. The schema for this file is available in the next section.</td>
</tr>
<tr>
<td>constraints.json</td>
<td>This file is expected to have the constraints on the features observed. The schema for this file is available in the next section.</td>
</tr>
<tr>
<td>constraintsViolations.json</td>
<td>This file is expected to have the list of violations found in this current set of data as compared to the baseline statistics and constraints file specified in the baseline_constraints and baseline_statistics path.</td>
</tr>
</tbody>
</table>

In addition, if the publish_cloudwatch_metrics value is "Enabled" container code can emit Amazon CloudWatch metrics in this location: /opt/ml/output/metrics/cloudwatch. The schema for these files is described in the following sections.

Topics

- Schema for Statistics (statistics.json file) (p. 1212)
- Schema for Constraints (constraints.json file) (p. 1214)

Schema for Statistics (statistics.json file)

The schema defined in the statistics.json file specifies the statistical parameters to be calculated for the baseline and data that is captured. It also configures the bucket to be used by KLL, a very compact quantiles sketch with lazy compaction scheme.

```json
{
    "version": 0,
    # dataset level stats
    "dataset": {
        "item_count": number
    },
    # feature level stats
}
```
"features": [  
  {  
    "name": "feature-name",
    "inferred_type": "Fractional" | "Integral",
    "numerical_statistics": {  
      "common": {  
        "num_present": number,
        "num_missing": number
      },
      "mean": number,
      "sum": number,
      "std_dev": number,
      "min": number,
      "max": number,
      "distribution": {  
        "kll": {  
          "buckets": {  
            "lower_bound": number,
            "upper_bound": number,
            "count": number
          }
        },
        "sketch": {  
          "parameters": {  
            "c": number,
            "k": number
          },
          "data": [  
            [num, num, num, num],
            [num, num]
          ]
        }
      }
    }
  },
  {  
    "name": "feature-name",
    "inferred_type": "String",
    "string_statistics": {  
      "common": {  
        "num_present": number,
        "num_missing": number
      },
      "distinct_count": number,
      "distribution": {  
        "categorical": {  
          "buckets": [  
            {  
              "value": "string",
              "count": number
            }
          ]
        }
      }
    }
  }
]
Notes

- The specified metrics are recognized by SageMaker in later visualization changes. The container can emit more metrics if required.
- **KLL sketch** is the recognized sketch. Custom containers can write their own representation, but it won’t be recognized by SageMaker in visualizations.
- By default, the distribution is materialized in 10 buckets. You can’t change this.

Schema for Constraints (constraints.json file)

A constraints.json file is used to express the constraints that a dataset must satisfy. Amazon SageMaker Model Monitor containers can use the constraints.json file to evaluate datasets against. Prebuilt containers provide the ability to generate the constraints.json file automatically for a baseline dataset. If you bring your own container, you can provide it with similar abilities or you can create the constraints.json file in some other way. Here is the schema for the constraint file that the prebuilt container uses. Bring your own containers can adopt the same format or enhance it as required.

```json
{
    "version": 0,
    "features": [
        {
            "name": "string",
            "inferred_type": "Integral" | "Fractional" | "String" | "Unknown",
            "completeness": number, # denotes observed non-null value percentage
            "num_constraints": {
                "is_non_negative": boolean,
            },
            "string_constraints": {
                "domains": [
                    "list of",
                    "observed values",
                    "for small cardinality"
                ],
            },
            "monitoringConfigOverrides": {
            }
        }
    ]

    "monitoring_config": {
        "evaluate_constraints": "Enabled",
        "emit_metrics": "Enabled",
        "datatype_check_threshold": 1.0,
        "domain_content_threshold": 1.0,
        "distribution_constraints": {
            "perform_comparison": "Enabled",
            "comparison_threshold": 0.1,
            "comparison_method": "Simple"||"Robust"
        }
    }
}
```
## Monitoring Constraints

<table>
<thead>
<tr>
<th>Constraint</th>
<th>Description</th>
</tr>
</thead>
</table>
| `evaluate_constraints`      | When Enabled, evaluates whether the current dataset being analyzed satisfies the constraints specified in the constraints.json file taken as a baseline. Valid values: Enabled or Disabled
|                             | Default: Enabled                                                                                                                                                                                             |
| `emit_metrics`              | When Enabled, emits CloudWatch metrics for the data contained in the file. Valid values: Enabled or Disabled
|                             | Default: Enabled                                                                                                                                                                                             |
| `datatype_check_threshold`  | If the threshold is above the value of the specified `datatype_check_threshold`, this causes a failure that is treated as a violation in the violation report. If the data types in the current execution are not the same as in the baseline dataset, this threshold is used to evaluate if it needs to be flagged as a violation. During the baseline step, the generated constraints suggest the inferred data type for each column. The `datatype_check_threshold` parameter can be tuned to adjust the threshold on when it is flagged as a violation. Valid values: float
|                             | Default: 1.0                                                                                                                                                                                                |
| `domain_content_threshold`  | If there are more unknown values for a String field in the current dataset than in the baseline dataset, this threshold can be used to dictate if it needs to be flagged as a violation. Valid values: float
|                             | Default: 1.0                                                                                                                                                                                                |
| `distribution_constraints`  | perform_comparison
|                             | When Enabled, this flag instructs the code to perform a distribution comparison between the baseline distribution and the distribution observed for the current dataset. Valid values: Enabled or Disabled
|                             | Default: Enabled                                                                                                                                                                                             |
### Constraint Description

<table>
<thead>
<tr>
<th>Constraint</th>
<th>Description</th>
</tr>
</thead>
</table>
| comparison_threshold | If the threshold is above the value set for the comparison_threshold, this causes a failure that is treated as a violation in the violation report. The distance is calculated by getting the maximum absolute difference between the cumulative distribution functions of two distributions.  
Valid values: float  
Default: 1.0. |
| comparison_method  | Whether to calculate linf_simple or linf_robust. The linf_simple is based on the maximum absolute difference between the cumulative distribution functions of two distributions. Calculating linf_robust is based on linf_simple, but is used when there are not enough samples. The linf_robust formula is based on the Two-sample Kolmogorov–Smirnov test.  
Valid values: linf_simple or linf_robust. |

### CloudWatch Metrics for Bring Your Own Containers

If the publish_cloudwatch_metrics value is Enabled in the Environment map in the /opt/ml/processing/processingjobconfig.json file, the container code emits Amazon CloudWatch metrics in this location: /opt/ml/output/metrics/cloudwatch.

The schema for this file is closely based on the CloudWatch PutMetrics API. The namespace is not specified here. It defaults to /aws/sagemaker/Endpoint/data-metrics. However, you can specify dimensions. We recommend that you add the Endpoint and MonitoringSchedule dimensions at a minimum.

```json
{
  "MetricName": "",
  "Timestamp": "2019-11-26T03:00:00Z",
  "Dimensions": [{"Name":"Endpoint","Value":"endpoint_0"},
                  {"Name":"MonitoringSchedule","Value":"schedule_0"}]
  "Value": Float,
  # Either the Value or the StatisticValues field can be populated and not both.
  "StatisticValues": {
    "SampleCount": Float,
    "Sum": Float,
    "Minimum": Float,
    "Maximum": Float
  },
  "Unit": "Count", # Optional
}
```
Create a Monitoring Schedule with an AWS CloudFormation Custom Resource

To use AWS CloudFormation to create a monitoring schedule, use an AWS CloudFormation custom resource. The custom resource is in Python. To deploy it, see Python Lambda deployment.

Custom Resource

Start by adding a custom resource to your AWS CloudFormation template. This points to an AWS Lambda function that you create in the next step.

This resource enables you to customize the parameters for the monitoring schedule. You can add or remove more parameters by modifying the AWS CloudFormation resource and the Lambda function in the following example resource.

```json
{
  "AWSTemplateFormatVersion": "2010-09-09",
  "Resources": {
    "MonitoringSchedule": {
      "Type": "Custom::MonitoringSchedule",
      "Version": "1.0",
      "Properties": {
        "ScheduleName": "YourScheduleName",
        "EndpointName": "YourEndpointName",
        "BaselineConstraintsUri": "s3://your-baseline-constraints/constraints.json",
        "BaselineStatisticsUri": "s3://your-baseline-stats/statistics.json",
        "PostAnalyticsProcessorSourceUri": "s3://your-post-processor/postprocessor.py",
        "RecordPreprocessorSourceUri": "s3://your-preprocessor/preprocessor.py",
        "InputLocalPath": "/opt/ml/processing/endpointdata",
        "OutputLocalPath": "/opt/ml/processing/localpath",
        "OutputS3URI": "s3://your-output-uri",
        "ImageURI": "111111111111.dkr.ecr.us-west-2.amazonaws.com/your-image",
        "ScheduleExpression": "cron(0 * ? * * *)",
        "PassRoleArn": "arn:aws:iam::111111111111:role/AmazonSageMaker-ExecutionRole"
      }
    }
  }
}
```

Lambda Custom Resource Code

This AWS CloudFormation custom resource uses the Custom Resource Helper AWS library, which you can install with pip using `pip install crhelper`.

This Lambda function is invoked by AWS CloudFormation during the creation and deletion of the stack. This Lambda function is responsible for creating and deleting the monitoring schedule and using the parameters defined in the custom resource described in the preceding section.

```python
import boto3
import botocore
import logging
from crhelper import CfnResource
from botocore.exceptions import ClientError
```
logger = logging.getLogger(__name__)
sm = boto3.client('sagemaker')

# cfhelper makes it easier to implement a CloudFormation custom resource
helper = CfnResource()

# CFN Handlers

def handler(event, context):
    helper(event, context)

@helper.create
def create_handler(event, context):
    
    Called when CloudFormation custom resource sends the create event

    create_monitoring_schedule(event)

@helper.delete
def delete_handler(event, context):
    
    Called when CloudFormation custom resource sends the delete event

    schedule_name = get_schedule_name(event)
    delete_monitoring_schedule(schedule_name)

@helper.poll_create
def poll_create(event, context):
    
    Return true if the resource has been created and false otherwise so
    CloudFormation polls again.

    schedule_name = get_schedule_name(event)
    logger.info('Polling for creation of schedule: %s', schedule_name)
    return is_schedule_ready(schedule_name)

@helper.update
def noop():
    
    Not currently implemented but crhelper will throw an error if it isn't added

    pass

# Helper Functions

def get_schedule_name(event):
    return event['ResourceProperties']['ScheduleName']

def create_monitoring_schedule(event):
    schedule_name = get_schedule_name(event)
    monitoring_schedule_config = create_monitoring_schedule_config(event)

    logger.info('Creating monitoring schedule with name: %s', schedule_name)

    sm.create_monitoring_schedule(
        MonitoringScheduleName=schedule_name,
        MonitoringScheduleConfig=monitoring_schedule_config)

def is_schedule_ready(schedule_name):
    is_ready = False
schedule = sm.describe_monitoring_schedule(MonitoringScheduleName=schedule_name)
status = schedule['MonitoringScheduleStatus']

if status == 'Scheduled':
    logger.info('Monitoring schedule (%s) is ready', schedule_name)
    is_ready = True
elif status == 'Pending':
    logger.info('Monitoring schedule (%s) still creating, waiting and polling
    again...', schedule_name)
else:
    raise Exception('Monitoring schedule ({}) has unexpected status: {}'
'.format(schedule_name, status))

return is_ready

def create_monitoring_schedule_config(event):
    props = event['ResourceProperties']

    return {
        "ScheduleConfig": {
            "ScheduleExpression": props["ScheduleExpression"],
        },
        "MonitoringJobDefinition": {
            "BaselineConfig": {
                "ConstraintsResource": {
                    "S3Uri": props["BaselineConstraintsUri"],
                },
                "StatisticsResource": {
                    "S3Uri": props["BaselineStatisticsUri"],
                }
            },
            "MonitoringInputs": [
                {
                    "EndpointInput": {
                        "EndpointName": props["EndpointName"],
                        "LocalPath": props["InputLocalPath"],
                    }
                }
            ],
            "MonitoringOutputConfig": {
                "MonitoringOutputs": [
                    {
                        "S3Output": {
                            "S3Uri": props["OutputS3URI"],
                            "LocalPath": props["OutputLocalPath"],
                        }
                    }
                ],
            },
            "MonitoringResources": {
                "ClusterConfig": {
                    "InstanceCount": 1,
                    "InstanceType": "ml.t3.medium",
                    "VolumeSizeInGB": 50,
                }
            },
            "MonitoringAppSpecification": {
                "ImageUri": props["ImageURI"],
                "RecordPreprocessorSourceUri": props["PostAnalyticsProcessorSourceUri"],
                "PostAnalyticsProcessorSourceUri": props["PostAnalyticsProcessorSourceUri"],
            },
            "StoppingCondition": {
                "MaxRuntimeInSeconds": 300
            },
            "RoleArn": props["PassRoleArn"],
def delete_monitoring_schedule(schedule_name):
    logger.info('Deleting schedule: %s', schedule_name)
    try:
        sm.delete_monitoring_schedule(MonitoringScheduleName=schedule_name)
    except ClientError as e:
        if e.response['Error']['Code'] == 'ResourceNotFound':
            logger.info('Resource not found, nothing to delete')
        else:
            logger.error('Unexpected error while trying to delete monitoring schedule')
            raise e

Deploy an Inference Pipeline

An inference pipeline is a Amazon SageMaker model that is composed of a linear sequence of two to five containers that process requests for inferences on data. You use an inference pipeline to define and deploy any combination of pretrained SageMaker built-in algorithms and your own custom algorithms packaged in Docker containers. You can use an inference pipeline to combine preprocessing, predictions, and post-processing data science tasks. Inference pipelines are fully managed.

You can add SageMaker Spark ML Serving and scikit-learn containers that reuse the data transformers developed for training models. The entire assembled inference pipeline can be considered as a SageMaker model that you can use to make either real-time predictions or to process batch transforms directly without any external preprocessing.

Within an inference pipeline model, SageMaker handles invocations as a sequence of HTTP requests. The first container in the pipeline handles the initial request, then the intermediate response is sent as a request to the second container, and so on, for each container in the pipeline. SageMaker returns the final response to the client.

When you deploy the pipeline model, SageMaker installs and runs all of the containers on each Amazon Elastic Compute Cloud (Amazon EC2) instance in the endpoint or transform job. Feature processing and inferences run with low latency because the containers are co-located on the same EC2 instances. You define the containers for a pipeline model using the CreateModel operation or from the console. Instead of setting one PrimaryContainer, you use the Containers parameter to set the containers that make up the pipeline. You also specify the order in which the containers are executed.

A pipeline model is immutable, but you can update an inference pipeline by deploying a new one using the UpdateEndpoint operation. This modularity supports greater flexibility during experimentation.

There are no additional costs for using this feature. You pay only for the instances running on an endpoint.

Topics
- Sample Notebooks for Inference Pipelines (p. 1221)
- Feature Processing with Spark ML and Scikit-learn (p. 1221)
- Create a Pipeline Model (p. 1222)
- Run Real-time Predictions with an Inference Pipeline (p. 1224)
- Run Batch Transforms with Inference Pipelines (p. 1226)
- Inference Pipeline Logs and Metrics (p. 1227)
- Troubleshoot Inference Pipelines (p. 1232)
Sample Notebooks for Inference Pipelines

For a sample notebook that uploads and processes a dataset, trains a model, and builds a pipeline model, see the Inference Pipelines with Spark ML and XGBoost on Abalone notebook. This notebook shows how you can build your machine learning pipeline by using Spark feature Transformers and the SageMaker XGBoost algorithm. After training the model, the sample shows how to deploy the pipeline (feature Transformer and XGBoost) for real-time predictions and also performs a batch transform job using the same pipeline.

For an example that shows how to do both pre-processing and post-processing by using an inference pipeline, see Deploy Apache Spark pre-processing and post-processing with XGBoost for real-time prediction requests in Amazon SageMaker using Inference Pipelines.

For more examples that show how to create and deploy inference pipelines, see the Inference Pipelines with SparkML and BlazingText on DBPedia and Training using SparkML on EMR and hosting on SageMaker sample notebooks. For instructions on creating and accessing Jupyter notebook instances that you can use to run the example in SageMaker, see Use Amazon SageMaker Notebook Instances (p. 124).

To see a list of all the SageMaker samples, after creating and opening a notebook instance, choose the SageMaker Examples tab. There are three inference pipeline notebooks. The first two inference pipeline notebooks just described are located in the advanced_functionality folder and the third notebook is in the sagemaker-python-sdk folder. To open a notebook, choose its Use tab, then choose Create copy.

Feature Processing with Spark ML and Scikit-learn

Before training a model with either Amazon SageMaker built-in algorithms or custom algorithms, you can use Spark and scikit-learn preprocessors to transform your data and engineer features.

Feature Processing with Spark ML

You can run Spark ML jobs with AWS Glue, a serverless ETL (extract, transform, load) service, from your SageMaker notebook. You can also connect to existing EMR clusters to run Spark ML jobs with Amazon EMR. To do this, you need an AWS Identity and Access Management (IAM) role that grants permission for making calls from your SageMaker notebook to AWS Glue.

Note
To see which Python and Spark versions AWS Glue supports, refer to AWS Glue Release Notes.

After engineering features, you package and serialize Spark ML jobs with MLeap into MLeap containers that you can add to an inference pipeline. You don't need to use externally managed Spark clusters. With this approach, you can seamlessly scale from a sample of rows to terabytes of data. The same transformers work for both training and inference, so you don't need to duplicate preprocessing and feature engineering logic or develop a one-time solution to make the models persist. With inference pipelines, you don't need to maintain outside infrastructure, and you can make predictions directly from data inputs.

When you run a Spark ML job on AWS Glue, a Spark ML pipeline is serialized into MLeap format. Then, you can use the job with the SparkML Model Serving Container in a SageMaker Inference Pipeline. MLeap is a serialization format and execution engine for machine learning pipelines. It supports Spark, Scikit-learn, and TensorFlow for training pipelines and exporting them to a serialized pipeline called an MLeap Bundle. You can deserialize Bundles back into Spark for batch-mode scoring or into the MLeap runtime to power real-time API services.
Feature Processing with Sci-kit Learn

You can run and package scikit-learn jobs into containers directly in Amazon SageMaker. For an example of Python code for building a scikit-learn featurizer model that trains on Fisher’s Iris flower data set and predicts the species of Iris based on morphological measurements, see IRIS Training and Prediction with Sagemaker Scikit-learn.

Create a Pipeline Model

To create a pipeline model that can be deployed to an endpoint or used for a batch transform job, use the Amazon SageMaker console or the CreateModel operation.

To create an inference pipeline (console)

1. Open the Amazon SageMaker console at https://console.aws.amazon.com/sagemaker/.
2. Choose Models, and then choose Create models from the Inference group.
3. On the Create model page, provide a model name, choose an IAM role, and, if you want to use a private VPC, specify VPC values.
4. To add information about the containers in the inference pipeline, choose Add container, then choose Next.
5. Complete the fields for each container in the order that you want to execute them, up to the maximum of five. Complete the Container input options, Location of inference code image, and, optionally, Location of model artifacts, Container host name, and Environmental variables fields.
**Container definition 1**

- **Container Input options**
  - Provide model artifacts and inference image.

- **Provide model artifacts and inference image**
  - **Location of inference code image**
    - The registry path where the inference code image is stored in Amazon ECR.
    - `123456789012.dkr.ecr.us-east-2.amazonaws.com/myimage:v1`
  - **Location of model artifacts - optional**
    - The URL for the S3 location where model artifacts are stored.
    - `s3://bucket/path-to-your-data/`
    - The path must point to a single gzip-compressed tar archive (`.tar.gz` suffix).
  - **Container host name - optional**
    - The DNS host name for the container.

- **Environment variables - optional**
  - Key | Value | Remove
  - `key1` | `value1` | Remove
  - `key2` | `value2` | Remove

**Add environment variable**

---

- **Container definition 2 - optional**

- **Container Input options**
  - Provide model artifacts and inference image.

- **Provide model artifacts and inference image**
  - **Location of inference code image**
    - The registry path where the inference code image is stored in Amazon ECR.
    - `123456789012.dkr.ecr.us-east-2.amazonaws.com/myimage:v1`
  - **Location of model artifacts - optional**
    - The URL for the S3 location where model artifacts are stored.
    - `s3://bucket/path-to-your-data/`
    - The path must point to a single gzip-compressed tar archive (`.tar.gz` suffix).
  - **Container host name - optional**
    - The DNS host name for the container.

- **Environment variables - optional**
  - Key | Value | Remove

**Add environment variable**

---

- **Container definition 3 - optional**

- **Container Input options**
  - Provide model artifacts and inference image.

- **Provide model artifacts and inference image**
  - **Location of inference code image**
    - The registry path where the inference code image is stored in Amazon ECR.
    - `123456789012.dkr.ecr.us-east-2.amazonaws.com/myimage:v1`
The **MyInferencePipelineModel** page summarizes the settings for the containers that provide input for the model. If you provided the environment variables in a corresponding container definition, SageMaker shows them in the **Environment variables** field.

### Run Real-time Predictions with an Inference Pipeline

You can use trained models in an inference pipeline to make real-time predictions directly without performing external preprocessing. When you configure the pipeline, you can choose to use the built-in feature transformers already available in Amazon SageMaker. Or, you can implement your own transformation logic using just a few lines of scikit-learn or Spark code.

**MLeap**, a serialization format and execution engine for machine learning pipelines, supports Spark, scikit-learn, and TensorFlow for training pipelines and exporting them to a serialized pipeline called an MLeap Bundle. You can deserialize Bundles back into Spark for batch-mode scoring or into the MLeap runtime to power real-time API services.

The containers in a pipeline listen on the port specified in the `SAGEMAKER_BIND_TO_PORT` environment variable (instead of 8080). When running in an inference pipeline, SageMaker automatically provides this environment variable to containers. If this environment variable isn't present, containers default to using 1224.
port 8080. To indicate that your container complies with this requirement, use the following command to add a label to your Dockerfile:

```
LABEL com.amazonaws.sagemaker.capabilities.accept-bind-to-port=true
```

If your container needs to listen on a second port, choose a port in the range specified by the `SAGEMAKER_SAFE_PORT_RANGE` environment variable. Specify the value as an inclusive range in the format `"XXXX-YYYY"`, where XXXX and YYYY are multi-digit integers. SageMaker provides this value automatically when you run the container in a multicontainer pipeline.

**Note**

To use custom Docker images in a pipeline that includes SageMaker built-in algorithms, you need an Amazon Elastic Container Registry (Amazon ECR) policy. Your Amazon ECR repository must grant SageMaker permission to pull the image. For more information, see Troubleshoot Amazon ECR Permissions for Inference Pipelines (p. 1232).

### Create and Deploy an Inference Pipeline Endpoint

The following code creates and deploys a real-time inference pipeline model with SparkML and XGBoost models in series using the SageMaker SDK.

```python
from sagemaker.model import Model
from sagemaker.pipeline_model import PipelineModel
from sagemaker.sparkml.model import SparkMLModel

sparkml_data = 's3://{}/{}/{}'.format(s3_model_bucket, s3_model_key_prefix, 'model.tar.gz')
sparkml_model = SparkMLModel(model_data=sparkml_data)
xgb_model = Model(model_data=xgb_model.model_data, image=training_image)

model_name = 'serial-inference-' + timestamp_prefix
endpoint_name = 'serial-inference-ep-' + timestamp_prefix
sm_model = PipelineModel(name=model_name, role=role, models=[sparkml_model, xgb_model])
sm_model.deploy(initial_instance_count=1, instance_type='ml.c4.xlarge',
endpoint_name=endpoint_name)
```

### Request Real-Time Inference from an Inference Pipeline Endpoint

The following example shows how to make real-time predictions by calling an inference endpoint and passing a request payload in JSON format:

```python
import sagemaker
from sagemaker.predictor import json_serializer, json_deserializer, RealTimePredictor
from sagemaker.content_types import CONTENT_TYPE_CSV, CONTENT_TYPE_JSON

payload = {
    "input": [
        {
            "name": "Pclass",
            "type": "float",
            "val": "1.0"
        },
        {
            "name": "Embarked",
            "type": "string",
            "val": "Q"
        },
        {
            "name": "Age",
            "type": "float",
            "val": "12.5"
        }
    ]
}
```

1225
The response you get from `predictor.predict(payload)` is the model's inference result.

**Realtime inference pipeline example**

You can run this example notebook using the SKLearn predictor that shows how to deploy an endpoint, run an inference request, then deserialize the response. Find this notebook and more examples in the Amazon SageMaker example GitHub repository.

**Run Batch Transforms with Inference Pipelines**

To get inferences on an entire dataset you run a batch transform on a trained model. To run inferences on a full dataset, you can use the same inference pipeline model created and deployed to an endpoint for real-time processing in a batch transform job. To run a batch transform job in a pipeline, you download the input data from Amazon S3 and send it in one or more HTTP requests to the inference pipeline model. For an example that shows how to prepare data for a batch transform, see the "Preparing Data for Batch Transform" section of the ML Pipeline with SparkML and XGBoost - Training and Inference sample notebook. For information about Amazon SageMaker batch transforms, see Get Inferences for an Entire Dataset with Batch Transform (p. 14).

**Note**

To use custom Docker images in a pipeline that includes Amazon SageMaker built-in algorithms, you need an Amazon Elastic Container Registry (ECR) policy. Your Amazon ECR repository must grant SageMaker permission to pull the image. For more information, see Troubleshoot Amazon ECR Permissions for Inference Pipelines (p. 1232).

The following example shows how to run a transform job using the Amazon SageMaker Python SDK. In this example, `model_name` is the inference pipeline that combines SparkML and XGBoost models (created in previous examples). The Amazon S3 location specified by `input_data_path` contains the input data, in CSV format, to be downloaded and sent to the Spark ML model. After the transform
job has finished, the Amazon S3 location specified by `output_data_path` contains the output data returned by the XGBoost model in CSV format.

```python
import sagemaker
input_data_path = 's3://{}/{{}}/{{}}'.format(default_bucket, 'key', 'file_name')
output_data_path = 's3://{}{}'.format(default_bucket, 'key')
transform_job = sagemaker.transformer.Transformer(
    model_name = model_name,
    instance_count = 1,
    instance_type = 'ml.m4.xlarge',
    strategy = 'SingleRecord',
    assemble_with = 'Line',
    output_path = output_data_path,
    base_transform_job_name = 'inference-pipelines-batch',
    sagemaker_session = sagemaker.Session(),
    accept = CONTENT_TYPE_CSV,
)
transform_job.transform(data = input_data_path,
    content_type = CONTENT_TYPE_CSV,
    split_type = 'Line')
```

### Inference Pipeline Logs and Metrics

Monitoring is important for maintaining the reliability, availability, and performance of Amazon SageMaker resources. To monitor and troubleshoot inference pipeline performance, use Amazon CloudWatch logs and error messages. For information about the monitoring tools that SageMaker provides, see Monitor Amazon SageMaker (p. 1704).

#### Use Metrics to Monitor Multi-container Models

To monitor the multi-container models in Inference Pipelines, use Amazon CloudWatch. CloudWatch collects raw data and processes it into readable, near real-time metrics. SageMaker training jobs and endpoints write CloudWatch metrics and logs in the `AWS/SageMaker` namespace.

The following tables list the metrics and dimensions for the following:

- Endpoint invocations
- Training jobs, batch transform jobs, and endpoint instances

A *dimension* is a name/value pair that uniquely identifies a metric. You can assign up to 10 dimensions to a metric. For more information on monitoring with CloudWatch, see Monitor Amazon SageMaker with Amazon CloudWatch (p. 1704).

#### Endpoint Invocation Metrics

The `AWS/SageMaker` namespace includes the following request metrics from calls to `InvokeEndpoint`.

Metrics are reported at a 1-minute intervals.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invocation4XXErrors</td>
<td>The number of <code>InvokeEndpoint</code> requests that the model returned a 4xx HTTP response code for. For each 4xx response, SageMaker sends a 1.</td>
</tr>
<tr>
<td></td>
<td>Units: None</td>
</tr>
<tr>
<td></td>
<td>Valid statistics: Average, Sum</td>
</tr>
<tr>
<td>Metric</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Invocation5XXErrors</td>
<td>The number of InvokeEndpoint requests that the model returned a 5xx HTTP response code for. For each 5xx response, SageMaker sends a 1. Units: None Valid statistics: Average, Sum</td>
</tr>
<tr>
<td>Invocations</td>
<td>The number of InvokeEndpoint requests sent to a model endpoint. To get the total number of requests sent to a model endpoint, use the Sum statistic. Units: None Valid statistics: Sum, Sample Count</td>
</tr>
<tr>
<td>InvocationsPerInstance</td>
<td>The number of endpoint invocations sent to a model, normalized by InstanceCount in each ProductionVariant. SageMaker sends 1/numberOfInstances as the value for each request, where numberOfInstances is the number of active instances for the ProductionVariant at the endpoint at the time of the request. Units: None Valid statistics: Sum</td>
</tr>
<tr>
<td>ModelLatency</td>
<td>The time the model or models took to respond. This includes the time it took to send the request, to fetch the response from the model container, and to complete the inference in the container. ModelLatency is the total time taken by all containers in an inference pipeline. Units: Microseconds Valid statistics: Average, Sum, Min, Max, Sample Count</td>
</tr>
<tr>
<td>OverheadLatency</td>
<td>The time added to the time taken to respond to a client request by SageMaker for overhead. OverheadLatency is measured from the time that SageMaker receives the request until it returns a response to the client, minus the ModelLatency. Overhead latency can vary depending on request and response payload sizes, request frequency, and authentication or authorization of the request, among other factors. Units: Microseconds Valid statistics: Average, Sum, Min, Max, Sample Count</td>
</tr>
<tr>
<td>ContainerLatency</td>
<td>The time it took for an Inference Pipelines container to respond as viewed from SageMaker. ContainerLatency includes the time it took to send the request, to fetch the response from the model's container, and to complete inference in the container. Units: Microseconds Valid statistics: Average, Sum, Min, Max, Sample Count</td>
</tr>
</tbody>
</table>

Dimensions for Endpoint Invocation Metrics
<table>
<thead>
<tr>
<th>Dimension</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EndpointName,</td>
<td>Filters endpoint invocation metrics for a ProductionVariant at the specified endpoint and for the specified variant.</td>
</tr>
<tr>
<td>VariantName,</td>
<td></td>
</tr>
<tr>
<td>ContainerName</td>
<td></td>
</tr>
</tbody>
</table>

For an inference pipeline endpoint, CloudWatch lists per-container latency metrics in your account as **Endpoint Container Metrics** and **Endpoint Variant Metrics** in the **SageMaker** namespace, as follows. The **ContainerLatency** metric appears only for inferences pipelines.

For each endpoint and each container, latency metrics display names for the container, endpoint, variant, and metric.

**Training Job, Batch Transform Job, and Endpoint Instance Metrics**

The namespaces /aws/sagemaker/TrainingJobs, /aws/sagemaker/TransformJobs, and /aws/sagemaker/Endpoints include the following metrics for training jobs and endpoint instances.

Metrics are reported at a 1-minute intervals.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPUUtilization</td>
<td>The percentage of CPU units that are used by the containers running on an instance. The value ranges from 0% to 100%, and is multiplied by the number of CPUs. For example, if there are four CPUs, CPUUtilization can range from 0% to 400%. For training jobs, CPUUtilization is the CPU utilization of the algorithm container running on the instance. For batch transform jobs, CPUUtilization is the CPU utilization of the transform container running on the instance. For multi-container models, CPUUtilization is the sum of CPU utilization by all containers running on the instance. For endpoint variants, CPUUtilization is the sum of CPU utilization by all of the containers running on the instance. Units: Percent</td>
</tr>
<tr>
<td>Metric</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| MemoryUtilization | The percentage of memory that is used by the containers running on an instance. This value ranges from 0% to 100%. For training jobs, MemoryUtilization is the memory used by the algorithm container running on the instance.  
For batch transform jobs, MemoryUtilization is the memory used by the transform container running on the instance.  
For multi-container models, MemoryUtilization is the sum of memory used by all containers running on the instance.  
For endpoint variants, MemoryUtilization is the sum of memory used by all of the containers running on the instance.  
Units: Percent |
| GPUUtilization    | The percentage of GPU units that are used by the containers running on an instance. GPUUtilization ranges from 0% to 100% and is multiplied by the number of GPUs. For example, if there are four GPUs, GPUUtilization can range from 0% to 400%.  
For training jobs, GPUUtilization is the GPU used by the algorithm container running on the instance.  
For batch transform jobs, GPUUtilization is the GPU used by the transform container running on the instance.  
For multi-container models, GPUUtilization is the sum of GPU used by all containers running on the instance.  
For endpoint variants, GPUUtilization is the sum of GPU used by all of the containers running on the instance.  
Units: Percent |
| GPUMemoryUtilization | The percentage of GPU memory used by the containers running on an instance. GPUMemoryUtilization ranges from 0% to 100% and is multiplied by the number of GPUs. For example, if there are four GPUs, GPUMemoryUtilization can range from 0% to 400%.  
For training jobs, GPUMemoryUtilization is the GPU memory used by the algorithm container running on the instance.  
For batch transform jobs, GPUMemoryUtilization is the GPU memory used by the transform container running on the instance.  
For multi-container models, GPUMemoryUtilization is sum of GPU used by all containers running on the instance.  
For endpoint variants, GPUMemoryUtilization is the sum of the GPU memory used by all of the containers running on the instance.  
Units: Percent |
### Metric Description

<table>
<thead>
<tr>
<th>Metric</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DiskUtilization</td>
<td>The percentage of disk space used by the containers running on an instance. DiskUtilization ranges from 0% to 100%. This metric is not supported for batch transform jobs.</td>
</tr>
<tr>
<td></td>
<td>For training jobs, DiskUtilization is the disk space used by the algorithm container running on the instance.</td>
</tr>
<tr>
<td></td>
<td>For endpoint variants, DiskUtilization is the sum of the disk space used by all of the provided containers running on the instance.</td>
</tr>
<tr>
<td>Units: Percent</td>
<td></td>
</tr>
</tbody>
</table>

### Dimensions for Training Job, Batch Transform Job, and Endpoint Instance Metrics

<table>
<thead>
<tr>
<th>Dimension</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Host</td>
<td>For training jobs, Host has the format [training-job-name]/algo-[instance-number-in-cluster]. Use this dimension to filter instance metrics for the specified training job and instance. This dimension format is present only in the /aws/sagemaker/TrainingJobs namespace.</td>
</tr>
<tr>
<td></td>
<td>For batch transform jobs, Host has the format [transform-job-name]/[instance-id]. Use this dimension to filter instance metrics for the specified batch transform job and instance. This dimension format is present only in the /aws/sagemaker/TransformJobs namespace.</td>
</tr>
<tr>
<td></td>
<td>For endpoints, Host has the format [endpoint-name]/[production-variant-name]/[instance-id]. Use this dimension to filter instance metrics for the specified endpoint, variant, and instance. This dimension format is present only in the /aws/sagemaker/Endpoints namespace.</td>
</tr>
</tbody>
</table>

To help you debug your training jobs, endpoints, and notebook instance lifecycle configurations, SageMaker also sends anything an algorithm container, a model container, or a notebook instance lifecycle configuration sends to stdout or stderr to Amazon CloudWatch Logs. You can use this information for debugging and to analyze progress.

### Use Logs to Monitor an Inference Pipeline

The following table lists the log groups and log streams SageMaker sends to Amazon CloudWatch.

A log stream is a sequence of log events that share the same source. Each separate source of logs into CloudWatch makes up a separate log stream. A log group is a group of log streams that share the same retention, monitoring, and access control settings.

#### Logs

<table>
<thead>
<tr>
<th>Log Group Name</th>
<th>Log Stream Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>/aws/sagemaker/TrainingJobs</td>
<td>[training-job-name]/algo-[instance-number-in-cluster]-[epoch_timestamp]</td>
</tr>
<tr>
<td>/aws/sagemaker/Endpoints/[EndpointName]</td>
<td>[production-variant-name]/[instance-id]</td>
</tr>
<tr>
<td>/aws/sagemaker/Endpoints/[EndpointName]</td>
<td>[production-variant-name]/[instance-id]</td>
</tr>
</tbody>
</table>
## Log Group Name | Log Stream Name
---|---
[production-variant-name]/[instance-id]/[container-name provided in the SageMaker model] (For Inference Pipelines) For Inference Pipelines logs, if you don't provide container names, CloudWatch uses **container-1, container-2**, and so on, in the order that containers are provided in the model.
/aws/sagemaker/NotebookInstances [notebook-instance-name]/[LifecycleConfigHook]
/aws/sagemaker/TransformJobs [transform-job-name]/[instance-id]-[epoch_timestamp]
[transform-job-name]/[instance-id]-[epoch_timestamp]/data-log [container-name provided in the SageMaker model] (For Inference Pipelines) For Inference Pipelines logs, if you don't provide container names, CloudWatch uses **container-1, container-2**, and so on, in the order that containers are provided in the model.

**Note**
SageMaker creates the /aws/sagemaker/NotebookInstances log group when you create a notebook instance with a lifecycle configuration. For more information, see Customize a Notebook Instance Using a Lifecycle Configuration Script (p. 127).

For more information about SageMaker logging, see Log Amazon SageMaker Events with Amazon CloudWatch (p. 1712).

### Troubleshoot Inference Pipelines

To troubleshoot inference pipeline issues, use CloudWatch logs and error messages. If you are using custom Docker images in a pipeline that includes Amazon SageMaker built-in algorithms, you might also encounter permissions problems. To grant the required permissions, create an Amazon Elastic Container Registry (Amazon ECR) policy.

**Topics**
- Troubleshoot Amazon ECR Permissions for Inference Pipelines (p. 1232)
- Use CloudWatch Logs to Troubleshoot SageMaker Inference Pipelines (p. 1233)
- Use Error Messages to Troubleshoot Inference Pipelines (p. 1234)

### Troubleshoot Amazon ECR Permissions for Inference Pipelines

When you use custom Docker images in a pipeline that includes SageMaker built-in algorithms, you need an Amazon ECR policy. The policy allows your Amazon ECR repository to grant permission for SageMaker to pull the image. The policy must add the following permissions:

```json
{
  "Version": "2008-10-17",
  "Statement": [
    {
      "Sid": "allowSageMakerToPull",
      "Effect": "Allow",
      "Principal": {
        "Service": "sagemaker.amazonaws.com"
      }
    }
  ]
}
```
Use CloudWatch Logs to Troubleshoot SageMaker Inference Pipelines

SageMaker publishes the container logs for endpoints that deploy an inference pipeline to Amazon CloudWatch at the following path for each container.

/aws/sagemaker/Endpoints/{EndpointName}/{Variant}/{InstanceId}/{ContainerHostname}

For example, logs for this endpoint are published to the following log groups and streams:

- **EndpointName**: MyInferencePipelinesEndpoint
- **Variant**: MyInferencePipelinesVariant
- **InstanceId**: i-0179208609ff7e488
- **ContainerHostname**: MyContainerName1 and MyContainerName2

A **log stream** is a sequence of log events that share the same source. Each separate source of logs into CloudWatch makes up a separate log stream. A **log group** is a group of log streams that share the same retention, monitoring, and access control settings.

**To see the log groups and streams**

2. In the navigation page, choose Logs.
3. In Log Groups, filter on MyInferencePipelinesEndpoint:

![Image of CloudWatch console showing log groups and streams](image)

4. To see the log streams, on the CloudWatch Log Groups page, choose MyInferencePipelinesEndpoint, and then Search Log Group.
For a list of the logs that SageMaker publishes, see Inference Pipeline Logs and Metrics (p. 1227).

Use Error Messages to Troubleshoot Inference Pipelines

The inference pipeline error messages indicate which containers failed.

If an error occurs while SageMaker is invoking an endpoint, the service returns a ModelError (error code 424), which indicates which container failed. If the request payload (the response from the previous container) exceeds the limit of 5 MB, SageMaker provides a detailed error message, such as:

Received response from MyContainerName1 with status code 200. However, the request payload from MyContainerName1 to MyContainerName2 is 6000000 bytes, which has exceeded the maximum limit of 5 MB. See https://us-west-2.console.aws.amazon.com/cloudwatch/home?region=us-west-2#logEventViewer:group=/aws/sagemaker/Endpoints/MyInferencePipelinesEndpoint in account 123456789012 for more information.

If a container fails the ping health check while SageMaker is creating an endpoint, it returns a ClientError and indicates all of the containers that failed the ping check in the last health check.

Compile and Deploy Models with Neo

Neo is a capability of Amazon SageMaker that enables machine learning models to train once and run anywhere in the cloud and at the edge.

If you are a first time user of SageMaker Neo, we recommend you check out the Getting Started with Edge Devices section to get step-by-step instructions on how to compile and deploy to an edge device.

What is SageMaker Neo?

Generally, optimizing machine learning models for inference on multiple platforms is difficult because you need to hand-tune models for the specific hardware and software configuration of each platform. If you want to get optimal performance for a given workload, you need to know the hardware architecture, instruction set, memory access patterns, and input data shapes, among other factors. For traditional software development, tools such as compilers and profilers simplify the process. For machine learning, most tools are specific to the framework or to the hardware. This forces you into a manual trial-and-error process that is unreliable and unproductive.

Neo automatically optimizes Gluon, Keras, MXNet, PyTorch, TensorFlow, TensorFlow-Lite, and ONNX models for inference on Android, Linux, and Windows machines based on processors from Ambarella, ARM, Intel, Nvidia, NXP, Qualcomm, Texas Instruments, and Xilinx. Neo is tested with computer vision models available in the model zoos across the frameworks.

How it Works

Neo consists of a compiler and a runtime. First, the Neo compilation API reads models exported from various frameworks. It converts the framework-specific functions and operations into a framework-agnostic intermediate representation. Next, it performs a series of optimizations. Then it generates binary code for the optimized operations, writes them to a shared object library, and saves the model definition and parameters into separate files. Neo also provides a runtime for each target platform that loads and executes the compiled model.
You can create a Neo compilation job from either the SageMaker console, the AWS Command Line Interface (AWS CLI), a Python notebook, or the SageMaker SDK. With a few CLI commands, an API invocation, or a few clicks, you can convert a model for your chosen platform. You can deploy the model to a SageMaker endpoint or on an AWS IoT Greengrass device quickly.

Neo can optimize models with parameters either in FP32 or quantized to INT8 or FP16 bit-width.

**Neo Sample Notebooks**

For sample notebooks that use SageMaker Neo to train, compile, optimize, and deploy machine learning models to make inferences, see:

- MNIST Training, Compilation and Deployment with MXNet Module
- MNIST Training, Compilation and Deployment with Tensorflow Module
- Deploying pre-trained PyTorch vision models with SageMaker Neo
- Model Optimization with an Image Classification Example
- Model Optimization with XGBoost Example

For instructions on how to run these example notebooks in SageMaker, see Example Notebooks (p. 134). If you need instructions on how to create a notebook instance to run these examples, see Use Amazon SageMaker Notebook Instances (p. 124). To navigate to the relevant example in your notebook instance, choose the Amazon SageMaker Examples tab to see a list of all of the SageMaker samples. To open a notebook, choose its Use tab, then choose Create copy.
Use Neo to Compile a Model

This section shows how to create, describe, stop, and list compilation jobs. The following options are available in Amazon SageMaker Neo for managing the compilation jobs for machine learning models: the Neo CLI, the Amazon SageMaker console, or the Amazon SageMaker SDK.

What input data shapes does SageMaker Neo expect?

Before you compile your model, make sure your model is formatted correctly. Neo expects the name and shape of the expected data inputs for your trained model with JSON format or list format. The expected inputs are framework specific.

Below are the input shapes SageMaker Neo expects:

**TensorFlow**

Specify either the name and shape (NHWC format) of the expected data inputs using a dictionary format for your trained model. The dictionary formats required are as follows:

- For one input: {"input": [1,1024,1024,3]}
- For two inputs: {'data1': [1,28,28,1], 'data2': [1,28,28,1]}

**Keras**

Specify either the name and shape (NCHW format) of expected data inputs using a dictionary format for your trained model. Note that while Keras model artifacts should be uploaded in NHWC (channel-last) format, DataInputConfig should be specified in NCHW (channel-first) format. The dictionary formats required are as follows:

- For one input: {'input_1': [1,3,224,224]}
- For two inputs: {'input_1': [1,3,224,224], 'input_2': [1,3,224,224]}

**MXNet/ONNX**

Specify either the name and shape (NCHW format) of the expected data inputs in order using a dictionary format for your trained model. The dictionary formats required are as follows:

- For one input: {'data': [1,3,1024,1024]}
- For two inputs: {'var1': [1,1,28,28], 'var2': [1,1,28,28]}

**PyTorch**

Specify either the name and shape (NCHW format) of expected data inputs in order using a dictionary format for your trained model or you can specify the shape only using a list format. The dictionary formats required are as follows:

- For one input in dictionary format: {'input0': [1,3,224,224]}
- For one input in list format: [[1,3,224,224]]
- For two inputs in dictionary format: {'input0': [1,3,224,224], 'input1': [1,3,224,224]}
- For two inputs in list format: [[1,3,224,224], [1,3,224,224]]

**XGBoost**

Input data name and shape are not needed.
TFLite

Specify either the name and shape (NHWC format) of the expected data inputs in order using a
dictionary format for your trained model. The dictionary formats required are as follows:

- For one input: `{ 'input': [1, 224, 224, 3] }`

Topics

- Compile a Model (AWS Command Line Interface) (p. 1237)
- Compile a Model (Amazon SageMaker Console) (p. 1238)
- Compile a Model (Amazon SageMaker SDK) (p. 1241)

Compile a Model (AWS Command Line Interface)

This section shows how to manage Amazon SageMaker Neo compilation jobs for machine learning
models using AWS Command Line Interface (CLI). You can create, describe, stop, and list the compilation
jobs.

1. Create a Compilation Job

   With the CreateCompilationJob API operation, you can specify the data input format, the S3 bucket
   in which to store your model, the S3 bucket to which to write the compiled model, and the target
   hardware device or platform.

   For a target device:

   ```json
   {  
       "CompilationJobName": "neo-compilation-job-demo",
       "RoleArn": "arn:aws:iam::<your-account>:role/service-role/AmazonSageMaker-ExecutionRole-yyyyymmddThhmmss",
       "InputConfig": {  
           "S3Uri": "s3://<your-bucket>/sagemaker/neo-compilation-job-demo-data/train",
           "DataInputConfig": "{'data': [1, 3, 1024, 1024]}",
           "Framework": "MXNET"
       },
       "OutputConfig": {  
           "S3OutputLocation": "s3://<your-bucket>/sagemaker/neo-compilation-job-demo-data/compile",
           "TargetDevice": "ml_c5"
       },
       "StoppingCondition": {  
           "MaxRuntimeInSeconds": 300
       }
   }
   ```

   For a target platform:

   ```json
   {  
       "CompilationJobName": "neo-test-compilation-job",
       "RoleArn": "arn:aws:iam::<your-account>:role/service-role/AmazonSageMaker-ExecutionRole-yyyyymmddThhmmss",
       "InputConfig": {  
           "S3Uri": "s3://<your-bucket>/sagemaker/neo-compilation-job-demo-data/train",
           "DataInputConfig": "{'data': [1, 3, 1024, 1024]}",
           "Framework": "MXNET"
       },
       "OutputConfig": {
   ```
"S3OutputLocation": "s3:///your-bucket/sagemaker/neo-compilation-job-demo-data/compile",
    # A target platform configuration example for a p3.2xlarge instance
    "TargetPlatform": {
        "Os": "LINUX",
        "Arch": "X86_64",
        "Accelerator": "NVIDIA"
    },
    "CompilerOptions": "{'cuda-ver': '10.0', 'trt-ver': '6.0.1', 'gpu-code': 'sm_70'}",
    "StoppingCondition": {
        "MaxRuntimeInSeconds": 300
    }
}

Note
For the OutputConfig API operation, the TargetDevice and TargetPlatform API operations are mutually exclusive. You have to choose one of the two options.

To find the JSON string examples of DataInputConfig depending on frameworks, see What input data shapes Neo expects.

For more information about setting up the configurations, see the InputConfig, OutputConfig, and TargetPlatform API operations in the SageMaker API reference.

2. After you configure the JSON file, run the following command to create the compilation job:

```bash
aws sagemaker create-compilation-job \
--cli-input-json file://job.json \
--region us-west-2
# You should get CompilationJobArn
```

3. Describe the compilation job by running the following command:

```bash
aws sagemaker describe-compilation-job \
--compilation-job-name $JOB_NM \
--region us-west-2
```

4. Stop the compilation job by running the following command:

```bash
aws sagemaker stop-compilation-job \
--compilation-job-name $JOB_NM \
--region us-west-2
# There is no output for compilation-job operation
```

5. List the compilation job by running the following command:

```bash
aws sagemaker list-compilation-jobs \
--region us-west-2
```

Compile a Model (Amazon SageMaker Console)

You can create an Amazon SageMaker Neo compilation job in the Amazon SageMaker console.

1. In the Amazon SageMaker console, choose Compilation jobs, and then choose Create compilation job.
2. On the **Create compilation job** page, under **Job name**, enter a name. Then select an **IAM role**.

3. If you don't have an IAM role, choose **Create a new role**.

4. On the **Create an IAM role** page, choose **Any S3 bucket**, and choose **Create role**.
5. Within the **Input configuration** section, enter the full path of the Amazon S3 bucket URI that contains your model artifacts in the **Location of model artifacts** input field. Your model artifacts must be in a compressed tarball file format (.tar.gz).

**Warning**
If you specify a Amazon S3 bucket URI path that leads to .pth file, you will receive the following error after starting compilation: `ClientError: InputConfiguration: Unable to untar input model. Please confirm the model is a tar.gz file`.

For the **Data input configuration** field, enter the JSON string that specifies the shape of the input data. For **Machine learning framework**, choose the framework of your choice.

To find the JSON string examples of input data shapes depending on frameworks, see [What input data shapes Neo expects](#).

6. Go to the **Output configuration** section. Choose where you want to deploy your model. You can deploy your model to a **Target device** or a **Target platform**. Target devices include cloud and edge devices. Target platforms refer to specific OS, architecture, and accelerators you want your model to run on.

For **S3 Output location**, enter the path to the S3 bucket where you want to store the model. You can optionally add compiler options in JSON format under the **Compiler options** section.
7. Check the status of the compilation job when started. This status of the job can be found at the top of the Compilation Job page, as shown in the following screenshot. You can also check the status of it in the Status column.

8. Check the status of the compilation job when completed. You can check the status in the Status column as shown in the following screenshot.

Compile a Model (Amazon SageMaker SDK)

You can use the `compile_model` API in the Amazon SageMaker SDK for Python to compile a trained model and optimize it for specific target hardware. The API should be invoked on the estimator object used during model training.
Note
You must set `MMS_DEFAULT_RESPONSE_TIMEOUT` environment variable to `500` when compiling the model with MXNet or PyTorch. The environment variable is not needed for TensorFlow.

The following is an example of how you can compile a model using the `trained_model_estimator` object:

```python
# Replace the value of expected_trained_model_input below and
# specify the name & shape of the expected inputs for your trained model
# in json dictionary form
expected_trained_model_input = {'data':[1, 784]}

# Replace the example target_instance_family below to your preferred target_instance_family
compiled_model = trained_model_estimator.compile_model(target_instance_family='ml_c5',
                                        input_shape=expected_trained_model_input,
                                        output_path='insert s3 output path',
                                        env={'MMS_DEFAULT_RESPONSE_TIMEOUT': '500'})
```

The code compiles the model, saves the optimized model at `output_path`, and creates a SageMaker model that can be deployed to an endpoint. Sample notebooks of using the SDK for Python are provided in the Neo Model Compilation Sample Notebooks section.

Cloud Instances

Amazon SageMaker Neo provides compilation support for popular machine learning frameworks such as TensorFlow, PyTorch, MXNet, and more. You can deploy your compiled model to cloud instances and AWS Inferentia instances. For a full list of supported frameworks and instances types, see Supported Instance Types and Frameworks.

You can compile your model in one of three ways: through the AWS CLI, the SageMaker Console, or the SageMaker SDK for Python. See, Use Neo to Compile a Model for more information. Once compiled, your model artifacts are stored in the Amazon S3 bucket URI you specified during the compilation job. You can deploy your compiled model to cloud instances and AWS Inferentia instances using the SageMaker SDK for Python, AWS SDK for Python (Boto3), AWS CLI, or the AWS console.

If you deploy your model using AWS CLI, the console, or Boto3, you must select a Docker image Amazon ECR URI for your primary container. See Neo Inference Container Images for a list of Amazon ECR URIs.

Topics
- Supported Instance Types and Frameworks (p. 1242)
- Deploy a Model (p. 1245)
- Request Inferences from a Deployed Service (p. 1260)
- Inference Container Images (p. 1263)

Supported Instance Types and Frameworks

Amazon SageMaker Neo supports popular deep learning frameworks for both compilation and deployment. You can deploy your model either to a cloud instance or to AWS Inferentia instance types. The following describes frameworks SageMaker Neo supports and the cloud and Inferentia instance types you can deploy your model to.

For information on how to deploy your compiled model to a cloud or Inferentia instance, see Deploy a Model with Cloud Instances.

Cloud Instances

SageMaker Neo supports the following deep learning frameworks for CPU and GPU cloud instances:
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>MXNet</td>
<td>1.7.0</td>
<td>Supports 1.7.0 or earlier</td>
<td>Image Classification, Object Detection, Semantic Segmentation, Pose Estimation, Activity Recognition</td>
<td>One symbol file (.json) and one parameter file (.params)</td>
<td>GluonCV v0.8.0</td>
</tr>
<tr>
<td>ONNX</td>
<td>1.5.0</td>
<td>Supports 1.5.0 or earlier</td>
<td>Image Classification, SVM</td>
<td>One model file (.onnx)</td>
<td></td>
</tr>
<tr>
<td>Keras</td>
<td>2.2.4</td>
<td>Supports 2.2.4 or earlier</td>
<td>Image Classification</td>
<td>One model definition file (.h5)</td>
<td></td>
</tr>
<tr>
<td>PyTorch</td>
<td>1.4.0</td>
<td>Supports 1.4.0 or earlier</td>
<td>Image Classification</td>
<td>One model definition file (.pt or .pth) with input dtype of float32</td>
<td></td>
</tr>
<tr>
<td>TensorFlow</td>
<td>1.15.0</td>
<td>Supports 1.15.0 or earlier</td>
<td>Image Classification</td>
<td>*For saved models, one .pb or one .pbtxt file and a variables directory that contains variables *For frozen models, only one .pb or .pbtxt file</td>
<td></td>
</tr>
<tr>
<td>TensorFlow-Lite</td>
<td>1.13.1</td>
<td>Supports 1.13.1 or earlier</td>
<td>Image Classification, Object Detection</td>
<td>One model definition flatbuffer file (.tflite)</td>
<td></td>
</tr>
<tr>
<td>XGBoost</td>
<td>0.9</td>
<td>Supports 0.9 or earlier</td>
<td>Decision Trees</td>
<td>One XGBoost model file (.model) where the number of nodes in a tree is less than 2^31</td>
<td></td>
</tr>
<tr>
<td>DARKNET</td>
<td></td>
<td></td>
<td>Image Classification</td>
<td>One config (.cfg) file and</td>
<td></td>
</tr>
</tbody>
</table>
### Cloud Instances

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Object Detection</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>one weights (.weights) file</td>
<td></td>
</tr>
</tbody>
</table>

**Note**

“Model Version” is the version of the framework used to train and export the model.

You can deploy your SageMaker compiled model to one of the cloud instances listed below:

- `ml_m5`
- `ml_c4`
- `ml_c5`
- `ml_p2`
- `ml_p3`
- `ml_g4dn`

For information on the available vCPU, memory, and price per hour for each instance type, see Amazon SageMaker Pricing.

### AWS Inferentia

SageMaker Neo supports the following deep learning frameworks for Inferentia:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>MXNet</td>
<td>1.5.1</td>
<td>Supports 1.5.1 or earlier</td>
<td>Image Classification, Object Detection, Semantic Segmentation, Pose Estimation, Activity Recognition</td>
<td>One symbol file (.json) and one parameter file (.params)</td>
<td>GluonCV v0.8.0</td>
</tr>
<tr>
<td>PyTorch</td>
<td>1.5.1</td>
<td>Supports 1.5.1 or earlier</td>
<td>Image Classification</td>
<td>One model definition file (.pt or .pth) with input dtype of float32</td>
<td></td>
</tr>
<tr>
<td>TensorFlow</td>
<td>1.15.0</td>
<td>Supports 1.15.0 or earlier</td>
<td>Image Classification</td>
<td>*For saved models, one .pb or one .pbtxt file and a variables</td>
<td></td>
</tr>
</tbody>
</table>
Amazon SageMaker Developer Guide
Cloud Instances

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>directory that contains variables *For frozen models, only one .pb or .pbtxt file</td>
<td></td>
</tr>
</tbody>
</table>

**Note**

"Model Version" is the version of the framework used to train and export the model.

You can deploy your SageMaker Neo-compiled model to AWS Inferentia-based Amazon EC2 Inf1 instances. AWS Inferentia is Amazon's first custom silicon chip designed to accelerate deep learning. Currently, you can use the ml_inf1 instance to deploy your compiled models.

**Deploy a Model**

To deploy an Amazon SageMaker Neo-compiled model to an HTTPS endpoint, you must configure and create the endpoint for the model using Amazon SageMaker hosting services. Currently, developers can use Amazon SageMaker APIs to deploy modules on to ml.c5, ml.c4, ml.m5, ml.m4, ml.p3, ml.p2, and ml.inf1 instances.

For Inf1 instances, models need to be compiled specifically for ml.inf1 instances. Models compiled for other instance types are not guaranteed to work with ml.inf1 instances.

When you deploy a compiled model, you need to use the same instance for the target that you used for compilation. This creates a SageMaker endpoint that you can use to perform inferences. You can deploy a Neo-compiled model using any of the following: Amazon SageMaker SDK for Python, SDK for Python (Boto3), AWS Command Line Interface, and SageMaker Console.

**Note**

For deploying a model using AWS CLI, the console, or Boto3, see Neo Inference Container Images to select the inference image URI for your primary container.

**Topics**

- Prerequisites (p. 1245)
- Deploy a Compiled Model Using SageMaker SDK (p. 1250)
- Deploy a Compiled Model Using Boto3 (p. 1252)
- Deploy a Compiled Model Using the AWS CLI (p. 1254)
- Deploy a Compiled Model Using the Console (p. 1256)

**Prerequisites**

**Note**

Follow the instructions in this section if you compiled your model using AWS SDK for Python (Boto3), AWS CLI, or the SageMaker console.

To create a SageMaker Neo-compiled model, you need the following:

1. A Docker image Amazon ECR URI. You can select one that meets your needs from this list.
2. An entry point script file:
   a. **For PyTorch and MXNet models:**
If you trained your model using SageMaker, the training script must implement the functions described below. The training script serves as the entry point script during inference. In the example detailed in MNIST Training, Compilation and Deployment with MXNet Module and SageMaker Neo, the training script (mnist.py) implements the required functions.

If you did not train your model using SageMaker, you need to provide an entry point script (inference.py) file that can be used at the time of inference. Based on the framework—MXNet or PyTorch—the inference script location must conform to the SageMaker Python SDK Model Directory Structure for MXNet or Model Directory Structure for PyTorch.

When using Neo Inference Optimized Container images with PyTorch and MXNet on CPU and GPU instance types, the inference script must implement the following functions:

- model_fn: Loads the model.
- input_fn: Converts the incoming request payload into a numpy array.
- predict_fn: Performs the prediction.
- output_fn: Converts the prediction output into the response payload.
- Alternatively, you can define transform_fn to combine input_fn, predict_fn, and output_fn.

The following are examples of inference.py script within a directory named code (code/inference.py) for PyTorch and MXNet (Gluon and Module). The examples first load the model and then serve it on image data on a GPU:

**MXNet Module**

```python
import numpy as np
import json
import mxnet as mx
import neomxnet  # noqa: F401
from collections import namedtuple
Batch = namedtuple('Batch', ['data'])

ctx = mx.gpu()
def model_fn(model_dir):
    # The compiled model artifacts are saved with the prefix 'compiled'
    sym, arg_params, aux_params = mx.model.load_checkpoint('compiled', 0)
    mod = mx.mod.Module(symbol=sym, context=ctx, label_names=None)
    exe = mod.bind(for_training=False,
                   data_shapes=[('data', (1,3,224,224))],
                   label_shapes=mod._label_shapes)
    mod.set_params(arg_params, aux_params, allow_missing=True)

    # Run warm-up inference on empty data during model load (required for GPU)
    data = mx.nd.empty((1,3,224,224), ctx=ctx)
    mod.forward(Batch([data]))
    return mod

def transform_fn(mod, image, input_content_type, output_content_type):
    # pre-processing
    decoded = mx.image.imdecode(image)
    resized = mx.image.resize_short(decoded, 224)
    cropped, crop_info = mx.image.center_crop(resized, (224, 224))
    normalized = mx.image.color_normalize(cropped.astype(np.float32) / 255, mean=mx.nd.array([0.485, 0.456, 0.406]),
```
```python
std=mx.nd.array([0.229, 0.224, 0.225]))
transposed = normalized.transpose((2, 0, 1))
batchified = transposed.expand_dims(axis=0)
casted = batchified.astype(dtype='float32')
processed_input = casted.as_in_context(ctx)

# prediction/inference
mod.forward(Batch([processed_input]))

# post-processing
prob = mod.get_outputs()[0].asnumpy().tolist()
prob_json = json.dumps(prob)
return prob_json, output_content_type
```

**MXNet Gluon**

```python
import numpy as np
import json
import mxnet as mx
import neomxnet  # noqa: F401

# Change the context to mx.cpu() if deploying to a CPU endpoint
cctx = mx.gpu()

def model_fn(model_dir):
    # The compiled model artifacts are saved with the prefix 'compiled'
    block = mx.gluon.nn.SymbolBlock.imports('compiled-symbol.json', ['data'], 'compiled-0000.params', ctx=cctx)
    # Hybridize the model & pass required options for Neo: static_alloc=True & static_shape=True
    block.hybridize(static_alloc=True, static_shape=True)
    # Run warm-up inference on empty data during model load (required for GPU)
data = mx.nd.empty((1,3,224,224), ctx=cctx)
    warm_up = block(data)
    return block

def input_fn(image, input_content_type):
    # pre-processing
    decoded = mx.image.imdecode(image)
    resized = mx.image.resize_short(decoded, 224)
    cropped, crop_info = mx.image.center_crop(resized, (224, 224))
    normalized = mx.image.color_normalize(cropped.astype(np.float32) / 255,
        mean=mx.nd.array([0.485, 0.456, 0.406]),
        std=mx.nd.array([0.229, 0.224, 0.225]))
    transposed = normalized.transpose((2, 0, 1))
    batchified = transposed.expand_dims(axis=0)
    casted = batchified.astype(dtype='float32')
    processed_input = casted.as_in_context(ctx)
    return processed_input

def predict_fn(processed_input_data, block):
    # prediction/inference
    prediction = block(processed_input_data)
    return prediction

def output_fn(prediction, output_content_type):
    # post-processing
    prob = prediction.asnumpy().tolist()
    prob_json = json.dumps(prob)
    return prob_json, output_content_type
```
```python
import os
import torch
import torch.nn.parallel
import torch.optim
import torch.utils.data
import torch.utils.data.distributed
import torchvision.transforms as transforms
from PIL import Image
import io
import json
import pickle

def model_fn(model_dir):
    # Load the model and return it.
    # Providing this function is optional.
    # There is a default model_fn available which will load the model
    # compiled using SageMaker Neo. You can override it here.

    # The compiled model is saved as "compiled.pt"
    model_path = os.path.join(model_dir, 'compiled.pt')
    with torch.jit.load(model_path):
        model = torch.jit.load(model_path)

    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    model = model.to(device)

    # We recommend that you run warm-up inference during model load
    sample_input_path = os.path.join(model_dir, 'sample_input.pkl')
    with open(sample_input_path, 'rb') as input_file:
        model_input = pickle.load(input_file)

    if torch.is_tensor(model_input):
        model_input = model_input.to(device)
    elif isinstance(model_input, tuple):
        model_input = (inp.to(device) for inp in model_input if torch.is_tensor(inp))
    else:
        print("Only supports a torch tensor or a tuple of torch tensors")
        return model

def transform_fn(model, request_body, request_content_type, response_content_type):
    # Run prediction and return the output.
    # The function
    # 1. Pre-processes the input request
    # 2. Runs prediction
    # 3. Post-processes the prediction output.

    decoded = Image.open(io.BytesIO(request_body))
    preprocess = transforms.Compose([transforms.Resize(256), transforms.CenterCrop(224), transforms.ToTensor(), transforms.Normalize(mean=[1248]])
```

PyTorch
b. For inf1 instances or onnx, xgboost, keras container images

For all other Neo Inference-optimized container images, or inferentia instance types, the entry point script must implement the following functions for Neo Deep Learning Runtime:

- `neo_preprocess`: Converts the incoming request payload into a numpy array.
- `neo_postprocess`: Converts the prediction output from Neo Deep Learning Runtime into the response body.

**Note**
The preceding two functions do not use any of the functionalities of MXNet, PyTorch, or TensorFlow.

For examples of how to use these functions, see Neo Model Compilation Sample Notebooks.

c. For TensorFlow models

If your model requires custom pre- and post-processing logic before data is sent to the model, then you must specify an entry point script `inference.py` file that can be used at the time of inference. The script should implement either a either a pair of `input_handler` and `output_handler` functions or a single handler function.

**Note**
Note that if handler function is implemented, `input_handler` and `output_handler` are ignored.

The following is a code example of `inference.py` script that you can put together with the compile model to perform custom pre- and post-processing on an image classification model. The SageMaker client sends the image file as an `application/x-image` content type to the `input_handler` function, where it is converted to JSON. The converted image file is then sent to the Tensorflow Model Server (TFX) using then REST API.

```python
import json
import numpy as np
import json
import io
from PIL import Image

def input_handler(data, context):
    """ Pre-process request input before it is sent to TensorFlow Serving REST API ""
    Args:
    data (obj): the request data, in format of dict or string
    context (Context): an object containing request and configuration details

    Returns:
    (dict): a JSON-serializable dict that contains request body and headers
    """
    f = data.read()
    f = io.BytesIO(f)
```
image = Image.open(f).convert('RGB')
batch_size = 1
image = np.asarray(image.resize((512, 512)))
image = np.concatenate([image[np.newaxis, :, :] * batch_size])
body = json.dumps({'signature_name': 'serving_default', 'instances':
image.tolist()})
return body

def output_handler(data, context):
    """Post-process TensorFlow Serving output before it is returned to the client."
    Args:
    data (obj): the TensorFlow serving response
    context (Context): an object containing request and configuration details
    Returns:
    (bytes, string): data to return to client, response content type
    if data.status_code != 200:
        raise ValueError(data.content.decode('utf-8'))
    response_content_type = context.accept_header
    prediction = data.content
    return prediction, response_content_type

If there is no custom pre- or post-processing, the SageMaker client converts the file image to
JSON in a similar way before sending it over to the SageMaker endpoint.

For more information, see the Deploying to TensorFlow Serving Endpoints in the SageMaker
Python SDK.

3. The Amazon S3 bucket URI that contains the compiled model artifacts.

Deploy a Compiled Model Using SageMaker SDK

You must satisfy the prerequisites section if the model was compiled using AWS SDK for Python (Boto3),
AWS CLI, or the Amazon SageMaker console. Follow one of the following use cases to deploy a model
compiled with SageMaker Neo based on you how you compiled your model.

Topics
- If you compiled your model using the SageMaker SDK (p. 1250)
- If you compiled your model using MXNet or PyTorch (p. 1251)
- If you compiled your model using Boto3, SageMaker console, or the CLI for TensorFlow (p. 1252)

If you compiled your model using the SageMaker SDK

The sagemaker.Model object handle for the compiled model supplies the deploy() function, which
enables you to create an endpoint to serve inference requests. The function lets you set the number
and type of instances that are used for the endpoint. You must choose an instance for which you have
compiled your model. For example, in the job compiled in Compile a Model (Amazon SageMaker SDK)
section, this is ml_c5.

predictor = compiled_model.deploy(initial_instance_count = 1, instance_type =
'ml.c5.4xlarge')
# Print the name of newly created endpoint
print(predictor.endpoint_name)
If you compiled your model using MXNet or PyTorch

Create the SageMaker model and deploy it using the deploy() API under the framework-specific Model APIs. For MXNet, it is MXNetModel and for PyTorch, it is PyTorchModel. When you are creating and deploying an SageMaker model, you must set MMS_DEFAULT_RESPONSE_TIMEOUT environment variable to 500 and specify the entry_point parameter as the inference script (inference.py) and the source_dir parameter as the directory location (code) of the inference script. To prepare the inference script (inference.py) follow the Prerequisites step.

The following example shows how to use these functions to deploy a compiled model using the SageMaker SDK for Python:

**MXNet**

```python
from sagemaker.mxnet import MXNetModel

# Create SageMaker model and deploy an endpoint
sm_mxnet_compiled_model = MXNetModel(
    model_data='insert S3 path of compiled MXNet model archive',
    role='AmazonSageMaker-ExecutionRole',
    entry_point='inference.py',
    source_dir='code',
    framework_version='1.7.0',
    py_version='py3',
    image_uri='insert appropriate ECR Image URI for MXNet',
    env={'MMS_DEFAULT_RESPONSE_TIMEOUT': '500'},
)

# Replace the example instance_type below to your preferred instance_type
predictor = sm_mxnet_compiled_model.deploy(initial_instance_count = 1, instance_type = 'ml.p3.2xlarge')

# Print the name of newly created endpoint
print(predictor.endpoint_name)
```

**PyTorch**

```python
from sagemaker.pytorch import PyTorchModel

# Create SageMaker model and deploy an endpoint
sm_pytorch_compiled_model = PyTorchModel(
    model_data='insert S3 path of compiled PyTorch model archive',
    role='AmazonSageMaker-ExecutionRole',
    entry_point='inference.py',
    source_dir='code',
    framework_version='1.4.0',
    py_version='py3',
    image_uri='insert appropriate ECR Image URI for PyTorch',
    env={'MMS_DEFAULT_RESPONSE_TIMEOUT': '500'},
)

# Replace the example instance_type below to your preferred instance_type
predictor = sm_pytorch_compiled_model.deploy(initial_instance_count = 1, instance_type = 'ml.p3.2xlarge')

# Print the name of newly created endpoint
print(predictor.endpoint_name)
```

**Note**

The AmazonSageMakerFullAccess and AmazonS3ReadOnlyAccess policies must be attached to the AmazonSageMaker-ExecutionRole IAM role.
If you compiled your model using Boto3, SageMaker console, or the CLI for TensorFlow

Construct a TensorFlowModel object, then call deploy:

```python
role='AmazonSageMaker-ExecutionRole'
model_path='S3 path for model file'
framework_image='inference container arn'

.tf_model = TensorFlowModel(model_data=model_path,
framework_version='1.15.3',
role=role,
image_uri=framework_image)

instance_type='ml.c5.xlarge'
predictor = tf_model.deploy(instance_type=instance_type,
initial_instance_count=1)
```

See Deploying directly from model artifacts for more information.

You can select a Docker image Amazon ECR URI that meets your needs from this list.

For more information on how to construct a TensorFlowModel object, see the SageMaker SDK.

Note
Your first inference request might have high latency if you deploy your model on a GPU. This is because an optimized compute kernel is made on the first inference request. We recommend that you make a warm-up file of inference requests and store that alongside your model file before sending it off to a TFX. This is known as “warming up” the model.

The following code snippet demonstrates how to produce the warm-up file for image classification example in the prerequisites section:

```python
import tensorflow as tf
from tensorflow_serving.apis import classification_pb2
from tensorflow_serving.apis import inference_pb2
from tensorflow_serving.apis import model_pb2
from tensorflow_serving.apis import predict_pb2
from tensorflow_serving.apis import prediction_log_pb2
from tensorflow_serving.apis import regression_pb2
import numpy as np

with tf.python_io.TFRecordWriter("tf_serving_warmup_requests") as writer:
    img = np.random.uniform(0, 1, size=[224, 224, 3]).astype(np.float32)
    img = np.expand_dims(img, axis=0)
    test_data = np.repeat(img, 1, axis=0)
    request = predict_pb2.PredictRequest()
    request.model_spec.name = 'compiled_models'
    request.model_spec.signature_name = 'serving_default'
    request.inputs['Placeholder:0'].CopyFrom(tf.compat.v1.make_tensor_proto(test_data,
shape=test_data.shape, dtype=tf.float32))
    log = prediction_log_pb2.PredictionLog(
        predict_log=prediction_log_pb2.PredictLog(request=request))
    writer.write(log.SerializeToString())
```

For more information on how to “warm up” your model, see the TensorFlow TFX page.

Deploy a Compiled Model Using Boto3

You must satisfy the prerequisites section if the model was compiled using AWS SDK for Python (Boto3), AWS CLI, or the Amazon SageMaker console. Follow the steps below to create and deploy a SageMaker Neo-compiled model using Amazon Web Services SDK for Python (Boto3).

Topics
- Deploy the Model (p. 1253)
Deploy the Model

After you have satisfied the prerequisites, use the create_model, create_enpoint_config, and create_endpoint APIs.

The following example shows how to use these APIs to deploy a model compiled with Neo:

```python
import boto3
client = boto3.client('sagemaker')

# create sagemaker model
create_model_api_response = client.create_model(
    ModelName='my-sagemaker-model',
    PrimaryContainer={
        'Image': '<insert the ECR Image URI>',
        'ModelDataUrl': 's3://path/to/model/artifact/model.tar.gz',
        'Environment': {},
    },
    ExecutionRoleArn='ARN for AmazonSageMaker-ExecutionRole'
)
print("create_model API response", create_model_api_response)

# create sagemaker endpoint config
create_endpoint_config_api_response = client.create_endpoint_config(
    EndpointConfigName='sagemaker-neomxnet-endpoint-configuration',
    ProductionVariants=[
        {'VariantName': '<provide your variant name>',
         'ModelName': 'my-sagemaker-model',
         'InitialInstanceCount': 1,
         'InstanceType': '<provide your instance type here>'},
    ],
)
print("create_endpoint_config API response", create_endpoint_config_api_response)

# create sagemaker endpoint
create_endpoint_api_response = client.create_endpoint(
    EndpointName='provide your endpoint name',
    EndpointConfigName='sagemaker-neomxnet-endpoint-configuration',
)
print("create_endpoint API response", create_endpoint_api_response)
```

**Note**
The AmazonSageMakerFullAccess and AmazonS3ReadOnlyAccess policies must be attached to the AmazonSageMaker-ExecutionRole IAM role.

For full syntax of create_model, create_endpint_config, and create_endpoint APIs, see create_model, create_endpoint_config, and create_endpoint, respectively.

If you did not train your model using SageMaker, specify the following environment variables:

**MXNet and PyTorch**

"Environment": {
    "SAGEMAKER_PROGRAM": "inference.py",}
If you trained your model using SageMaker, specify the environment variable `SAGEMAKER_SUBMIT_DIRECTORY` as the full Amazon S3 bucket URI that contains the training script.

### Deploy a Compiled Model Using the AWS CLI

You must satisfy the prerequisites section if the model was compiled using AWS SDK for Python (Boto3), AWS CLI, or the Amazon SageMaker console. Follow the steps below to create and deploy a SageMaker Neo-compiled model using the AWS CLI.

**Topics**
- Deploy the Model (p. 1254)

**Deploy the Model**

After you have satisfied the prerequisites, use the `create-model`, `create-endpoint-config`, and `create-endpoint` AWS CLI commands. The following steps explain how to use these commands to deploy a model compiled with Neo:

**Create a Model**

From Neo Inference Container Images, select the inference image URI and then use create-model API to create a SageMaker model. You can do this with two steps:

1. Create a `create_model.json` file. Within the file, specify the name of the model, the image URI, the path to the model.tar.gz file in your Amazon S3 bucket, and your SageMaker execution role:

   ```json
   {
     "ModelName": "insert model name",
     "PrimaryContainer": {
       "Image": "insert the ECR Image URI",
       "ModelDataUrl": "insert S3 archive URL",
       "Environment": { "See details below" }
     },
     "ExecutionRoleArn": "ARN for AmazonSageMaker-ExecutionRole"
   }
   ```

   If you trained your model using SageMaker, specify the following environment variable:

   ```json
   "Environment": {
     "SAGEMAKER_SUBMIT_DIRECTORY" : "[Full S3 path for *.tar.gz file containing the training script]"
   }
   ```

1254
If you did not train your model using SageMaker, specify the following environment variables: MXNet and PyTorch

```
"Environment": {
  "SAGEMAKER_PROGRAM": "inference.py",
  "SAGEMAKER_SUBMIT_DIRECTORY": "/opt/ml/model/code",
  "SAGEMAKER_CONTAINER_LOG_LEVEL": "20",
  "SAGEMAKER_REGION": "insert your region",
  "MMS_DEFAULT_RESPONSE_TIMEOUT": "500"
}
```

TensorFlow

```
"Environment": {
  "SAGEMAKER_PROGRAM": "inference.py",
  "SAGEMAKER_SUBMIT_DIRECTORY": "/opt/ml/model/code",
  "SAGEMAKER_CONTAINER_LOG_LEVEL": "20",
  "SAGEMAKER_REGION": "insert your region"
}
```

**Note**
The AmazonSageMakerFullAccess and AmazonS3ReadOnlyAccess policies must be attached to the AmazonSageMaker-ExecutionRole IAM role.

2. Run the following command:

```
aws sagemaker create-model --cli-input-json file://create_model.json
```

For the full syntax of the create-model API, see create-model.

**Create an Endpoint Configuration**

After creating a SageMaker model, create the endpoint configuration using the create-endpoint-config API. To do this, create a JSON file with your endpoint configuration specifications. For example, you can use the following code template and save it as create_config.json:

```
{
  "EndpointConfigName": "<provide your endpoint config name>",
  "ProductionVariants": [
    {
      "VariantName": "<provide your variant name>",
      "ModelName": "my-sagemaker-model",
      "InitialInstanceCount": 1,
      "InstanceType": "<provide your instance type here>",
      "InitialVariantWeight": 1.0
    }
  ]
}
```

Now run the following AWS CLI command to create your endpoint configuration:

```
aws sagemaker create-endpoint-config --cli-input-json file://create_config.json
```

For the full syntax of the create-endpoint-config API, see create-endpoint-config.
Create an Endpoint

After you have created your endpoint configuration, create an endpoint using the create-endpoint API:

```
aws sagemaker create-endpoint --endpoint-name '<provide your endpoint name>' --endpoint-config-name '<insert your endpoint config name>'
```

For the full syntax of the create-endpoint API, see `create-endpoint`.

Deploy a Compiled Model Using the Console

You must satisfy the prerequisites section if the model was compiled using AWS SDK for Python (Boto3), the AWS CLI, or the Amazon SageMaker console. Follow the steps below to create and deploy a SageMaker Neo-compiled model using the Amazon SageMaker console.

Topics
- Deploy the Model (p. 1256)

Deploy the Model

After you have satisfied the prerequisites, use the following steps to deploy a model compiled with Neo:

1. Choose Models, and then choose Create models from the Inference group. On the Create model page, complete the Model name, IAM role, and VPC fields (optional), if needed.

2. To add information about the container used to deploy your model, choose Add container container, then choose Next. Complete the Container input options, Location of inference code image, and Location of model artifacts, and optionally, Container host name, and Environmental variables fields.
3. To deploy Neo-compiled models, choose the following:

- **Container input options**: Choose *Provide model artifacts and inference image*.

- **Location of inference code image**: Choose the inference image URI from Neo Inference Container Images, depending on the AWS Region and kind of application.

- **Location of model artifact**: Enter the Amazon S3 bucket URI of the compiled model artifact generated by the Neo compilation API.

- **Environment variables**:
  - Leave this field blank for *SageMaker XGBoost*.
  - If you trained your model using SageMaker, specify the environment variable `SAGEMAKER_SUBMIT_DIRECTORY` as the Amazon S3 bucket URI that contains the training script.
  - If you did not train your model using SageMaker, specify the following environment variables:

<table>
<thead>
<tr>
<th>Key</th>
<th>Values for MXNet and PyTorch</th>
<th>Values TensorFlow</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAGEMAKER_PROGRAM</td>
<td>inference.py</td>
<td>inference.py</td>
</tr>
<tr>
<td>SAGEMAKER_SUBMIT_DIRECTORY</td>
<td>/opt/ml/model/code</td>
<td>/opt/ml/model/code</td>
</tr>
<tr>
<td>SAGEMAKER_CONTAINER_LOG_LEVEL</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>Key</td>
<td>Values for MXNet and PyTorch</td>
<td>Values TensorFlow</td>
</tr>
<tr>
<td>---------------------</td>
<td>-------------------------------</td>
<td>-------------------</td>
</tr>
<tr>
<td>SAGEMAKER_REGION</td>
<td>&lt;your region&gt;</td>
<td>&lt;your region&gt;</td>
</tr>
<tr>
<td>MMS_DEFAULT_RESPONSE_TIMEOUT</td>
<td>500</td>
<td>Leave this field blank for TF</td>
</tr>
</tbody>
</table>

4. Confirm that the information for the containers is accurate, and then choose Create model. On the Create model landing page, choose Create endpoint.

5. In Create and configure endpoint diagram, specify the Endpoint name. For Attach endpoint configuration, choose Create a new endpoint configuration.

6. In New endpoint configuration page, specify the Endpoint configuration name.
7. Choose **Edit** next to the name of the model and specify the correct **Instance type** on the **Edit Production Variant** page. It is imperative that the **Instance type** value match the one specified in your compilation job.
8. Choose Save.

9. On the New endpoint configuration page, choose Create endpoint configuration, and then choose Create endpoint.

Request Inferences from a Deployed Service

If you have followed instructions in Deploy a Model (p. 1245), you should have a SageMaker endpoint set up and running. Regardless of how you deployed your Neo-compiled model, there are three ways you can submit inference requests:

**Topics**

- Request Inferences from a Deployed Service (Amazon SageMaker SDK) (p. 1260)
- Request Inferences from a Deployed Service (Boto3) (p. 1262)
- Request Inferences from a Deployed Service (AWS CLI) (p. 1263)

**Request Inferences from a Deployed Service (Amazon SageMaker SDK)**

Use the following the code examples to request inferences from your deployed service based on the framework you used to train your model. The code examples for the different frameworks are similar. The main difference is that TensorFlow requires application/json as the content type.
PyTorch and MXNet

If you are using **PyTorch v1.4 or later** or **MXNet 1.7.0 or later** and you have an Amazon SageMaker endpoint **InService**, you can make inference requests using the **predictor package of the SageMaker SDK for Python**.

**Note**
The API varies based on the SageMaker SDK for Python version:

- For version 1.x, use the **RealTimePredictor** and **Predict API**.
- For version 2.x, use the **Predictor** and the **Predict API**.

The following code example shows how to use these APIs to send an image for inference:

**SageMaker Python SDK v1.x**

```python
from sagemaker.predictor import RealTimePredictor
endpoint = 'insert name of your endpoint here'

# Read image into memory
payload = None
with open("image.jpg", 'rb') as f:
    payload = f.read()

predictor = RealTimePredictor(endpoint=endpoint, content_type='application/x-image')
inference_response = predictor.predict(data=payload)
print (inference_response)
```

**SageMaker Python SDK v2.x**

```python
from sagemaker.predictor import Predictor
endpoint = 'insert name of your endpoint here'

# Read image into memory
payload = None
with open("image.jpg", 'rb') as f:
    payload = f.read()

predictor = Predictor(endpoint)
inference_response = predictor.predict(data=payload)
print (inference_response)
```

**TensorFlow**

The following code example shows how to use the SageMaker Python SDK API to send an image for inference:

```python
from sagemaker.predictor import Predictor
from PIL import Image
import numpy as np
import json
endpoint = 'insert the name of your endpoint here'

# Read image into memory
image = Image.open(input_file)
batch_size = 1
image = np.asarray(image.resize((224, 224)))
```
image = image / 128 - 1
image = np.concatenate([[image[np.newaxis, :, :]] * batch_size]
body = json.dumps({"instances": image.tolist()})
predictor = Predictor(endpoint)
inference_response = predictor.predict(data=body)
print(inference_response)

Request Inferences from a Deployed Service (Boto3)

You can submit inference requests using SageMaker SDK for Python (Boto3) client and `invoke_endpoint()` API once you have an SageMaker endpoint InService. The following code example shows how to send an image for inference:

PyTorch and MXNet

```python
import boto3
import json

endpoint = 'insert name of your endpoint here'
runtime = boto3.Session().client('sagemaker-runtime')

# Read image into memory
with open(image, 'rb') as f:
    payload = f.read()
# Send image via InvokeEndpoint API
response = runtime.invoke_endpoint(EndpointName=endpoint, ContentType='application/x-image', Body=payload)

# Unpack response
result = json.loads(response['Body'].read().decode())
```

TensorFlow

For TensorFlow submit an input with `application/json` for the content type.

```python
from PIL import Image
import numpy as np
import json
import boto3

client = boto3.client('sagemaker-runtime')
input_file = 'path/to/image'
image = Image.open(input_file)
batch_size = 1
image = np.asarray(image.resize((224, 224)))
image = image / 128 - 1
image = np.concatenate([[image[np.newaxis, :, :]] * batch_size]
body = json.dumps({"instances": image.tolist()})
io_c_predictor_endpoint_name = 'insert name of your endpoint here'
content_type = 'application/json'
io_c_response = client.invoke_endpoint(
    EndpointName=ioc_predictor_endpoint_name,
    Body=body,
    ContentType=content_type
)
```

XGBoost

For an XGBoost application, you should submit a CSV text instead:
import boto3
import json

endpoint = 'insert your endpoint name here'
runtime = boto3.Session().client('sagemaker-runtime')

csv_text = '1,-1.0,1.0,1.5,2.6'
# Send CSV text via InvokeEndpoint API
response = runtime.invoke_endpoint(EndpointName=endpoint, ContentType='text/csv', Body=csv_text)
# Unpack response
result = json.loads(response['Body'].read().decode())

Note that BYOM allows for a custom content type. For more information, see runtime_InvokeEndpoint.

Request Inferences from a Deployed Service (AWS CLI)

Inference requests can be made with the sagemaker-runtime invoke-endpoint once you have an Amazon SageMaker endpoint InService. You can make inference requests with the AWS Command Line Interface (AWS CLI). The following example shows how to send an image for inference:

```
aws sagemaker-runtime invoke-endpoint --endpoint-name 'insert name of your endpoint here' --body fileb://image.jpg --content-type=application/x-image output_file.txt
```

An output_file.txt with information about your inference requests is made if the inference was successful.

For TensorFlow submit an input with application/json as the content type.

```
aws sagemaker-runtime invoke-endpoint --endpoint-name 'insert name of your endpoint here' --body fileb://input.json --content-type=application/json output_file.txt
```

Inference Container Images

Based on your use case, replace the highlighted portion in the inference image URI template provided below with appropriate values.

Amazon SageMaker XGBoost

```
aws_account_id.dkr.ecr.aws_region.amazonaws.com/xgboost-neo:latest
```

Replace aws_account_id from the table at the end of this page based on the aws_region you used.

TensorFlow

CPU or GPU instance types

```
aws_account_id.dkr.ecr.aws_region.amazonaws.com/sagemaker-inference-tensorflow:fx_version-instance_type-py3
```

Replace aws_account_id from the table at the end of this page based on the aws_region you used.
Replace `fx_version` with 1.15.3.

Replace `instance_type` with either `cpu` or `gpu`.

### Inferentia instance types

```
aws_account_id.dkr.ecr.aws_region.amazonaws.com/sagemaker-neo-tensorflow:fx_version-instance_type-py3
```

Replace `aws_account_id` from the table at the end of this page based on the `aws_region` you used. Note that for instance type `inf` only `us-east-1` and `us-west-2` are supported.

Replace `fx_version` with 1.15.0.

Replace `instance_type` with `inf`.

### MXNet

**CPU or GPU instance types**

```
aws_account_id.dkr.ecr.aws_region.amazonaws.com/sagemaker-inference-mxnet:fx_version-instance_type-py3
```

Replace `aws_account_id` from the table at the end of this page based on the `aws_region` you used.

Replace `fx_version` with 1.7.0.

Replace `instance_type` with either `cpu` or `gpu`.

### Inferentia instance types

```
aws_account_id.dkr.ecr.aws_region.amazonaws.com/sagemaker-neo-mxnet:fx_version-instance_type-py3
```

Replace `aws_region` with either `us-east-1` or `us-west-2`.

Replace `aws_account_id` from the table at the end of this page based on the `aws_region` you used.

Replace `fx_version` with 1.5.1.

Replace `instance_type` with `inf`.

### PyTorch

**CPU or GPU instance types**

```
aws_account_id.dkr.ecr.aws_region.amazonaws.com/sagemaker-inference-pytorch:fx_version-instance_type-py3
```

Replace `aws_account_id` from the table at the end of this page based on the `aws_region` you used.

Replace `fx_version` with 1.4.0.
Replace `instance_type` with either `cpu` or `gpu`.

Inferentia instance types

```
aws_account_id.dkr.ecr.aws_region.amazonaws.com/sagemaker-neo-pytorch:fx_version-instance_type-py3
```

Replace `aws_region` with either `us-east-1` or `us-west-2`.

Replace `aws_account_id` from the table at the end of this page based on the `aws_region` you used.

Replace `fx_version` with `1.5.1`.

Replace `instance_type` with `inf`.

The following table maps `aws_account_id` with `aws_region`. Use this table to find the correct inference image URI you need for your application.

<table>
<thead>
<tr>
<th><code>aws_account_id</code></th>
<th><code>aws_region</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>785573368785</td>
<td>us-east-1</td>
</tr>
<tr>
<td>007439368137</td>
<td>us-east-2</td>
</tr>
<tr>
<td>710691900526</td>
<td>us-west-1</td>
</tr>
<tr>
<td>301217895009</td>
<td>us-west-2</td>
</tr>
<tr>
<td>802834080501</td>
<td>eu-west-1</td>
</tr>
<tr>
<td>205493899709</td>
<td>eu-west-2</td>
</tr>
<tr>
<td>254080097072</td>
<td>eu-west-3</td>
</tr>
<tr>
<td>601324751636</td>
<td>eu-north-1</td>
</tr>
<tr>
<td>966458181534</td>
<td>eu-south-1</td>
</tr>
<tr>
<td>746233611703</td>
<td>eu-central-1</td>
</tr>
<tr>
<td>110948597952</td>
<td>ap-east-1</td>
</tr>
<tr>
<td>763008648453</td>
<td>ap-south-1</td>
</tr>
<tr>
<td>941853720454</td>
<td>ap-northeast-1</td>
</tr>
<tr>
<td>151534178276</td>
<td>ap-northeast-2</td>
</tr>
<tr>
<td>324986816169</td>
<td>ap-southeast-1</td>
</tr>
<tr>
<td>355873309152</td>
<td>ap-southeast-2</td>
</tr>
<tr>
<td>474822919863</td>
<td>cn-northwest-1</td>
</tr>
<tr>
<td>472730292857</td>
<td>cn-north-1</td>
</tr>
<tr>
<td>756306329178</td>
<td>sa-east-1</td>
</tr>
<tr>
<td>464438896020</td>
<td>ca-central-1</td>
</tr>
</tbody>
</table>
Edge Devices

Amazon SageMaker Neo provides compilation support for popular machine learning frameworks. You can deploy your Neo-compiled edge devices such as the Raspberry Pi 3, Texas Instruments’ Sitara, Jetson TX1, and more. For a full list of supported frameworks and edge devices, see Supported Frameworks, Devices, Systems, and Architectures.

You must configure your edge device so that it can use AWS services. One way to do this is to install DLR and Boto3 to your device. To do this, you must set up the authentication credentials. See Boto3 AWS Configuration for more information. Once your model is compiled and your edge device is configured, you can download the model from Amazon S3 to your edge device. From there, you can use the Deep Learning Runtime (DLR) to read the compiled model and make inferences.

For first-time users, we recommend you check out the Getting Started guide. This guide walks you through how to set up your credentials, compile a model, deploy your model to a Raspberry Pi 3, and make inferences on images.

Topics
- Supported Frameworks, Devices, Systems, and Architectures (p. 1266)
- Deploy Models (p. 1269)
- Getting Started with Neo on Edge Devices (p. 1270)

Supported Frameworks, Devices, Systems, and Architectures

Amazon SageMaker Neo supports the following frameworks and edge devices.

### Supported Frameworks

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>MXNet</td>
<td>1.6.0</td>
<td>Supports 1.6.0 or earlier</td>
<td>Image Classification, Object Detection, Semantic Segmentation, Pose Estimation, Activity Recognition</td>
<td>One symbol file (.json) and one parameter file (.params)</td>
<td>GluonCV v0.8.0</td>
</tr>
<tr>
<td>ONNX</td>
<td>1.5.0</td>
<td>Supports 1.5.0 or earlier</td>
<td>Image Classification, SVM</td>
<td>One model file (.onnx)</td>
<td></td>
</tr>
<tr>
<td>Framework</td>
<td>Version</td>
<td>Model Version</td>
<td>Models</td>
<td>Model Formats (packaged in *tar.gz)</td>
<td>Toolkits</td>
</tr>
<tr>
<td>-----------</td>
<td>---------</td>
<td>---------------</td>
<td>--------</td>
<td>------------------------------------</td>
<td>----------</td>
</tr>
<tr>
<td>Keras</td>
<td>2.2.4</td>
<td>Supports 2.2.4 or earlier</td>
<td>Image Classification</td>
<td>One model definition file (.h5)</td>
<td></td>
</tr>
<tr>
<td>PyTorch</td>
<td>1.4.0</td>
<td>Supports 1.4.0 or earlier</td>
<td>Image Classification</td>
<td>One model definition file (.pth)</td>
<td></td>
</tr>
<tr>
<td>TensorFlow</td>
<td>1.15.0</td>
<td>Supports 1.15.0 or earlier</td>
<td>Image Classification</td>
<td>*For saved models, one .pb or one .pbtxt file and a variables directory that contains variables *For frozen models, only one .pb or .pbtxt file</td>
<td></td>
</tr>
<tr>
<td>TensorFlow-Lite</td>
<td>1.13.1</td>
<td>Supports 1.13.1 or earlier</td>
<td>Image Classification, Object Detection</td>
<td>One model definition flatbuffer file (.tflite)</td>
<td></td>
</tr>
<tr>
<td>XGBoost</td>
<td>0.9</td>
<td>Supports 0.9 or earlier</td>
<td>Decision Trees</td>
<td>One XGBoost model file (.model) where the number of nodes in a tree is less than 2^31</td>
<td></td>
</tr>
<tr>
<td>DARKNET</td>
<td></td>
<td></td>
<td>Image Classification, Object Detection</td>
<td>nOne config (.cfg) file and one weights (.weights) file</td>
<td></td>
</tr>
</tbody>
</table>

### Devices

You can select a device using the dropdown list in the console or API. You can choose from one of the following edge devices:

<table>
<thead>
<tr>
<th>Device List</th>
<th>System on a Chip (SoC)</th>
<th>Operating System</th>
<th>Architecture</th>
<th>Accelerator</th>
<th>Compiler Options</th>
</tr>
</thead>
<tbody>
<tr>
<td>jetson_tx1</td>
<td>LINUX</td>
<td>ARM64</td>
<td>NVIDIA</td>
<td>{'gpu-code': 'sm_53', 'trt-ver': '6.0.1', 'cuda-ver': '10.0'}</td>
<td></td>
</tr>
<tr>
<td>Device List</td>
<td>System on a Chip (SoC)</td>
<td>Operating System</td>
<td>Architecture</td>
<td>Accelerator</td>
<td>Compiler Options</td>
</tr>
<tr>
<td>-------------</td>
<td>-----------------------</td>
<td>------------------</td>
<td>--------------</td>
<td>-------------</td>
<td>-----------------</td>
</tr>
<tr>
<td>jetson_tx2</td>
<td></td>
<td>LINUX</td>
<td>ARM64</td>
<td>NVIDIA</td>
<td>{'gpu-code': 'sm_62', 'trt-ver': '6.0.1', 'cuda-ver': '10.0'}</td>
</tr>
<tr>
<td>jetson.nano</td>
<td></td>
<td>LINUX</td>
<td>ARM64</td>
<td>NVIDIA</td>
<td>{'gpu-code': 'sm_53', 'trt-ver': '5.0.6', 'cuda-ver': '10.0'}</td>
</tr>
<tr>
<td>jetson_xavier</td>
<td></td>
<td>LINUX</td>
<td>ARM64</td>
<td>NVIDIA</td>
<td>{'gpu-code': 'sm_72', 'trt-ver': '5.1.6', 'cuda-ver': '10.0'}</td>
</tr>
<tr>
<td>rasp3b</td>
<td>ARM A56</td>
<td>LINUX</td>
<td>ARM_EABIHF</td>
<td></td>
<td>{'mattr': ['+neon']}</td>
</tr>
<tr>
<td>rasp4</td>
<td>ARM A72</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>imx8qm</td>
<td>NXP imx8</td>
<td>LINUX</td>
<td>ARM64</td>
<td></td>
<td></td>
</tr>
<tr>
<td>depleens</td>
<td>Intel Atom</td>
<td>LINUX</td>
<td>X86_64</td>
<td>INTEL_GRAPHICS</td>
<td></td>
</tr>
<tr>
<td>rk3399</td>
<td></td>
<td>LINUX</td>
<td>ARM64</td>
<td>MALI</td>
<td></td>
</tr>
<tr>
<td>rk3288</td>
<td></td>
<td>LINUX</td>
<td>ARM_EABIHF</td>
<td>MALI</td>
<td></td>
</tr>
<tr>
<td>aisage</td>
<td></td>
<td>LINUX</td>
<td>ARM64</td>
<td>MALI</td>
<td></td>
</tr>
<tr>
<td>sbe_c</td>
<td></td>
<td>LINUX</td>
<td>x86_64</td>
<td></td>
<td>{'mcpu': 'core-avx2'}</td>
</tr>
<tr>
<td>qcs605</td>
<td></td>
<td>ANDROID</td>
<td>ARM64</td>
<td>MALI</td>
<td>{'ANDROID_PLATFORM': 27}</td>
</tr>
<tr>
<td>qcs603</td>
<td></td>
<td>ANDROID</td>
<td>ARM64</td>
<td>MALI</td>
<td>{'ANDROID_PLATFORM': 27}</td>
</tr>
</tbody>
</table>

**Systems and Architectures**

The following look-up tables provide information regarding available operating systems and architectures for Neo model compilation jobs.

**Linux**

<table>
<thead>
<tr>
<th></th>
<th>X86_64</th>
<th>X86</th>
<th>ARM64</th>
<th>ARM_EABIHF</th>
<th>ARM_EABI</th>
</tr>
</thead>
<tbody>
<tr>
<td>No accelerator</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>(CPU)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nvidia GPU</td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Deploy Models

You can deploy the compute module to resource-constrained edge devices by: downloading the compiled model from Amazon S3 to your device and using DLR, or you can use AWS IoT Greengrass.

Before moving on, make sure your edge device must be supported by SageMaker Neo. See, Supported Frameworks, Devices, Systems, and Architectures to find out what edge devices are supported. Make sure that you specified your target edge device when you submitted the compilation job, see Use Neo to Compile a Model.

### Deploy a Compiled Model (DLR)

DLR is a compact, common runtime for deep learning models and decision tree models. DLR uses the TVM runtime, Treelite runtime, NVIDIA TensorRT™, and can include other hardware-specific runtimes. DLR provides unified Python/C++ APIs for loading and running compiled models on various devices.

You can install latest release of DLR package using the following pip command:

```
pip install dlr
```

For installation of DLR on GPU targets or non-x86 edge devices, please refer to Releases for prebuilt binaries, or Installing DLR for building DLR from source. For example, to install DLR for Raspberry Pi 3, you can use:

```
```
Deploy a Model (AWS IoT Greengrass)

AWS IoT Greengrass extends cloud capabilities to local devices. It enables devices to collect and analyze data closer to the source of information, react autonomously to local events, and communicate securely with each other on local networks. With AWS IoT Greengrass, you can perform machine learning inference at the edge on locally generated data using cloud-trained models. Currently, you can deploy models on to all AWS IoT Greengrass devices based on ARM Cortex-A, Intel Atom, and Nvidia Jetson series processors. For more information on deploying a Lambda inference application to perform machine learning inferences with AWS IoT Greengrass, see How to configure optimized machine learning inference using the AWS Management Console.

Getting Started with Neo on Edge Devices

This guide to getting started with Amazon SageMaker Neo shows you how to compile a model, set up your device, and make inferences on your device. Most of the code examples use Boto3. We provide commands using AWS CLI where applicable, as well as instructions on how to satisfy prerequisites for Neo.

Note

You can run the following code snippets on your local machine, within a SageMaker notebook, within SageMaker Studio, or (depending on your edge device) on your edge device. The setup is similar; however, there are two main exceptions if you run this guide within a SageMaker notebook instance or SageMaker Studio session:

- You do not need to install Boto3.
- You do not need to add the `AmazonSageMakerFullAccess` IAM policy.

This guide assumes you are running the following instructions on your edge device.

Prerequisites

1. **Install Boto3**

   If you are running these commands on your edge device, you must install the AWS SDK for Python (Boto3). Within a Python environment (preferably a virtual environment), run the following locally on your edge device's terminal or within a Jupyter notebook instance:

   **Terminal**

   ```bash
   $ pip install boto3
   ```

   **Jupyter Notebook**

   ```shell
   !pip install boto3
   ```

2. **Set Up AWS Credentials**

   You need to set up Amazon Web Services credentials on your device in order to run SDK for Python (Boto3). By default, the AWS credentials should be stored in the file `~/.aws/credentials` on your edge device. Within the credentials file, you should see two environment variables: `aws_access_key_id` and `aws_secret_access_key`.

   In your terminal, run:

   ```bash
   $ more ~/.aws/credentials
   [default]
   ```
aws_access_key_id = YOUR_ACCESS_KEY
aws_secret_access_key = YOUR_SECRET_KEY

The AWS General Reference Guide has instructions on how to get the necessary aws_access_key_id and aws_secret_access_key. For more information on how to set up credentials on your device, see the Boto3 documentation.

3. Set up an IAM Role and attach policies.

Neo needs access to your S3 bucket URI. Create an IAM role that can run SageMaker and has permission to access the S3 URI. You can create an IAM role either by using SDK for Python (Boto3), the console, or the AWS CLI. The following example illustrates how to create an IAM role using SDK for Python (Boto3):

```python
import boto3
AWS_REGION = 'aws-region'

# Create an IAM client to interact with IAM
iam_client = boto3.client('iam', region_name=AWS_REGION)
role_name = 'role-name'

# Create a dictionary describing the IAM policy you are attaching. This policy is used to create a new IAM role.
policy = {
    'Statement': [
        {
            'Action': 'sts:AssumeRole',
            'Effect': 'Allow',
            'Principal': {'Service': 'sagemaker.amazonaws.com'},
        },
    ],
    'Version': '2012-10-17'
}

# Create a new IAM role using the policy you defined above:
new_role = iam_client.create_role(
    AssumeRolePolicyDocument=json.dumps(policy),
    Path='/',
    RoleName=role_name,
)

# You need to know what your Amazon Resource Name (ARN) is when you create a compilation job in a later step, so store it in a variable as well.
role_arn = new_role['Role']['Arn']

# Now that you have created a new role, attach the permissions it needs to interact with Amazon SageMaker and Amazon S3:
iam_client.attach_role_policy(
    RoleName=role_name,
    PolicyArn='arn:aws:iam::aws:policy/AmazonSageMakerFullAccess'
)
```
4. **Create an Amazon S3 bucket to store your model artifacts**

SageMaker Neo will access your model artifacts from Amazon S3

**Boto3**

```python
# Create an S3 client
s3_client = boto3.client('s3', region_name=AWS_REGION)

# Name buckets
bucket='name-of-your-bucket'

# Check if bucket exists
if boto3.resource('s3').Bucket(bucket) not in boto3.resource('s3').buckets.all():
    s3_client.create_bucket(
        Bucket=bucket,
        CreateBucketConfiguration={
            'LocationConstraint': AWS_REGION
        }
    )
else:
    print(f'Bucket {bucket} already exists. No action needed.')
```

**CLI**

```bash
# Create an S3 bucket
$ aws s3 mb s3://name-of-your-bucket --region specify-your-region

# Check your bucket exists
$ aws s3 ls s3://name-of-your-bucket/
```

5. **Train a machine learning model**

See [Train a Model with Amazon SageMaker](#) for more information on how to train a machine learning model using Amazon SageMaker. You can optionally upload your locally trained model directly into an Amazon S3 URI bucket.

**Note**

Make sure the model is correctly formatted depending on the framework you used. See [What input data shapes does SageMaker Neo expect?](#)

If you do not have a model yet, use the curl command to get a local copy of the coco_ssd_mobilenet model from TensorFlow's website. The model you just copied is an object detection model trained from the COCO dataset. Type the following into your AWS CLI or Jupyter notebook or as part of a Python script:

```bash
model_zip_filename = './coco_ssd_mobilenet_v1_1.0.zip'
!curl http://storage.googleapis.com/download.tensorflow.org/models/tflite/coco_ssd_mobilenet_v1_1.0_quant_2018_06_29.zip --output {model_zip_filename}
```

Note that this particular example was packaged in a .zip file. Unzip this file and repackage it as a compressed tarfile (.tar.gz) before using it in later steps. Type the following in your Jupyter notebook or as part of a Python script:
# Extract model from zip file

```bash
unzip -u {model_zip_filename}
```

```python
model_filename = 'detect.tflite'
model_name = model_filename.split('.')[0]
```

# Compress model into .tar.gz so SageMaker Neo can use it

```bash
!tar -czf {model_tar} {model_filename}
```

6. **Upload trained model to an S3 bucket**

Once you have trained your machine learning mode, store it in an S3 bucket.

**Boto3**

```python
bucket='replace-with-your-bucket-name'

# Upload model
s3_client.upload_file(filename=model_filename, bucket=bucket, key=model_filename)
```

**CLI**

Replace your-model-filename and your-S3-bucket with the name of your S3 bucket.

```bash
$ aws s3 cp your-model-filename s3://your-S3-bucket
```

### Step 1: Compile the Model

Once you have satisfied the Prerequisites, you can compile your model with Amazon SageMaker Neo. You can compile your model using the AWS CLI, the console or the Amazon Web Services SDK for Python (Boto3), see Use Neo to Compile a Model. In this example, you will compile your model with Boto3.

To compile a model, SageMaker Neo requires the following information:

1. **The Amazon S3 bucket URI where you stored the trained model.**

   If you followed the prerequisites, the name of your bucket is stored in a variable named `bucket`. The following code snippet shows how to list all of your buckets using the AWS CLI:

   ```bash
   # aws s3 ls
   
   For example:
   
   ```bash
   # aws s3 ls
   2020-11-02 17:08:50 bucket
   ```

2. **The Amazon S3 bucket URI where you want to save the compiled model.**

   The code snippet below concatenates your Amazon S3 bucket URI with the name of an output directory called `output`:

   ```python
   s3_output_location = f's3://{bucket}/output'
   ```

3. **The machine learning framework you used to train your model.**
Define the framework you used to train your model.

```
framework = 'framework-name'
```

For example, if you wanted to compile a model that was trained using TensorFlow, you could either use `tflite` or `tensorflow`. Use `tflite` if you want to use a lighter version of TensorFlow that uses less storage memory.

```
framework = 'tflite'
```

For a complete list of Neo-supported frameworks, see Supported Frameworks, Devices, Systems, and Architectures.

4. **The shape of your model's input.**

Neo requires the name and shape of your input tensor. The name and shape are passed in as key-value pairs. `value` is a list of the integer dimensions of an input tensor and `key` is the exact name of an input tensor in the model.

```
data_shape = '{"name": [tensor-shape]}'
```

For example:

```
data_shape = '{"normalized_input_image_tensor": [1, 300, 300, 3]}'
```

**Note**

Make sure the model is correctly formatted depending on the framework you used. See What input data shapes does SageMaker Neo expect? The key in this dictionary must be changed to the new input tensor's name.

5. **Either the name of the target device to compile for or the general details of the hardware platform**

```
target_device = 'target-device-name'
```

For example, if you want to deploy to a Raspberry Pi 3, use:

```
target_device = 'rasp3b'
```

You can find the entire list of supported edge devices in Supported Frameworks, Devices, Systems, and Architectures.

Now that you have completed the previous steps, you can submit a compilation job to Neo.

```
# Create a SageMaker client so you can submit a compilation job
sagemaker_client = boto3.client('sagemaker', region_name=AWS_REGION)

# Give your compilation job a name
compilation_job_name = 'getting-started-demo'
print(f'Compilation job for {compilation_job_name} started')
response = sagemaker_client.create_compilation_job(
    CompilationJobName=compilation_job_name,
    RoleArn=role_arn,
    InputConfig=
```
If you want additional information for debugging, include the following print statement:

```python
print(response)
```

If the compilation job is successful, your compiled model is stored in the output Amazon S3 bucket you specified earlier (s3_output_location). Download your compiled model locally:

```python
object_path = f'output/{model}-{target_device}.tar.gz'
neo_compiled_model = f'compiled-{model}.tar.gz'
s3_client.download_file(bucket, object_path, neo_compiled_model)
```

### Step 2: Set Up Your Device

You will need to install packages on your edge device so that your device can make inferences. You will also need to either install AWS IoT Greengrass core or Deep Learning Runtime (DLR). In this example, you will install packages required to make inferences for the `coco_ssd_mobilenet` object detection algorithm and you will use DLR.

1. **Install additional packages**

   In addition to Boto3, you must install certain libraries on your edge device. What libraries you install depends on your use case.

   For example, for the `coco_ssd_mobilenet` object detection algorithm you downloaded earlier, you need to install NumPy for data manipulation and statistics, PIL to load images, and Matplotlib to generate plots. You also need a copy of TensorFlow if you want to gauge the impact of compiling with Neo versus a baseline.

   ```bash
   !pip3 install numpy pillow tensorflow matplotlib
   ```

2. **Install inference engine on your device**
To run your Neo-compiled model, install the **Deep Learning Runtime (DLR)** on your device. DLR is a compact, common runtime for deep learning models and decision tree models. On x86_64 CPU targets running Linux, you can install the latest release of the DLR package using the following `pip` command:

```
!pip install dlr
```

For installation of DLR on GPU targets or non-x86 edge devices, refer to [Releases](#) for prebuilt binaries, or [Installing DLR](#) for building DLR from source. For example, to install DLR for Raspberry Pi 3, you can use:

```
```

### Step 3: Make Inferences on Your Device

In this example, you will use Boto3 to download the output of your compilation job onto your edge device. You will then import DLR, download an example images from the dataset, resize this image to match the model's original input, and then you will make a prediction.

1. **Download your compiled model from Amazon S3 to your device and extract it from the compressed tarfile.**

   ```
   # Download compiled model locally to edge device
   object_path = f'output/{model_name}-{target_device}.tar.gz'
   neo_compiled_model = f'compiled-{model_name}.tar.gz'
   s3_client.download_file(bucket_name, object_path, neo_compiled_model)

   # Extract model from .tar.gz so DLR can use it
   !mkdir ./dlr_model # make a directory to store your model (optional)
   !tar -xzvf ./compiled-detect.tar.gz --directory ./dlr_model
   ```

2. **Import DLR and an initialized `DLRModel` object.**

   ```
   import dlr
   device = 'cpu'
   model = dlr.DLRModel('./dlr_model', device)
   ```

3. **Download an image for inferencing and format it based on how your model was trained.**

   For the `coco_ssd.mobilenet` example, you can download an image from the [COCO dataset](#) and then reform the image to 300x300:

   ```
   from PIL import Image

   # Download an image for model to make a prediction
   input_image_filename = './input_image.jpg'
   !curl https://farm9.staticflickr.com/8325/8077197378_79efb4805e.jpg --output
   (input_image_filename)

   # Format image so model can make predictions
   resized_image = image.resize((300, 300))

   # Model is quantized, so convert the image to uint8
   x = np.array(resized_image).astype('uint8')
   ```
4. **Use DLR to make inferences.**

Finally, you can use DLR to make a prediction on the image you just downloaded:

```python
out = model.run(x)
```

For more examples using DLR to make inferences from a Neo-compiled model on an edge device, see the [neo-ai-dlr Github repository](#).

## Troubleshoot Errors

This section contains information about how to understand and prevent common errors, the error messages they generate, and guidance on how to resolve these errors. Before moving on, ask yourself the following questions:

- **Did you encounter an error before you deployed your model?** If yes, see [Troubleshoot Neo Compilation Errors](#).
- **Did you encounter an error after you compiled your model?** If yes, see [Troubleshoot Neo Inference Errors](#).

### Error Classification Types

This list classifies the **user errors** you can receive from Neo. These include access and permission errors and load errors for each of the supported frameworks. All other errors are **system errors**.

#### Client permission error

Neo passes the errors for these straight through from the dependent service.

- **Access Denied** when calling `sts:AssumeRole`
- **Any 400 error** when calling Amazon S3 to download or upload a client model
- **PassRole error**

#### Load error

Assuming that the Neo compiler successfully loaded .tar.gz from Amazon S3, check whether the tarball contains the necessary files for compilation. The checking criteria is framework-specific:

- **TensorFlow**: Expects only protobuf file (*.pb or *.pbtxt). For saved models, expects one variables folder.
- **Pytorch**: Expect only one pytorch file (*.pth).
- **MXNET**: Expect only one symbol file (*.json) and one parameter file (*.params).
- **XGBoost**: Expect only one XGBoost model file (*.model). The input model has size limitation.

#### Compilation error

Assuming that the Neo compiler successfully loaded .tar.gz from Amazon S3, and that the tarball contains necessary files for compilation. The checking criteria is:

- **OperatorNotImplemented**: An operator has not been implemented.
- **OperatorAttributeNotImplemented**: The attribute in the specified operator has not been implemented.
- **OperatorAttributeRequired**: An attribute is required for an internal symbol graph, but it is not listed in the user input model graph.
• **OperatorAttributeValueNotValid**: The value of the attribute in the specific operator is not valid.

**Topics**
- Troubleshoot Neo Compilation Errors (p. 1278)
- Troubleshoot Neo Inference Errors (p. 1279)

## Troubleshoot Neo Compilation Errors

This section contains information about how to understand and prevent common compilation errors, the error messages they generate, and guidance on how to resolve these errors.

**Topics**
- How to Use This Page (p. 1278)
- Framework-Related Errors (p. 1278)
- Infrastructure-Related Errors (p. 1279)

### How to Use This Page

Attempt to resolve your error by the going through these sections in the following order:

1. Check that the input of your compilation job satisfies the input requirements. See What input data shapes does SageMaker Neo expect?
2. Check common framework-specific errors.
3. Check if your error is an infrastructure error.

### Framework-Related Errors

**TensorFlow**

<table>
<thead>
<tr>
<th>Error</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>InputConfiguration: Exactly one .pb file is allowed for TensorFlow models.</td>
<td>Make sure you only provide one .pb or .pbtxt file.</td>
</tr>
<tr>
<td>InputConfiguration: Exactly one .pb or .pbtxt file is allowed for TensorFlow models.</td>
<td>Make sure you only provide one .pb or .pbtxt file.</td>
</tr>
<tr>
<td>ClientError: InputConfiguration: TVM cannot convert <code>&lt;model zoo&gt;</code> model. Please make sure the framework you selected is correct. The following operators are not implemented: <code>{&lt;operator name&gt;}</code></td>
<td>Check the operator you chose is supported. See SageMaker Neo Supported Frameworks and Operators.</td>
</tr>
</tbody>
</table>

**Keras**

<table>
<thead>
<tr>
<th>Error</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>InputConfiguration: No h5 file provided in <code>&lt;model path&gt;</code></td>
<td>Check your h5 file is in the Amazon S3 URI you specified. Or</td>
</tr>
</tbody>
</table>
# Amazon SageMaker Developer Guide

## Troubleshoot Errors

<table>
<thead>
<tr>
<th>Error</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>InputConfiguration: Multiple h5 files provided, &lt;model path&gt;, when only one is allowed</td>
<td>Check that the h5 file is correctly formatted.</td>
</tr>
<tr>
<td></td>
<td>Check you are only providing one h5 file.</td>
</tr>
<tr>
<td>ClientError: InputConfiguration: Unable to load provided Keras model. Error: 'sample_weight_mode'</td>
<td>Check the Keras version you specified is supported. See, supported frameworks for cloud instances and edge devices.</td>
</tr>
<tr>
<td></td>
<td>Check that your model input follows NCHW format. See What input data shapes does SageMaker Neo expect?</td>
</tr>
</tbody>
</table>

### MXNet

<table>
<thead>
<tr>
<th>Error</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>ClientError: InputConfiguration: Only one parameter file is allowed for MXNet model. Please make sure the framework you select is correct.</td>
<td>SageMaker Neo will select the first parameter file given for compilation.</td>
</tr>
</tbody>
</table>

### Infrastructure-Related Errors

<table>
<thead>
<tr>
<th>Error</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>ClientError: InputConfiguration: S3 object does not exist. Bucket: &lt;bucket&gt;, Key: &lt;bucket key&gt;</td>
<td>Check the Amazon S3 URI your provided.</td>
</tr>
<tr>
<td>ClientError: InputConfiguration: Bucket &lt;bucket name&gt; is in region &lt;region name&gt; which is different from AWS Sagemaker service region &lt;service region&gt;</td>
<td>Create an Amazon S3 bucket that is in the same region as the service.</td>
</tr>
<tr>
<td>ClientError: InputConfiguration: Unable to untar input model. Please confirm the model is a tar.gz file</td>
<td>Check that your model in Amazon S3 is compressed into a tar.gz file.</td>
</tr>
</tbody>
</table>

## Troubleshoot Neo Inference Errors

This section contains information about how to prevent and resolve some of the common errors you might encounter upon deploying and/or invoking the endpoint. This section applies to PyTorch 1.4.0 or later and MXNet v1.7.0 or later.

- Make sure the first inference (warm-up inference) on a valid input data is done in model_fn(), otherwise the following error message may be seen on the terminal when `predict API` is called:

```
An error occurred (ModelError) when calling the InvokeEndpoint operation: Received server error (0) from <users-sagemaker-endpoint> with message "Your invocation timed out while waiting for a response from container model. Review the latency metrics for each container in Amazon CloudWatch, resolve the issue, and try again."
```
Make sure that the environment variables in the following table are set. If they are not set, the following error message might show up:

### On the terminal:

An error occurred (ModelError) when calling the InvokeEndpoint operation: Received server error (503) from <users-sagemaker-endpoint> with message "{"code": 503, "type": "InternalServerException", "message": "Prediction failed" } ".

### In CloudWatch:

W-9001-model-stdout com.amazonaws.ml.mms.wlm.WorkerLifeCycle - AttributeError: 'NoneType' object has no attribute 'transform'

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAGEMAKER_PROGRAM</td>
<td>inference.py</td>
</tr>
<tr>
<td>SAGEMAKER_SUBMIT_DIRECTORY</td>
<td>/opt/ml/model/code</td>
</tr>
<tr>
<td>SAGEMAKER_CONTAINER_LOG_LEVEL</td>
<td>20</td>
</tr>
<tr>
<td>SAGEMAKER_REGION</td>
<td>&lt;your region&gt;</td>
</tr>
</tbody>
</table>

Make sure that the MMS_DEFAULT_RESPONSE_TIMEOUT environment variable is set to 500 or a higher value while creating the Amazon SageMaker model; otherwise, the following error message may be seen on the terminal:

An error occurred (ModelError) when calling the InvokeEndpoint operation: Received server error (0) from <users-sagemaker-endpoint> with message "Your invocation timed out while waiting for a response from container model. Review the latency metrics for each container in Amazon CloudWatch, resolve the issue, and try again."

### SageMaker Edge Manager

Amazon SageMaker Edge Manager provides model management for edge devices so you can optimize, secure, monitor, and maintain machine learning models on fleets of edge devices such as smart cameras, robots, personal computers, and mobile devices.

### Why Use Edge Manager?

Many machine learning (ML) use cases require running ML models on a fleet of edge devices, which allows you to get predictions in real-time, preserves the privacy of the end users, and lowers the cost of network connectivity. With the increasing availability of low-power edge hardware designed for ML, it is now possible to run multiple complex neural network models on edge devices.

However, operating ML models on edge devices is challenging, because devices, unlike cloud instances, have limited compute, memory, and connectivity. After the model is deployed, you need to continuously monitor the models, because model drift can cause the quality of model to decay overtime. Monitoring models across your device fleets is difficult because you need to write custom code to collect data samples from your device and recognize skew in predictions. In addition, models are often hard-coded into the application. To update the model, you must rebuild and update the entire application or device firmware, which can disrupt your operations.
With SageMaker Edge Manager, you can optimize, run, monitor, and update machine learning models across fleets of devices at the edge.

How Does it Work?

At a high level, there are five main components in the SageMaker Edge Manager workflow: compiling models with SageMaker Neo, packaging Neo-compiled models, deploying models to your devices, running models on the SageMaker inference engine (Edge Manager agent), and maintaining models on the devices.

SageMaker Edge Manager uses SageMaker Neo to optimize your models for the target hardware in one click, then to cryptographically sign your models before deployment. Using SageMaker Edge Manager, you can sample model input and output data from edge devices and send it to the cloud for monitoring and analysis, and view a dashboard that tracks and visually reports on the operation of the deployed models within the SageMaker console.

SageMaker Edge Manager extends capabilities that were previously only available in the cloud to the edge, so developers can continuously improve model quality by using Amazon SageMaker Model Monitor for drift detection, then relabel the data with SageMaker Ground Truth and retrain the models in SageMaker.

How Do I Use SageMaker Edge Manager?

If you are a first time user of SageMaker Edge Manager, we recommend that you test out the Getting Started section.

Getting Started

This guide demonstrates how to complete the necessary steps to register, deploy, and manage a fleet of devices, and how to satisfy Amazon SageMaker Edge Manager prerequisites.

Topics
- Prerequisites (p. 1282)
- Package a Compiled Model (p. 1287)
- Create a Fleet (p. 1288)
- Register Devices (p. 1288)
- Check Device and Fleet (p. 1288)
Prerequisites

1. **Set up an AWS Account.**

Create an AWS account and create an IAM administrator user. For instructions on how to set up your AWS account, see How do I create and activate a new AWS account? For instructions on how to create an administrator user in your AWS account, see Creating your first IAM admin user and group.

2. **Set up an IAM Role and attach policies.**

SageMaker Edge Manager needs access to your Amazon S3 bucket URI. To facilitate this, create an IAM role that can run SageMaker and has permission to access your Amazon S3 URI. You can create an IAM role by using the AWS SDK for Python (Boto3), the SageMaker console, or the AWS CLI. Below is an example of how to create an IAM role using Boto3:

```python
import boto3
import json

AWS_REGION = 'us-west-2'

# Create an IAM client to interact with IAM
iam_client = boto3.client('iam', region_name=AWS_REGION)

role_name = 'edge-manager-demo'

Create a dictionary describing the IAM policy you are attaching to your IAM role. The first statement ID in the statement array, DeviceS3Access, grants devices access to Amazon S3. SageMakerEdgeApis grants access to APIs you need to get a device heartbeat and to register your device. You need to use a role alias in a later step in order to authenticate connected devices to AWS IoT using X.509 certificates. To do this, include the CreateIoTRoleAliasIamPermissions and CreateIoTRoleAlias statements.

```python
policy = {
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "DeviceS3Access",
            "Effect": "Allow",
            "Action": [
                "s3:PutObject"
            ],
            "Resource": [
                "arn:aws:s3:::*SageMaker*",
                "arn:aws:s3:::*Sagemaker*",
                "arn:aws:s3:::*sagemaker*"
            ]
        },
        {
            "Sid": "SageMakerEdgeApis",
            "Effect": "Allow",
            "Action": [
                "sagemaker:SendHeartbeat",
                "sagemaker:GetDeviceRegistration"
            ],
            "Resource": "*"
        },
        {
            "Sid": "CreateIoTRoleAlias",
            "Effect": "Allow",
            "Action": [
                "iot:CreateRoleAlias",
                "iot:DescribeRoleAlias",
                "iot:UpdateRoleAlias"
            ]
        }
    ]
}
```
Create a new IAM role using the policy you defined:

```python
new_role = iam_client.create_role(
    AssumeRolePolicyDocument=json.dumps(policy),
    Path='/',
    RoleName=role_name,
)
```

You need your Amazon Resource Name (ARN) when you create a packaging job in a later step, so store it in a variable as well.

```python
role_arn = new_role['Role']['Arn']
```

After you create a new role, attach the additional permissions it needs to interact with SageMaker and Amazon S3:

```python
iam_client.attach_role_policy(
    RoleName=role_name,
)

iam_client.attach_role_policy(
    RoleName=role_name,
    PolicyArn="arn:aws:iam::aws:policy/AmazonS3FullAccess"
);
```

3. **Create an Amazon S3 bucket.**

Edge Manager accesses your model in Amazon S3 and stores your sample data from your device fleet into Amazon S3. Create an Amazon S3 bucket with the following:

```python
# Create an S3 client so SageMaker can interact with S3
s3_client = boto3.client("s3", region_name=AWS_REGION)

# Name buckets
bucket="demo-bucket"
```
4. **Train a machine learning model.**

See [Train a Model with Amazon SageMaker](#) for more information on how to train a machine learning model using SageMaker. You can optionally upload your locally trained model directly into an Amazon S3 URI bucket.

If you do not have a model yet, use the pre-trained Keras MobileNetV2 model.

```python
import tensorflow as tf
model = tf.keras.applications.MobileNetV2()
model.save("mobilenet_v2.h5")
```

The model comes packaged as a Hierarchical Data Format (HDF). SageMaker expects saved models to be in a compressed tarfile (.tar.gz) file format. Repackage the HDF model into a compressed tarfile with the following:

```python
import tarfile
with tarfile.open("mobilenet_v2.tar.gz", mode="w:gz") as archive:
    archive.add("mobilenet_v2.h5")
```

5. **Upload your trained model to Amazon S3 bucket.**

Once you have a machine learning model, store it in an Amazon S3 bucket.

```python
model_filename= "mobilenet_v2.tar.gz"
# Upload model
s3_client.upload_file(filename=model_filename, bucket=bucket, key=model_filename)
```

6. **Compile your model with SageMaker Neo.**

Compile your machine learning model with SageMaker Neo for an edge device. You need to know your Amazon S3 bucket URI where you stored the trained model, the machine learning framework you used to train your model, the shape of your model's input, and your target device.

For the Keras MobileNet V2 model use the following:

```python
data_shape = '{"input_1":[1,3,224,224]}'
framework = 'keras'
target_device = 'ml_c5'
```

For more information on how to compile your model with SageMaker Neo, see [Compile and Deploy Models with Neo](#).

7. **Create and register AWS IoT thing objects and configure IAM roles.**

SageMaker Edge Manager takes advantage of the AWS IoT Core services to facilitate the connection between the edge devices and endpoints in the AWS cloud. You can take advantage of existing AWS IoT functionality after you set up your devices to work with Edge Manager.
To connect your device to AWS IoT you need to create AWS IoT thing objects, create and register a client certificate with AWS IoT, and create and configure the IAM role for your devices.

- Create AWS IoT thing objects with the following:

```python
# Create an IoT client so you can interact with IoT
iot = boto3.client('iot', region_name=region)

iot.create_thing_type(
    thingTypeName=iot_thing_type
)

# Create two AWS IoT thing objects
iot.create_thing(
    thingName='sample-device-1',
    thingTypeName="edge-manager-demo"
)

iot.create_thing(
    iot_thing_num2_name='sample-device-2',
    thingTypeName="edge-manager-demo"
)
```

- After creating the AWS IoT thing objects, you will need to create X.509 device certificate for your thing objects. This certificate authenticates your device to AWS IoT Core.

Use the following to create a private key, public key, and X.509 certificate file.

```python
# Creates a 2048-bit RSA key pair and issues an X.509
# certificate using the issued public key.
cert = iot.create_keys_and_certificate(setAsActive=True)

# Get certificate from dictionary object and save in its own file
with open('./device.pem.crt', 'w') as f:
    for line in cert['certificatePem'].split('
'):
        f.write(line)

# Get private key from dictionary object and save in its own file
with open('./private.pem.key', 'w') as f:
    for line in cert['keyPair']['PrivateKey'].split('
'):
        f.write(line)

# Get a private key from dictionary object and save in its own file
with open('./public.pem.key', 'w') as f:
    for line in cert['keyPair']['PublicKey'].split('
'):
        f.write(line)
```

- Create an AWS IoT policy document. This policy authorizes your device to interact with AWS IoT services.

To do this, create a role alias so that you can change the role of the device without having to update the device. An AWS IoT role alias provides a mechanism for connected devices to authenticate to AWS IoT using X.509 certificates and then obtain short-lived AWS credentials from an IAM role that is associated with an AWS IoT role alias.

```python
device_role_name="demo-role-name"
```
role_alias_name = device_role_name + "-alias"

# Create role alias with boto3 iot client
role_alias = iot.create_role_alias(
    roleAlias=role_alias_name,
    roleArn=role_arn,
    credentialDurationSeconds=3600
)

# Define policy permissions
alias_policy = {
    "Version": "2012-10-17",
    "Statement": {
        "Effect": "Allow",
        "Action": "iot:AssumeRoleWithCertificate",
        "Resource": role_alias['roleAliasArn']
    }
}

# Create the policy with the permissions defined
aliaspolicy = iot.create_policy(
    policyName='aliaspolicy',
    policyDocument=json.dumps(alias_policy),
)

# Attach the policy to client
iot.attach_policy(
    policyName='aliaspolicy',
    target=create_cert['certificateArn']
)

Get your AWS account-specific endpoint for the credentials provider. Edge devices need the endpoint in order to assume credentials.

# Get the unique endpoint specific to your AWS account that is making the call.
iot_endpoint = iot.describe_endpoint(
    endpointType='iot:CredentialProvider'
)

endpoint = "https://{}/role-aliases/{}/credentials".format(iot_endpoint['endpointAddress'],role_alias_name)

Use the endpoint to make an HTTPS request to the credentials provider to return a security token. The following example command uses curl, but you can use any HTTP client.

curl --cert iot.pem.crt --key iot_key.pem.key --cacert AmazonRootCA1.pem $endpoint

If the certificate is verified, upload the keys and certificate to your Amazon S3 bucket URI.

# Organize S3 bucket, define a folder to store IoT keys and certificate
iot_certificates = folder + "/iot_certificates"
s3_client.upload_file(filename="public.pem.key", bucket=bucket,key=iot_certificates)
s3_client.upload_file(filename="device.pem.key", bucket=bucket,key=iot_certificates)
s3_client.upload_file(filename="AmazonRootCA1.pem", bucket=bucket,key=iot_certificates)
Package a Compiled Model

Packaging jobs take SageMaker Neo–compiled models and make any changes necessary to deploy the model with the inference engine, Edge Manager agent. To package your model, create an edge packaging job with the create_edge_packaging API or the SageMaker console.

You need to provide the name that you used for your Neo compilation job, a name for the packaging job, a role ARN (see Prerequisites (p. 1282) section), a name for the model, a model version, and the Amazon S3 bucket URI for the output of the packaging job. The following is an example using the API.

```python
# Create SageMaker client so you can submit a packaging job
sagemaker_client = boto3.client("sagemaker", region_name=AWS_REGION)

edge_packaging_name='edge-edge-packaging-demo'
compilation_job_name='getting-started-demo'

# Folder within S3 and full S3 bucket URI
folder="package_output"
s3_output_location=f"s3://{bucket}/{folder}"

# Give your model a name and version for reference
model_name="sample-model"
model_version="1.1"

sagemaker_client.create_edge_packaging_job(
    EdgePackagingJobName=edge_packaging_name,
    CompilationJobName=compilation_job_name,
    RoleArn=role_arn,
    ModelName=model_name,
    ModelVersion=model_version,
    OutputConfig={
        "S3OutputLocation": s3_output_location,
    }
)

# Optional - Poll every 30 sec to check completion status
import time

while True:
    response = sagemaker_client.describe_edge_packaging_job(
        EdgePackagingJobName=edge_packaging_name
    )
    if response["EdgePackagingJobStatus"] == 'Completed':
        break
    elif response["EdgePackagingJobStatus"] == 'Failed':
        raise RuntimeError('Packaging job failed')
    print('Packaging model...')
    time.sleep(30)
    print('Done!')
```

Once the packaging job is complete, download the model locally to your device. You can use the Amazon S3 client previously defined to download the model:

```python
s3_client.download_file(bucket=fleet_bucket,
    key=folder,
    filename="mobilenet_v2.tar.gz"
)
```

You can also download your model using the Amazon S3 console at. Search for your bucket by providing a partial name of your bucket in the search field.
Create a Fleet

Device fleets are collections of logically grouped devices that you can use to analyze your data. In this example, you create a fleet consisting of two devices.

To create a fleet, use the `create_device_fleet` API. You need to provide a name for your fleet and an Amazon S3 bucket URI where you want sampled data taken from your devices to be stored.

```python
device_fleet_name="sample-device-fleet"
sagemaker_client.create_device_fleet(
    DeviceFleetName=device_fleet_name,
    OutputConfig={
        'S3OutputLocation': s3_output_location
    }
)
```

Register Devices

**Important**

Device registration is required to use any part of SageMaker Edge Manager.

To interact with the cloud, you need to register your device with SageMaker Edge Manager. In this example, you register two devices to the fleet you created.

```python
sagemaker_client.register_devices(
    DeviceFleetName=device_fleet_name,
    Devices=[
        {
            "DeviceName": "sample-device-1",
            "IotThingName": "sample-thing-name-1"
        },
        {
            "DeviceName": "sample-device-2",
            "IotThingName": "sample-thing-name-2"
        }
    ]
)
```

Check Device and Fleet

Check your device or fleet is connected and sampling data. Making periodic checks, manually or automatically, allows you to check your device or fleet is working properly.

- **Check a single device in your fleet.**

  Check that your device is working. You need to provide the name of the fleet to which the device belongs and the unique device identifier.

  ```python
  sagemaker_client.describe_device(
      DeviceName="sample-device-1",
      DeviceFleetName=device_fleet_name
  )
  
  For the given model, you can see the name, model version, latest sample time, and when the last inference was made.
  ```

  ```json
  { "DeviceName": "sample-device",
    "DeviceFleetName": "sample-device-fleet",
  }"
The timestamp provided by `LatestHeartbeat` indicates the last signal that was received from the device. `LatestSampleTime` and `LatestInference` describe the time stamp of the last data sample and inference, respectively.

- **Check your fleet.**

Check that your fleet is working with `GetDeviceFleetReport`. Provide the name of the fleet the device belongs to.

```python
sagemaker_client.get_device_fleet_report(
    DeviceFleetName=device_fleet_name
)
```

For a given model, you can see the name, model version, latest sample time, and when the last inference was made, along with the Amazon S3 bucket URI where the data samples are stored.

```
{
    "DeviceFleetName": "sample-device-fleet",
    "OutputConfig": {
        "S3OutputLocation": "s3://fleet-bucket/package_output",
    },
    "AgentVersions": [{"Version": "1.1", "AgentCount": 2}]
    "DeviceStats": {"Connected": 2, "Registered": 2},
    "Models": [{
        "ModelName": "sample-model",
        "ModelVersion": "1.1",
        "OfflineDeviceCount": 0,
        "ConnectedDeviceCount": 2,
        "ActiveDeviceCount": 2,
        "SamplingDeviceCount": 100
    }]
}
```

## Devices and Fleets

Fleets are collections of logically grouped devices you can use to collect and analyze data. You can use SageMaker Edge Manager to operate machine learning models on a fleet of smart cameras, smart speakers, robots, and other edge devices.

Create a fleet and register your devices either programatically with the AWS SDK for Python (Boto3) or through the SageMaker console.

**Topics**

- Inference Engine (Edge Manager Agent) (p. 1290)
Inference Engine (Edge Manager Agent)

The Edge Manager agent is an inference engine for your edge devices. Use the agent to make predictions with models loaded onto your edge devices. The agent also collects model metrics and captures data at specific intervals. Sample data is stored in your Amazon S3 bucket.

The agent is released in binary format for all the supported operating systems. Check that your operating system is supported and meets the minimum OS requirement in the following table:

Linux

**Version:** Ubuntu 18.04

**Supported Binary Formats:** x86-64 bit (ELF binary) and ARMv8 64 bit (ELF binary)

Windows

**Version:** Windows 10 version 1909

**Supported Binary Formats:** x86-32 bit (DLL) and x86-64 bit (DLL)

Installing Edge Manager agent

To use Edge Manager agent, you first need to obtain the release artifacts. The release artifacts are stored in an Amazon S3 bucket in the us-west-2 Region. To download the artifacts, specify your operating system and the `VERSION`. The `VERSION` is broken into three components: `<MAJOR_VERSION>.<YYYY-MM-DD>-<SHA-7>`, where:

- `<MAJOR_VERSION>`: The release version. The release version is currently set to 1.
- `<YYYY-MM-DD>`: The time stamp of the artifacts release.
- `<SHA-7>`: The repository commit ID from which the release is built.

You need to provide the `MAJOR_VERSION` and the time stamp in `<YYYY-MM-DD>` format. We suggest you use the latest artifact release time stamp. Use the following to get the latest time stamp.

```
aws s3 ls s3://sagemaker-edge-release-store-us-west-2-windows-x86/Releases/ | sort -r
```

Based on your operating system, use the following commands to install the artifacts.

Windows 32-bit

```
aws s3 cp s3://sagemaker-edge-release-store-us-west-2-windows-x86/Releases/<VERSION>/sha256_hex.shasum .
```

Windows 64-bit

```
aws s3 cp s3://sagemaker-edge-release-store-us-west-2-windows-x64/Releases/<VERSION>/.<VERSION>.zip .
aws s3 cp s3://sagemaker-edge-release-store-us-west-2-windows-x64/Releases/<VERSION>/sha256_hex.shasum .
```
Linux x86-64

aws s3 cp s3://sagemaker-edge-release-store-us-west-2-linux-x64/Releases/<VERSION>/<VERSION>.tgz
aws s3 cp s3://sagemaker-edge-release-store-us-west-2-linux-x64/Releases/<VERSION>/sha256_hex.shasum

Linux ARMv8

aws s3 cp s3://sagemaker-edge-release-store-us-west-2-linux-armv8/Releases/<VERSION>/<VERSION>.tgz
aws s3 cp s3://sagemaker-edge-release-store-us-west-2-linux-armv8/Releases/<VERSION>/sha256_hex.shasum

You will also need to download a Root Certificate. This certificate validates model artifacts signed by AWS before loading them onto your edge devices.

Replace `{bucket_name}` corresponding to your platform from the list of supported operation systems.

# Windows and Linux
aws s3 cp s3://{bucket_name}/Certificates/<VERSION>/<VERSION>.pem

Running SageMaker Edge Manager agent

SageMaker Edge Manager agent can be run as a standalone process in the form of an Executable and Linkable Format (ELF) executable binary or can be linked against as a Dynamic Shared Object (.dll). Running as a standalone executable binary is the preferred mode and is supported on Linux. Running as a shared object (.dll) is supported on Windows.

On Linux, we recommend that you run the binary via a service that’s a part of your initialization (init) system. If you want to run the binary directly, you can do so in a terminal as shown below. If you have a modern OS, there are no other installations necessary prior to running the agent, since all the requirements are statically built into the executable. This gives you flexibility to run the agent on the terminal, as a service, or within a container.

./sagemaker_edge_agent -a <ADDRESS_TO_SOCKET> -c <PATH_TO_CONFIG_FILE>

Obtain AWS IoT Credentials Certificates

SageMaker Edge Manager takes advantage of the AWS IoT Core services to facilitate the connection between the edge devices and endpoints in the AWS cloud. By doing so, you can take advantage of existing AWS IoT functionality after you set up your devices to work with Edge Manager.

To connect your device to AWS IoT, you need to create AWS IoT thing objects, create and register a client certificate with AWS IoT, and create and configure an IAM role for your devices.

Follow the instructions from the How to Eliminate the Need for Hardcoded AWS Credentials in Devices by Using the AWS IoT Credentials Provider blog post for detailed instructions on how to create and register a client certificate with AWS IoT, and create and configure an IAM role for your devices. For general information on authorizing AWS services, see Authorizing direct calls to AWS services.

Models on Edge Devices

The Edge Manager agent can load one model at a time and make inference with that model on edge devices. The agent validates the model signature and loads into memory all the artifacts produced by
the edge packaging job. This step requires all the required certificates described in previous steps to be installed along with rest of the binary installation. If the model's signature cannot be validated, then loading of the model fails with appropriate return code and reason.

See Installing Edge Manager agent (p. 1290) for instructions on how to download the root certificate and binary release artifacts. For information about creating and registering a client certificate with AWS IoT and creating and configuring an IAM role for your devices see Security in AWS IoT.

Create a Fleet

You can create a fleet programmatically with the AWS SDK for Python (Boto3) or through the Amazon SageMaker console.

Create a Fleet (Boto3)

Use the CreateDeviceFleet API to create a fleet. Specify a name for the fleet, as well as an Amazon S3 URI where you want the device to store sampled data. You can optionally include an Amazon Resource Role (ARN) role to use an alias for AWS IoT, a description of the fleet, tags, and an AWS KMS Key ID.

```python
import boto3
def create_device_fleet

sagemaker_client.create_device_fleet(    DeviceFleetName="sample-fleet-name",    IAMRole="arn:aws:iam::999999999:role/rolename",    Description="fleet description",    OutputConfig={        S3OutputLocation="s3://bucket/",        KMSKeyId: "1234abcd-12ab-34cd-56ef-1234567890ab",    },    Tags=[        {            "Key": "string",            "Value" : "string"        }    ],)

Use DescribeDeviceFleet API to get a description of fleets you created.

```python
def describe_device_fleet

By default, it returns the name of the fleet, the device fleet ARN, the Amazon S3 bucket URI, the IAM role, the role alias created in AWS IoT, a timestamp of when the fleet was created, and a timestamp of when the fleet was last modified.

A description of the fleet is returned if one was provided.

```json
{    "DeviceFleetName": "sample-fleet-name",    "DeviceFleetArn": "arn:aws:sagemaker:us-west-2:9999999999:device-fleet/sample-fleet-name",    "IAMRole": "arn:aws:iam::9999999999:role/rolename",    "Description": "this is a sample fleet",}
Create a Fleet (Console)

You can create a Edge Manager packaging job using the Amazon SageMaker console.

1. In the SageMaker console, choose Edge Inference and then choose Edge device fleets.
2. Choose Create device fleet.
3. Enter a name for the device fleet in the Device fleet name field. Choose Next.
4. On the **Output configuration** page, specify the Amazon S3 bucket URI where you want to store sample data from your device fleet. You can optionally add an encryption key as well by selecting an existing AWS KMS key from the dropdown list or by entering a key's ARN. Choose **Submit**.
5. Choose the name of your device fleet to be redirected to the device fleet details. This page displays the name of the device fleet, ARN, description (if you provided one), date the fleet was created, last time the fleet was modified, Amazon S3 bucket URI, AWS KMS key ID (if provided), AWS IoT alias (if provided), and IAM role. If you added tags, they appear in the **Device fleet tags** section.

**Register a Device**

**Important**

Device registration is required to use any part of SageMaker Edge Manager.

You can create a fleet programmatically with the AWS SDK for Python (Boto3) or through the Amazon SageMaker console.

**Register a Device (Boto3)**

To register your device, first create and register an AWS IoT thing object and configure an IAM role. SageMaker Edge Manager takes advantage of the AWS IoT Core services to facilitate the connection between the edge devices and the cloud. You can take advantage of existing AWS IoT functionality after you set up your devices to work with Edge Manager.

To connect your device to AWS IoT you need to create AWS IoT thing objects, create and register a client certificate with AWS IoT, and create and configure IAM role for your devices.

See the **Getting Started Guide** for an in-depth example or the **Explore AWS IoT Core services in hands-on tutorial**.

Use the `RegisterDevices` API to register your device. Provide the name of the fleet of which you want the devices to be a part, as well as a name for the device. You can optionally add a description to the device, tags, and AWS IoT thing name associated with the device.

```python
sagemaker_client.register_devices(  
    DeviceFleetName="sample-fleet-name",  
    Devices=[
```
Register a Device (Console)

You can register your device using the Amazon SageMaker console.

1. In the SageMaker console, choose **Edge Inference** and then choose choose **Edge devices**.
2. Choose **Register devices**.

3. In the **Device properties** section, enter the name of the fleet the device belongs to under the **Device fleet name** field. Choose **Next**.

4. In the **Device source** section, add your devices one by one. You must include a **Device Name** for each device in your fleet. You can optionally provide a description (in the **Description** field) and an Internet of Things (IoT) object name (in the **IoT name** field). Choose **Submit** once you have added all your devices.
The Devices page displays the name of the device you have added, the fleet to which it belongs, when it was registered, the last heartbeat, and the description and AWS IoT name, if you provided one.

Choose a device to view the device’s details, including the device name, fleet, ARN, description, IoT Thing name, when the device was registered, and the last heartbeat.

Check Status

Check that your device or fleet is connected and sampling data. Making periodic checks, manually or automatically, allows you to check that your device or fleet is working properly.

Use the Amazon S3 console to interactively choose a fleet for a status check. You can also use the AWS SDK for Python (Boto3). The following describes different APIs from Boto3 you can use to check the status of your device or fleet. Use the API that best fits your use case.

- Check an individual device.

To check the status of an individual device, use DescribeDevice API. A list containing one or more models is provided if a models have been deployed to the device.

```python
sagemaker_client.describe_device(  
    DeviceName="sample-device-1",  
    DeviceFleetName="sample-fleet-name"
)
```

Running DescribeDevice returns:

```json
{
  "DeviceName": "sample-device",
  "Description": "this is a sample device",
  "DeviceFleetName": "sample-device-fleet",
  "IoTThingName": "SampleThing",
  "RegistrationTime": 1600977370,
  "LatestHeartbeat": 1600977370,
  "Models": [
```
• Check a fleet of devices.

To check the status of the fleet, use the GetDeviceFleetReport API. Provide the name of the device fleet to get a summary of the fleet.

```python
sagemaker_client.get_device_fleet_report(
    DeviceFleetName="sample-fleet-name"
)
```

• Check for a heartbeat.

Each device within a fleet periodically generates a signal, or "heartbeat". The heartbeat can be used to check that the device is communicating with Edge Manager. If the timestamp of the last heartbeat is not being updated, the device may be failing.

Check the last heartbeat with made by a device with the DescribeDevice API. Specify the name of the device and the fleet to which the edge device belongs.

```python
sagemaker_client.describe_device(
    DeviceName="sample-device-1",
    DeviceFleetName="sample-fleet-name"
)
```

### Package Model

SageMaker Edge Manager packaging jobs take Amazon SageMaker Neo–compiled models and make any changes necessary to deploy the model with the inference engine, Edge Manager agent.

**Topics**

- Prerequisites (p. 1298)
- Package a Model (Amazon SageMaker Console) (p. 1300)
- Package a Model (Boto3) (p. 1303)

**Prerequisites**

To package a model, you must do the following:

1. **Compile your machine learning model with SageMaker Neo.**

   If you have not already done so, compile your model with SageMaker Neo. For more information on how to compile your model, see Compile and Deploy Models with Neo. If you are first-time user of SageMaker Neo, go through Getting Started with Neo Edge Devices.

2. **Get the name of your compilation job.**

   Provide the name of the compilation job name you used when you compiled your model with SageMaker Neo. Open the Amazon SageMaker console and choose Compilation jobs to find a list of...
compilations that have been submitted to your AWS account. The names of submitted compilation jobs are in the **Name** column.

3. **Get your IAM ARN.**

You need an Amazon Resource Name (ARN) of an IAM role that you can use to download and upload the model and contact SageMaker Neo.

Use one of the following methods to get your IAM ARN:

- **Programmatically with the SageMaker Python SDK**

  ```python
  import sagemaker
  # Initialize SageMaker Session object so you can interact with AWS resources
  sess = sagemaker.Session()
  # Get the role ARN
  role = sagemaker.get_execution_role()
  print(role)
  >> arn:aws:iam::<your-aws-account-id>:role/<your-role-name>
  ```

  For more information about using the SageMaker Python SDK, see the [SageMaker Python SDK API](https://docs.aws.amazon.com/sagemaker/latest/dg/)

- **Using the AWS Identity and Access Management (IAM) console**

  Sign in to the AWS Management Console and open the IAM console. In the IAM **Resources** section, choose **Roles** to view a list of roles in your AWS account. Select or create a role that has `AmazonSageMakerFullAccess`, `AWSIoTFullAccess`, and `AmazonS3FullAccess`.

  For more information on IAM, see [What is IAM?](https://docs.aws.amazon.com/)

4. **Have an S3 bucket URI.**

You need to have at least one Amazon Simple Storage Service (Amazon S3) bucket URI to store your Neo-compiled model, the output of the Edge Manager packaging job, and sample data from your device fleet.

Use one of the following methods to create an Amazon S3 bucket:

- **Programmatically with the SageMaker Python SDK**

  You can use the default Amazon S3 bucket during a session. A default bucket is created based on the following format: `sagemaker-{region}-{aws-account-id}`. To create a default bucket with the SageMaker Python SDK, use the following:

  ```python
  import sagemaker
  session=sagemaker.create_session()
  bucket=session.default_bucket()
  ```

- **Using the Amazon S3 console**

  Open the [Amazon S3 console](https://aws.amazon.com/) and see [How do I create an S3 Bucket?](https://docs.aws.amazon.com/for step-by-step instructions.)
Package a Model (Amazon SageMaker Console)

You can create a SageMaker Edge Manager packaging job using the Amazon SageMaker console. Before continuing, make sure you have satisfied the Prerequisites.

1. In the SageMaker console, choose Edge Inference and then choose Edge packaging jobs, as shown in the following image.

2. On the Job properties page, enter a name for your packaging job under Edge packaging job name. Name your model and give it a version: enter this under Model name and Model version, respectively.

3. Next, select an IAM role. You can chose a role or let AWS create a role for you. You can optionally specify a resource key ARN and job tags.

4. Choose Next.
5. Specify the name of the compilation job you used when compiling your model with SageMaker Neo in the **Compilation job name** field. Choose **Next**.
6. On the **Output configuration** page, enter the Amazon S3 bucket URI in which you want to store the output of the packaging job.

### Output configuration
Use the fields below to specify the S3 bucket URI where you want devices to store sample data. You can also (optionally) specify an encryption key.

**S3 bucket URI**
Enter your S3 bucket URI where you want devices to store sample data.

`s3://bucket-example/sagemaker-edge/device-output`

To find a path, [go to Amazon S3](https://aws.amazon.com/s3/)

**Encryption key - optional**
Encrypt your data. Choose an existing KMS key or enter a key's ARN.

**No Custom Encryption**

The **Status** column on the **Edge packaging** jobs page should read **IN PROGRESS**. Once the packaging job is complete, the status updates to **COMPLETED**.

Selecting a packaging job directs you to that job's settings. The **Job settings** section displays the job name, ARN, status, creation time, last modified time, duration of the packaging job, and role ARN.

The **Input configuration** section displays the location of the model artifacts, the data input configuration, and the machine learning framework of the model.
The **Output configuration** section displays the output location of the packaging job, the target device for which the model was compiled, and any tags you created.

7. Choose the name of your device fleet to be redirected to the device fleet details. This page displays the name of the device fleet, ARN, description (if you provided one), date the fleet was created, last time the fleet was modified, Amazon S3 bucket URI, AWS KMS key ID (if provided), AWS IoT alias (if provided), and IAM role. If you added tags, they appear in the **Device fleet tags** section.

## Package a Model (Boto3)

You can create a SageMaker Edge Manager packaging job with the AWS SDK for Python (Boto3). Before continuing, make sure you have satisfied the **Prerequisites**.

To request an edge packaging job, use `CreateEdgePackagingJob`. You need to provide a name to your edge packaging job, the name of your SageMaker Neo compilation job, your role Amazon resource name (ARN), a name for your model, a version for your model, and the Amazon S3 bucket URI where you want to store the output of your packaging job.

```python
# Import AWS SDK for Python (Boto3)
import boto3

# Create Edge client so you can submit a packaging job
sagemaker_client = boto3.client("sagemaker", region_name='aws-region')

sagemaker_client.create_edge_packaging_job(
    EdgePackagingJobName="edge-packaging-name",
    CompilationJobName="neo-compilation-name",
    RoleArn="arn:aws:iam::99999999999:role/rolename",
    ModelName="sample-model-name",
    ModelVersion="model-version",
    OutputConfig={
        "S3OutputLocation": "s3://your-bucket/",
    }
)
```

You can check the status of an edge packaging job using `DescribeEdgePackagingJob`:

```python
response = sagemaker_client.describe_edge_packaging_job(
    EdgePackagingJobName="edge-packaging-name")
```

This returns a dictionary that can be used to poll the status of the packaging job:

```python
# Optional - Poll every 30 sec to check completion status
import time

while True:
    response = sagemaker_client.describe_edge_packaging_job(
        EdgePackagingJobName="edge-packaging-name")

    if response['EdgePackagingJobStatus'] == 'Completed':
        break
    elif response['EdgePackagingJobStatus'] == 'Failed':
        raise RuntimeError('Packaging job failed')
    print('Packaging model...')
    time.sleep(30)
print('Done!')
```

For a list of packaging jobs, use `ListEdgePackagingJobs`. You can use this API to search for a specific packaging job. Provide a partial name to filter packaging job names for `NameContains`, a partial name
for ModelNameContains to filter for jobs in which the model name contains the name you provide. Also specify with which column to sort with SortBy, and by which direction to sort for SortOrder (either Ascending or Descending).

```python
sagemaker_client.list_edge_packaging_jobs(
    "NameContains": "sample",
    "ModelNameContains": "sample",
    "SortBy": "column-name",
    "SortOrder": "Descending"
)
```

To stop a packaging job, use StopEdgePackagingJob and provide the name of your edge packaging job.

```python
sagemaker_client.stop_edge_packaging_job(
    EdgePackagingJobName="edge-packaging-name"
)
```

For a full list of Edge Manager APIs, see the Boto3 documentation.

**Manage Model**

SageMaker Edge Manager agent provides a list of Model Management APIs that implement control plane and data plane APIs on edge devices. Along with this documentation, we recommend going through the sample client implementation which shows canonical usage of the below described APIs.

The proto file is available as a part of the release artifacts (inside the release tarball). In this doc, we list and describe the usage of APIs listed in this proto file.

**Note**

There is one-to-one mapping for these APIs on Windows release and a sample code for an application implement in C# is shared with the release artifacts for Windows. Below instructions are for running the Agent as a standalone process, applicable for to the release artifacts for Linux.

Extract the archive based on your OS. Where VERSION is broken into three components: <MAJOR_VERSION>.<YYYY-MM-DD>-<SHA-7>. See Installing Edge Manager agent (p. 1290) for information on how to obtain the release version (<MAJOR_VERSION>), time stamp of the release artifact (<YYYY-MM-DD>), and the repository commit ID (SHA-7)

Linux

The zip archive can be extracted with the command:

```
tar -xvzf <VERSION>.tgz
```

Windows

The zip archive can be extracted with the UI or command:

```
unzip <VERSION>.tgz
```

The release artifact hierarchy (after extracting the tar/zip archive) is shown below. The agent proto file is available under api/.

```
0.20201205.7ee4b0b
```
Load Model

The Edge Manager agent software on edge device supports loading one model at a time, and calling inference on that model. This API validates the model signature and loads into memory all the artifacts produced by the EdgePackagingJob operation. This step requires all the required certificates to be installed along with rest of the agent binary installation. If the model's signature cannot be validated then this step fails with appropriate return code and error messages in the log.

```cpp
// perform load for a model
// Note:
// 1. currently only local filesystem paths are supported for loading models.
// 2. currently only one model could be loaded at any time, loading of multiple models simultaneously shall be implemented in the future.
// 3. users are required to unload any loaded model to load another model.
// Status Codes:
// 1. OK - load is successful
// 2. UNKNOWN - unknown error has occurred
// 3. INTERNAL - an internal error has occurred
// 4. NOT_FOUND - model doesn't exist at the url
// 5. ALREADY_EXISTS - model with the same name is already loaded
// 6. RESOURCE_EXHAUSTED - memory is not available to load the model
// 7. FAILED_PRECONDITION - model is not compiled for the machine.
// rpc LoadModel(LoadModelRequest) returns (LoadModelResponse);
```

Input

```cpp```
```
```
Manage Model

// Message LoadModelRequest
string url = 1; // Model name needs to match regex "^[a-zA-Z0-9](-*[a-zA-Z0-9])*"  
string name = 2; 

Output

// // Response for LoadModel rpc call // 
message LoadModelResponse { 
  Model model = 1; 
}

// Model represents the metadata of a model // url - url representing the path of the model // name - name of model // input_tensor_metadatas - TensorMetadata array for the input tensors // output_tensor_metadatas - TensorMetadata array for the output tensors // // Note: // 1. input and output tensor metadata could empty for dynamic models. // 
message Model { 
  string url = 1; 
  string name = 2; 
  repeated TensorMetadata input_tensor_metadatas = 3; 
  repeated TensorMetadata output_tensor_metadatas = 4; 
}

Unload Model

Unloads a previously loaded model. It is identified via the model alias which was provided during loadModel. If the alias is not found or model is not loaded then returns error.

// Perform unload for a model // Status Codes: // 1. OK - unload is successful // 2. UNKNOWN - unknown error has occurred // 3. INTERNAL - an internal error has occurred // 4. NOT_FOUND - model doesn't exist // 
rpc UnLoadModel(UnLoadModelRequest) returns (UnLoadModelResponse); 

Input

// // Request for UnLoadModel rpc call // 
// message UnLoadModelRequest { 
  string name = 1; // Model name needs to match regex "^[a-zA-Z0-9](-*[a-zA-Z0-9])*" 
}

Output

//
// response for UnLoadModel rpc call
//
message UnLoadModelResponse {}

List Models

Lists all the loaded models and their aliases.

// lists the loaded models
// Status Codes:
// 1. OK - unload is successful
// 2. UNKNOWN - unknown error has occurred
// 3. INTERNAL - an internal error has occurred
//
rpc ListModels(ListModelsRequest) returns (ListModelsResponse);

Input

//
// request for ListModels rpc call
//
message ListModelsRequest {}

Output

//
// response for ListModels rpc call
//
message ListModelsResponse {
  repeated Model models = 1;
}

Describe Model

Describes a model that is loaded on the agent.

// Status Codes:
// 1. OK - load is successful
// 2. UNKNOWN - unknown error has occurred
// 3. INTERNAL - an internal error has occurred
// 4. NOT_FOUND - model doesn’t exist at the url
//
rpc DescribeModel(DescribeModelRequest) returns (DescribeModelResponse);

Input

//
// request for DescribeModel rpc call
//
message DescribeModelRequest {
  string name = 1;
}
Capture Data

Allows the client application to capture input and output tensors in Amazon S3 bucket, and optionally the auxiliary. The client application is expected to pass a unique capture ID along with each call to this API. This can be later used to query status of the capture.

```protobuf
// allows users to capture input and output tensors along with auxiliary data.
// Status Codes:
// 1. OK - data capture successfully initiated
// 2. UNKNOWN - unknown error has occurred
// 3. INTERNAL - an internal error has occurred
// 5. ALREADY_EXISTS - capture initiated for the given capture_id
// 6. RESOURCE_EXHAUSTED - buffer is full cannot accept any more requests.
// 7. OUT_OF_RANGE - timestamp is in the future.
// 8. INVALID_ARGUMENT - capture_id is not of expected format.
// rpc CaptureData(CaptureDataRequest) returns (CaptureDataResponse);
```

Input

```protobuf
enum Encoding {
  CSV = 0;
  JSON = 1;
  NONE = 2;
  BASE64 = 3;
}

// AuxiliaryData represents a payload of extra data to be capture along with inputs and outputs of inference
// encoding - supports the encoding of the data
// data - represents the data of shared memory, this could be passed in two ways:
// a. send across the raw bytes of the multi-dimensional tensor array
// b. send a SharedMemoryHandle which contains the posix shared memory segment id and offset in bytes to location of multi-dimensional tensor array.
// message AuxiliaryData {
  string name = 1;
  Encoding encoding = 2;
  oneof data {
    bytes byte_data = 3;
    SharedMemoryHandle shared_memory_handle = 4;
  }
}

// Tensor represents a tensor, encoded as contiguous multi-dimensional array.
// tensor_metadata - represents metadata of the shared memory segment
// data_or_handle - represents the data of shared memory, this could be passed in two ways:
```
// a. send across the raw bytes of the multi-dimensional tensor array
// b. send a SharedMemoryHandle which contains the posix shared memory segment
// id and offset in bytes to location of multi-dimensional tensor array.

message Tensor {
  TensorMetadata tensor_metadata = 1; //optional in the predict request
  oneof data {
    bytes byte_data = 4;
    // will only be used for input tensors
    SharedMemoryHandle shared_memory_handle = 5;
  }
}

// request for CaptureData rpc call

message CaptureDataRequest {
  string model_name = 1;
  string capture_id = 2; //uuid string
  Timestamp inference_timestamp = 3;
  repeated Tensor input_tensors = 4;
  repeated Tensor output_tensors = 5;
  repeated AuxiliaryData inputs = 6;
  repeated AuxiliaryData outputs = 7;
}

Output

// response for CaptureData rpc call

message CaptureDataResponse {}

Get Capture Status

Depending on the models loaded the input and output tensors can be large (for many edge devices). Capture to the cloud can be time consuming. So the CaptureData() is implemented as an asynchronous operation. A capture ID is a unique identifier that the client provides during capture data call, this ID can be used to query the status of the asynchronous call.

// allows users to query status of capture data operation
// Status Codes:
// 1. OK - data capture successfully initiated
// 2. UNKNOWN - unknown error has occurred
// 3. INTERNAL - an internal error has occurred
// 4. NOT_FOUND - given capture id doesn't exist.
// rpc GetCaptureDataStatus(GetCaptureDataStatusRequest) returns
// (GetCaptureDataStatusResponse);

Input

// request for GetCaptureDataStatus rpc call

message GetCaptureDataStatusRequest {
  string capture_id = 1;
}
Output

```cpp
enum CaptureDataStatus {
    FAILURE = 0;
    SUCCESS = 1;
    IN_PROGRESS = 2;
    NOT_FOUND = 3;
}

// response for GetCaptureDataStatus rpc call
message GetCaptureDataStatusResponse {
    CaptureDataStatus status = 1;
}
```

Predict

The `predict` API performs inference on a previously loaded model. It accepts a request in the form of a tensor that is directly fed into the neural network. The output is the output tensor (or scalar) from the model. This is a blocking call.

```cpp
// perform inference on a model.

// Note:
// 1. users can chose to send the tensor data in the protobuf message or
// through a shared memory segment on a per tensor basis, the Predict
// method with handle the decode transparently.
// 2. serializing large tensors into the protobuf message can be quite expensive,
// based on our measurements it is recommended to use shared memory of
// tenors larger than 256KB.
// 3. SMEdge IPC server will not use shared memory for returning output tensors,
// i.e., the output tensor data will always send in byte form encoded
// in the tensors of PredictResponse.
// 4. currently SMEdge IPC server cannot handle concurrent predict calls, all
// these call will be serialized under the hood. this shall be addressed
// in a later release.
// Status Codes:
// 1. OK - prediction is successful
// 2. UNKNOWN - unknown error has occurred
// 3. INTERNAL - an internal error has occurred
// 4. NOT_FOUND - when model not found
// 5. INVALID_ARGUMENT - when tenors types mismatch
//
// rpc Predict(PredictRequest) returns (PredictResponse);
```

Input

```cpp
// request for Predict rpc call
message PredictRequest {
    string name = 1;
    repeated Tensor tensors = 2;
}

// Tensor represents a tensor, encoded as contiguous multi-dimensional array.
// tensor_metadata - represents metadata of the shared memory segment
// data_or_handle - represents the data of shared memory, this could be passed in
two ways:
```
// a. send across the raw bytes of the multi-dimensional tensor
// memory segment
// tensor array.

message Tensor {
  TensorMetadata tensor_metadata = 1;  // optional in the predict request
  oneof data {
    bytes byte_data = 4;
    // will only be used for input tensors
    SharedMemoryHandle shared_memory_handle = 5;
  }
}

// Tensor represents a tensor, encoded as contiguous multi-dimensional array.
// tensor_metadata - represents metadata of the shared memory segment
// data_or_handle - represents the data of shared memory, this could be passed in
// two ways:
// a. send across the raw bytes of the multi-dimensional tensor
// memory segment
// tensor array.

message Tensor {
  TensorMetadata tensor_metadata = 1;  // optional in the predict request
  oneof data {
    bytes byte_data = 4;
    // will only be used for input tensors
    SharedMemoryHandle shared_memory_handle = 5;
  }
}

// TensorMetadata represents the metadata for a tensor
// name - name of the tensor
// data_type - data type of the tensor
// shape - array of dimensions of the tensor

message TensorMetadata {
  string name = 1;
  DataType data_type = 2;
  repeated int32 shape = 3;
}

// SharedMemoryHandle represents a posix shared memory segment
// offset - offset in bytes from the start of the shared memory segment.
// segment_id - shared memory segment id corresponding to the posix shared memory
// segment.
// size - size in bytes of shared memory segment to use from the offset position.

message SharedMemoryHandle {
  uint64 size = 1;
  uint64 offset = 2;
  uint64 segment_id = 3;
}
Use Amazon SageMaker Elastic Inference (EI)

This feature is not available in the China Regions.

By using Amazon Elastic Inference (EI), you can speed up the throughput and decrease the latency of getting real-time inferences from your deep learning models that are deployed as Amazon SageMaker hosted models, but at a fraction of the cost of using a GPU instance for your endpoint. EI allows you to add inference acceleration to a hosted endpoint for a fraction of the cost of using a full GPU instance. Add an EI accelerator in one of the available sizes to a deployable model in addition to a CPU instance type, and then add that model as a production variant to an endpoint configuration that you use to deploy a hosted endpoint. You can also add an EI accelerator to a SageMaker notebook instance so that you can test and evaluate inference performance when you are building your models.

Elastic Inference is supported in EI-enabled versions of TensorFlow, Apache MXNet, and PyTorch. To use any other deep learning framework, export your model by using ONNX, and then import your model into MXNet. You can then use your model with EI as an MXNet model. For information about importing an ONNX model into MXNet, see Importing an ONNX model into MXNet.

Topics

- How EI Works (p. 1312)
- Choose an EI Accelerator Type (p. 1313)
- Use EI in a SageMaker Notebook Instance (p. 1313)
- Use EI on a Hosted Endpoint (p. 1314)
- Frameworks that Support EI (p. 1314)
- Use EI with SageMaker Built-in Algorithms (p. 1314)
- EI Sample Notebooks (p. 1314)
- Set Up to Use EI (p. 1314)
- Attach EI to a Notebook Instance (p. 1318)
- Use EI on Amazon SageMaker Hosted Endpoints (p. 1320)

How EI Works

Amazon Elastic Inference accelerators are network attached devices that work along with SageMaker instances in your endpoint to accelerate your inference calls. Elastic Inference accelerates inference by allowing you to attach fractional GPUs to any SageMaker instance. You can select the client instance to run your application and attach an Elastic Inference accelerator to use the right amount of GPU acceleration for your inference needs. Elastic Inference helps you lower your cost when not fully utilizing your GPU instance for inference. We recommend trying Elastic Inference with your model using different CPU instances and accelerator sizes.
The following EI accelerator types are available. You can configure your endpoints or notebook instances with any EI accelerator type.

In the table, the throughput in teraflops (TFLOPS) is listed for both single-precision floating-point (F32) and half-precision floating-point (F16) operations. The memory in GB is also listed.

<table>
<thead>
<tr>
<th>Accelerator Type</th>
<th>F32 Throughput in TFLOPS</th>
<th>F16 Throughput in TFLOPS</th>
<th>Memory in GB</th>
</tr>
</thead>
<tbody>
<tr>
<td>ml.eia2.medium</td>
<td>1</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>ml.eia2.large</td>
<td>2</td>
<td>16</td>
<td>4</td>
</tr>
<tr>
<td>ml.eia2.xlarge</td>
<td>4</td>
<td>32</td>
<td>8</td>
</tr>
<tr>
<td>ml.eia1.medium</td>
<td>1</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>ml.eia1.large</td>
<td>2</td>
<td>16</td>
<td>2</td>
</tr>
<tr>
<td>ml.eia1.xlarge</td>
<td>4</td>
<td>32</td>
<td>4</td>
</tr>
</tbody>
</table>

Choose an EI Accelerator Type

Consider the following factors when choosing an accelerator type for a hosted model:

- Models, input tensors and batch sizes influence the amount of accelerator memory you need. Start with an accelerator type that provides at least as much memory as the file size of your trained model. Factor in that a model might use significantly more memory than the file size at runtime.
- Demands on CPU compute resources, main system memory, and GPU-based acceleration and accelerator memory vary significantly between different kinds of deep learning models. The latency and throughput requirements of the application also determine the amount of compute and acceleration you need. Thoroughly test different configurations of instance types and EI accelerator sizes to make sure you choose the configuration that best fits the performance needs of your application.

For more information on selecting an EI accelerator, see:

- Amazon Elastic Inference Overview
- Choosing an Instance and Accelerator Type for Your Model
- Optimizing costs in Amazon Elastic Inference with TensorFlow

Use EI in a SageMaker Notebook Instance

Typically, you build and test machine learning models in a SageMaker notebook before you deploy them for production. You can attach EI to your notebook instance when you create the notebook instance. You can set up an endpoint that is hosted locally on the notebook instance by using the local mode supported by TensorFlow, MXNet, and PyTorch estimators and models in the Amazon SageMaker Python SDK to test inference performance. Elastic Inference enabled PyTorch is not currently supported on notebook instances. For instructions on how to attach EI to a notebook instance and set up a local endpoint for inference, see Attach EI to a Notebook Instance (p. 1318). There are also Elastic Inference-enabled SageMaker Notebook Jupyter kernels for Elastic Inference-enabled versions of TensorFlow and Apache MXNet. For information about using SageMaker notebook instances, see Use Amazon SageMaker Notebook Instances
Use EI on a Hosted Endpoint

When you are ready to deploy your model for production to provide inferences, you create a SageMaker hosted endpoint. You can attach EI to the instance where your endpoint is hosted to increase its performance at providing inferences. For instructions on how to attach EI to a hosted endpoint instance, see Use EI on Amazon SageMaker Hosted Endpoints (p. 1320).

Frameworks that Support EI

Amazon Elastic Inference is designed to be used with AWS enhanced versions of TensorFlow, Apache MXNet, or PyTorch machine learning frameworks. These enhanced versions of the frameworks are automatically built into containers when you use the Amazon SageMaker Python SDK, or you can download them as binary files and import them in your own Docker containers.

You can download the EI-enabled TensorFlow binary files from the public amazonei-tensorflow Amazon S3 bucket to the TensorFlow serving containers. For more information about building a container that uses the EI-enabled version of TensorFlow, see Amazon Elastic Inference with TensorFlow in SageMaker.

You can download the EI-enabled MXNet binary files from the public amazonei-apachemxnet Amazon S3 bucket to the MXNet serving containers. For more information about building a container that uses the EI-enabled version of MXNet, see Amazon Elastic Inference with MXNet in SageMaker.

You can download the EI-enabled PyTorch binary files from the public amazonei-pytorch Amazon S3 bucket to the PyTorch serving containers. For more information about building a container that uses the EI-enabled version of PyTorch, see Amazon Elastic Inference with PyTorch in SageMaker.

To use Elastic Inference in a hosted endpoint, you can choose any of the following frameworks depending on your needs.

- SageMaker Python SDK - Deploy TensorFlow models
- SageMaker Python SDK - Deploy MXNet models
- SageMaker Python SDK - Deploy PyTorch models

If you need to create a custom container for deploying your model that is complex and requires extensions to a framework that the SageMaker pre-built containers do not support, use the low-level AWS SDK for Python (Boto 3).

Use EI with SageMaker Built-in Algorithms

Currently, the Image Classification Algorithm (p. 690) and Object Detection Algorithm (p. 770) built-in algorithms support EI. For an example that uses the Image Classification algorithm with EI, see End-to-End Multiclass Image Classification Example.

EI Sample Notebooks

The following Sample notebooks provide examples of using EI in SageMaker:

- Using Amazon Elastic Inference with MXNet on Amazon SageMaker
- Using Amazon Elastic Inference with MXNet on an Amazon SageMaker Notebook Instance
- Using Amazon Elastic Inference with a pre-trained TensorFlow Serving model on SageMaker

Set Up to Use EI

Use the instructions in this topic only if one of the following applies to you:
• You want to use a customized role or permission policy.
• You want to use a VPC for your hosted model or notebook instance.

Note
If you already have an execution role that has the AmazonSageMakerFullAccess managed policy attached (this is true for any IAM role that you create when you create a notebook instance, training job, or model in the console) and you are not connecting to an EI model or notebook instance in a VPC, you do not need to make any of these changes to use EI in Amazon SageMaker.

Topics
• Set Up Required Permissions (p. 1315)
• Use a Custom VPC to Connect to EI (p. 1317)

Set Up Required Permissions

To use EI in SageMaker, the role that you use to open a notebook instance or create a deployable model must have a policy with the required permissions attached. You can attach the AmazonSageMakerFullAccess managed policy, which contains the required permissions, to the role, or you can add a custom policy that has the required permissions. For information about creating an IAM role, see Creating a Role for an AWS Service (Console) in the AWS Identity and Access Management User Guide. For information about attaching a policy to a role, see Adding and Removing IAM Policies.

Add these permissions specifically for connecting EI in an IAM policy.

```
{
   "Effect": "Allow",
   "Action": [
      "elastic-inference:Connect",
      "ec2:DescribeVpcEndpoints"
   ],
   "Resource": "*"
}
```

The following IAM policy is the complete list of required permissions to use EI in SageMaker.

```
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Effect": "Allow",
         "Action": [
            "elastic-inference:Connect",
            "ec2:DescribeVpcEndpoints"
         ],
         "Resource": "*"
      },
      {
         "Effect": "Allow",
         "Action": [
            "sagemaker:*"
         ],
         "Resource": "*"
      },
      {
         "Effect": "Allow",
         "Action": [
```

1315
"Action": [
    "ecr:GetAuthorizationToken",
    "ecr:GetDownloadUrlForLayer",
    "ecr:BatchGetImage",
    "ecr:BatchCheckLayerAvailability",
    "cloudwatch:PutMetricData",
    "cloudwatch:PutMetricAlarm",
    "cloudwatch:DescribeAlarms",
    "cloudwatch:DeleteAlarms",
    "ec2:CreateNetworkInterface",
    "ec2:CreateNetworkInterfacePermission",
    "ec2:DeleteNetworkInterface",
    "ec2:DeleteNetworkInterfacePermission",
    "ec2:DescribeNetworkInterfaces",
    "ec2:DescribeVpcs",
    "ec2:DescribeDhcpOptions",
    "ec2:DescribeSubnets",
    "ec2:DescribeSecurityGroups",
    "application-autoscaling:DeleteScalingPolicy",
    "application-autoscaling:DeleteScheduledAction",
    "application-autoscaling:DeregisterScalableTarget",
    "application-autoscaling:DescribeScalingTargets",
    "application-autoscaling:DescribeScalingActivities",
    "application-autoscaling:DescribeScalingPolicies",
    "application-autoscaling:DescribeScheduledActions",
    "application-autoscaling:PutScalingPolicy",
    "application-autoscaling:PutScheduledAction",
    "application-autoscaling:RegisterScalableTarget",
    "logs:CreateLogGroup",
    "logs:CreateLogStream",
    "logs:DescribeLogStreams",
    "logs:GetLogEvents",
    "logs:PutLogEvents"
],
"Resource": "**",

"Effect": "Allow",
"Action": [
    "s3:GetObject",
    "s3:PutObject",
    "s3:DeleteObject"
],
]
],
"Effect": "Allow",
"Action": ["s3:CreateBucket", "s3:GetBucketLocation", "s3:ListBucket", "s3:ListAllMyBuckets"]
],
"Resource": "**"
],
"Effect": "Allow",
"Action": ["s3:GetObject"]
],
"Resource": "**",
"Condition": {"StringLike": {"s3:bucket": "**"}}
"StringEqualsIgnoreCase": {
    "s3:ExistingObjectTag/SageMaker": "true"
  }
},
{
  "Action": "iam:CreateServiceLinkedRole",
  "Effect": "Allow",
  "Resource": "arn:aws:iam::*:role/aws-service-role/sagemaker.application-autoscaling.amazonaws.com/AWSServiceRoleForApplicationAutoScaling_SageMakerEndpoint",
  "Condition": {
    "StringLike": {
      "iam:AWSServiceName": "sagemaker.application-autoscaling.amazonaws.com"
    }
  }
},
{
  "Effect": "Allow",
  "Action": [
    "iam:PassRole"
  ],
  "Resource": "*",
  "Condition": {
    "StringEquals": {
      "iam:PassedToService": "sagemaker.amazonaws.com"
    }
  }
}
}

Use a Custom VPC to Connect to EI

To use EI with SageMaker in a VPC, you need to create and configure two security groups, and set up a PrivateLink VPC interface endpoint. EI uses VPC interface endpoint to communicate with SageMaker endpoints in your VPC. The security groups you create are used to connect to the VPC interface endpoint.

Set up Security Groups to Connect to EI

To use EI within a VPC, you need to create two security groups:

- A security group to control access to the VPC interface endpoint that you will set up for EI.
- A security group that allows SageMaker to call into the first security group.

To configure the two security groups

1. Create a security group with no outbound connections. You will attach this to the VPC endpoint interface you create in the next section.
2. Create a second security group with no inbound connections, but with an outbound connection to the first security group.
3. Edit the first security group to allow inbound connections only to the second security group an all outbound connections.

For more information about VPC security groups, see Security Groups for Your VPC in the Amazon Virtual Private Cloud User Guide.
Set up a VPC Interface Endpoint to Connect to EI

To use EI with SageMaker in a custom VPC, you need to set up a VPC interface endpoint (PrivateLink) for the EI service.

- Set up a VPC interface endpoint (PrivateLink) for the EI. Follow the instructions at Creating an Interface Endpoint. In the list of services, choose `com.amazonaws.<region>.elastic-inference.runtime`. For Security group, make sure you select the first security group you created in the previous section to the endpoint.
- When you set up the interface endpoint, choose all of the Availability Zones where EI is available. EI fails if you do not set up at least two Availability Zones. For information about VPC subnets, see VPCs and Subnets.

Attach EI to a Notebook Instance

To test and evaluate inference performance using EI, you can attach EI to a notebook instance when you create or update a notebook instance. You can then use EI in local mode to host a model at an endpoint hosted on the notebook instance. You should test various sizes of notebook instances and EI accelerators to evaluate the configuration that works best for your use case.

Set Up to Use EI

To use EI locally in a notebook instance, create a notebook instance with an EI instance.

To create a notebook instance with an EI instance

1. Open the Amazon SageMaker console at https://console.aws.amazon.com/sagemaker/.
2. In the navigation pane, choose Notebook instances.
3. Choose Create notebook instance.
4. For Notebook instance name, provide a unique name for your notebook instance.
5. For notebook instance type, choose a CPU instance such as ml.t2.medium.
6. For Elastic Inference (EI), choose an instance from the list, such as ml.eia2.medium.
7. For IAM role, choose an IAM role that has the required permissions to use SageMaker and EI.
8. (Optional) For VPC - Optional, if you want the notebook instance to use a VPC, choose one from the available list. Otherwise, leave it as No VPC. If you use a VPC follow the instructions at Use a Custom VPC to Connect to EI (p. 1317).
9. (Optional) For Lifecycle configuration - optional, either leave it as No configuration or choose a lifecycle configuration. For more information, see Customize a Notebook Instance Using a Lifecycle Configuration Script (p. 127).
10. (Optional) For Encryption key - optional, Optional) If you want SageMaker to use an AWS Key Management Service (AWS KMS) key to encrypt data in the ML storage volume attached to the notebook instance, specify the key.
11. (Optional) For Volume Size In GB - optional, leave the default value of 5.
12. (Optional) For Tags, add tags to the notebook instance. A tag is a label you assign to help manage your notebook instances. A tag consists of a key and a value, both of which you define.
13. Choose Create Notebook Instance.

After you create your notebook instance with EI attached, you can create a Jupyter notebook and set up an EI endpoint that is hosted locally on the notebook instance.

Topics
- Use EI in Local Mode in SageMaker (p. 1319)
Use EI in Local Mode in SageMaker

To use EI locally in an endpoint hosted on a notebook instance, use local mode with the Amazon SageMaker Python SDK versions of either the TensorFlow, MXNet, or PyTorch estimators or models. For more information about local mode support in the SageMaker Python SDK, see https://github.com/aws/sagemaker-python-sdk#sagemaker-python-sdk-overview.

Topics
- Use EI in Local Mode with SageMaker TensorFlow Estimators and Models (p. 1319)
- Use EI in Local Mode with SageMaker Apache MXNet Estimators and Models (p. 1319)
- Use EI in Local Mode with SageMaker PyTorch Estimators and Models (p. 1319)

Use EI in Local Mode with SageMaker TensorFlow Estimators and Models

To use EI with TensorFlow in local mode, specify local for `instance_type` and `local_sagemaker_notebook` for `accelerator_type` when you call the `deploy` method of an estimator or a model object. For more information about Amazon SageMaker Python SDK TensorFlow estimators and models, see https://sagemaker.readthedocs.io/en/stable/frameworks/tensorflow/index.html.

The following code shows how to use local mode with an estimator object. To call the `deploy` method, you must have previously either:

- Trained the model by calling the `fit` method of an estimator.
- Pass a model artifact when you initialize the model object.

```python
# Deploys the model to a local endpoint
tf_predictor = tf_model.deploy(initial_instance_count=1,
                                 instance_type='local',
                                 accelerator_type='local_sagemaker_notebook')
```

Use EI in Local Mode with SageMaker Apache MXNet Estimators and Models

To use EI with MXNet in local mode, specify local for `instance_type` and `local_sagemaker_notebook` for `accelerator_type` when you call the `deploy` method of an estimator or a model object. For more information about Amazon SageMaker Python SDK MXNet estimators and models, see https://sagemaker.readthedocs.io/en/stable/frameworks/mxnet/index.html.

The following code shows how to use local mode with an estimator object. You must have previously called the `fit` method of the estimator to train the model.

```python
# Deploys the model to a local endpoint
mxnet_predictor = mxnet_estimator.deploy(initial_instance_count=1,
                                          instance_type='local',
                                          accelerator_type='local_sagemaker_notebook')
```

For a complete example of using EI in local mode with MXNet, see the sample notebook at https://github.com/awslabs/amazon-sagemaker-examples/blob/master/sagemaker-python-sdk/mxnet_mnist/mxnet_mnist_elastic_inference_local.ipynb.

Use EI in Local Mode with SageMaker PyTorch Estimators and Models

To use EI with PyTorch in local mode, when you call the `deploy` method of an estimator or a model object, specify local for `instance_type` and `local_sagemaker_notebook` for
accelerator_type. For more information about Amazon SageMaker Python SDK PyTorch estimators and models, see SageMaker PyTorch Estimators and Models.

The following code shows how to use local mode with an estimator object. You must have previously called the `fit` method of the estimator to train the model.

```python
# Deploys the model to a local endpoint
pytorch_predictor = pytorch_estimator.deploy(initial_instance_count=1,
                                           instance_type='local',
                                           accelerator_type='local_sagemaker_notebook')
```

Use EI on Amazon SageMaker Hosted Endpoints

To use Elastic Inference (EI) in Amazon SageMaker with a hosted endpoint for real-time inference, specify an EI accelerator when you create the deployable model to be hosted at that endpoint. You can do this in one of the following ways:

- Use the Amazon SageMaker Python SDK versions of either the TensorFlow, MXNet, or PyTorch and the SageMaker pre-built containers for TensorFlow, MXNet, and PyTorch
- Build your own container, and use the low-level SageMaker API (Boto 3). You will need to import the EI-enabled version of either TensorFlow, MXNet, or PyTorch from the provided Amazon S3 locations into your container, and use one of those versions to write your training script.
- Use either the Image Classification Algorithm (p. 690) or Object Detection Algorithm (p. 770) build-in algorithms, and use the AWS SDK for Python (Boto3) to run your training job and create your deployable model and hosted endpoint.

Topics

- Use EI with a SageMaker TensorFlow Container (p. 1320)
- Use EI with a SageMaker MXNet Container (p. 1321)
- Use EI with a SageMaker PyTorch Container (p. 1321)
- Use EI with Your Own Container (p. 1322)

Use EI with a SageMaker TensorFlow Container


SageMaker provides default model training and inference code for your convenience. For custom file formats, you might need to implement custom model training and inference code.

Use an Estimator Object

To use an estimator object with EI, when you use the `deploy` method, include the `accelerator_type` input argument. The estimator returns a predictor object, which we call its `deploy` method, as shown in the example code.

```python
# Deploy an estimator using EI (using the accelerator_type input argument)
predictor = estimator.deploy(initial_instance_count=1,
                              instance_type='ml.m4.xlarge',
                              accelerator_type='ml.eia2.medium')
```
Use a Model Object

To use a model object with EI, when you use the deploy method, include the `accelerator_type` input argument. The estimator returns a predictor object, which we call its deploy method, as shown in the example code.

```python
# Deploy a model using EI (using the accelerator_type input argument)
predictor = model.deploy(initial_instance_count=1,
                        instance_type='ml.m4.xlarge',
                        accelerator_type='ml.eia2.medium')
```

Use EI with a SageMaker MXNet Container

To use MXNet with EI in SageMaker, you need to call the `deploy` method of either the `Estimator` or `Model` objects. You then specify an accelerator type using the `accelerator_type` input argument. For information about using MXNet in the Amazon SageMaker Python SDK, see https://sagemaker.readthedocs.io/en/stable/frameworks/mxnet/index.html

For your convenience, SageMaker provides default model training and inference code. For custom file formats, you might need to write custom model training and inference code.

Use an Estimator Object

To use an estimator object with EI, when you use the deploy method, include the `accelerator_type` input argument. The estimator returns a predictor object, which we call its deploy method, as shown in the example code.

```python
# Deploy an estimator using EI (using the accelerator_type input argument)
predictor = estimator.deploy(initial_instance_count=1,
                        instance_type='ml.m4.xlarge',
                        accelerator_type='ml.eia2.medium')
```

Use a Model Object

To use a model object with EI, when you use the deploy method, include the `accelerator_type` input argument. The estimator returns a predictor object, which we call its deploy method, as shown in the example code.

```python
# Deploy a model using EI (using the accelerator_type input argument)
predictor = model.deploy(initial_instance_count=1,
                        instance_type='ml.m4.xlarge',
                        accelerator_type='ml.eia2.medium')
```

For a complete example of using EI with MXNet in SageMaker, see the sample notebook at https://github.com/awslabs/amazon-sagemaker-examples/blob/master/sagemaker-python-sdk/mxnet_mnist/mxnet_mnist_elastic_inference.ipynb

Use EI with a SageMaker PyTorch Container

To use PyTorch with EI in SageMaker, you need to call the `deploy` method of either the `Estimator` or `Model` objects. You then specify an accelerator type using the `accelerator_type` input argument. For information about using PyTorch in the Amazon SageMaker Python SDK, see SageMaker PyTorch Estimators and Models.

For your convenience, SageMaker provides default model training and inference code. For custom file formats, you might need to write custom model training and inference code.
Use an Estimator Object

To use an estimator object with EI, when you use the deploy method, include the `accelerator_type` input argument. The estimator returns a predictor object, which we call its deploy method, as shown in this example code.

```python
# Deploy an estimator using EI (using the accelerator_type input argument)
predictor = estimator.deploy(initial_instance_count=1,
                             instance_type='ml.m4.xlarge',
                             accelerator_type='ml.eia2.medium')
```

Use a Model Object

To use a model object with EI, when you use the deploy method, include the `accelerator_type` input argument. The model returns a predictor object, which we call its deploy method, as shown in this example code.

```python
# Deploy a model using EI (using the accelerator_type input argument)
predictor = model.deploy(initial_instance_count=1,
                          instance_type='ml.m4.xlarge',
                          accelerator_type='ml.eia2.medium')
```

Use EI with Your Own Container

To use EI with a model in a custom container that you build, use the low-level AWS SDK for Python (Boto 3). Download and import the AWS EI-enabled versions of TensorFlow, Apache MXNet, or PyTorch machine learning frameworks, and write your training script using those frameworks.

Import the EI Version of TensorFlow, MXNet, or PyTorch into Your Docker Container


Create an EI Endpoint with AWS SDK for Python (Boto 3)

To create an endpoint by using AWS SDK for Python (Boto 3), you first create an endpoint configuration. The endpoint configuration specifies one or more models (called production variants) that you want to host at the endpoint. To attach EI to one or more of the production variants hosted at the endpoint, you specify one of the EI instance types as the `AcceleratorType` field for that `ProductionVariant`. You then pass that endpoint configuration when you create the endpoint.

Create an Endpoint Configuration

To use EI, you need to specify an accelerator type in the endpoint configuration.
# Create Endpoint Configuration
from time import gmtime, strftime

endpoint_config_name = 'ImageClassificationEndpointConfig-' + strftime("%Y-%m-%d-%H-%M-%S", gmtime())
print(endpoint_config_name)
create_endpoint_config_response = sagemaker.create_endpoint_config(
    EndpointConfigName = endpoint_config_name,
    ProductionVariants=[
        {'InstanceType':'ml.m4.xlarge',
        'InitialInstanceCount':1,
        'ModelName':model_name,
        'VariantName':'AllTraffic',
        'AcceleratorType':'ml.eia2.medium'}])
print("Endpoint Config Arn: " + create_endpoint_config_response['EndpointConfigArn'])

Create an Endpoint

After you create an endpoint configuration with an accelerator type, you can create an endpoint.

endpoint_name = 'ImageClassificationEndpoint-' + strftime("%Y-%m-%d-%H-%M-%S", gmtime())
endpoint_response = sagemaker.create_endpoint(
    EndpointName=endpoint_name,
    EndpointConfigName=endpoint_config_name)

After creating the endpoint, you can invoke it using the `invoke_endpoint` method in a Boto3 runtime object, as you would any other endpoint.

Use Batch Transform

Use batch transform when you need to do the following:

- Preprocess datasets to remove noise or bias that interferes with training or inference from your dataset.
- Get inferences from large datasets.
- Run inference when you don’t need a persistent endpoint.
- Associate input records with inferences to assist the interpretation of results.

To filter input data before performing inferences or to associate input records with inferences about those records, see Associate Prediction Results with Input Records (p. 1326). For example, you can filter input data to provide context for creating and interpreting reports about the output data.

For more information about batch transforms, see Get Inferences for an Entire Dataset with Batch Transform (p. 14).

Topics

- Use Batch Transform to Get Inferences from Large Datasets (p. 1324)
- Speed up a Batch Transform Job (p. 1325)
- Use Batch Transform to Test Production Variants (p. 1325)
- Batch Transform Errors (p. 1325)
- Batch Transform Sample Notebooks (p. 1325)
- Associate Prediction Results with Input Records (p. 1326)
**Use Batch Transform to Get Inferences from Large Datasets**

Batch transform automatically manages the processing of large datasets within the limits of specified parameters. For example, suppose that you have a dataset file, `input1.csv`, stored in an S3 bucket. The content of the input file might look like the following:

```
Record1-Attribute1, Record1-Attribute2, Record1-Attribute3, ..., Record1-AttributeM
Record2-Attribute1, Record2-Attribute2, Record2-Attribute3, ..., Record2-AttributeM
Record3-Attribute1, Record3-Attribute2, Record3-Attribute3, ..., Record3-AttributeM
...
RecordN-Attribute1, RecordN-Attribute2, RecordN-Attribute3, ..., RecordN-AttributeM
```

When a batch transform job starts, SageMaker initializes compute instances and distributes the inference or preprocessing workload between them. Batch Transform partitions the Amazon S3 objects in the input by key and maps Amazon S3 objects to instances. When you have multiple files, one instance might process `input1.csv`, and another instance might process the file named `input2.csv`.

To keep large payloads below the `MaxPayloadInMB` limit, you can split an input file into several mini-batches. For example, you might create a mini-batch from `input1.csv` by including only two of the records:

```
Record3-Attribute1, Record3-Attribute2, Record3-Attribute3, ..., Record3-AttributeM
Record4-Attribute1, Record4-Attribute2, Record4-Attribute3, ..., Record4-AttributeM
```

**Note**

SageMaker processes each input file separately. It doesn’t combine mini-batches from different input files to comply with the `MaxPayloadInMB` limit.

To split input files into mini-batches, when you create a batch transform job, set the `SplitType` parameter value to `Line`. If `SplitType` is set to `None` or if an input file can’t be split into mini-batches, SageMaker uses the entire input file in a single request.

If the batch transform job successfully processes all of the records in an input file, it creates an output file with the same name and the `.out` file extension. For multiple input files, such as `input1.csv` and `input2.csv`, the output files are named `input1.csv.out` and `input2.csv.out`. The batch transform job stores the output files in the specified location in Amazon S3, such as `s3://awsexamplebucket/output/`.

The predictions in an output file are listed in the same order as the corresponding records in the input file. The output file `input1.csv.out`, based on the input file shown earlier, would look like the following:

```
Inference1-Attribute1, Inference1-Attribute2, Inference1-Attribute3, ..., Inference1-AttributeM
Inference2-Attribute1, Inference2-Attribute2, Inference2-Attribute3, ..., Inference2-AttributeM
Inference3-Attribute1, Inference3-Attribute2, Inference3-Attribute3, ..., Inference3-AttributeM
...
InferenceN-Attribute1, InferenceN-Attribute2, InferenceN-Attribute3, ..., InferenceN-AttributeM
```

To combine the results of multiple output files into a single output file, set the `AssembleWith` parameter to `Line`. 

1324
When the input data is very large and is transmitted using HTTP chunked encoding, to stream the data to the algorithm, set `MaxPayloadInMB` to 0. Amazon SageMaker built-in algorithms don't support this feature.

For information about using the API to create a batch transform job, see the `CreateTransformJob` API. For more information about the correlation between batch transform input and output objects, see `OutputDataConfig`. For an example of how to use batch transform, see Step 6.2: Deploy the Model with Batch Transform (p. 70).

**Speed up a Batch Transform Job**

If you are using the `CreateTransformJob` API, you can reduce the time it takes to complete batch transform jobs by using optimal values for parameters such as `MaxPayloadInMB`, `MaxConcurrentTransforms`, or `BatchStrategy`. If you are using the SageMaker console, you can specify these optimal parameter values in the Additional configuration section of the Batch transform job configuration page. SageMaker automatically finds the optimal parameter settings for built-in algorithms. For custom algorithms, provide these values through an execution-parameters endpoint.

**Use Batch Transform to Test Production Variants**

To test different models or various hyperparameter settings, create a separate transform job for each new model variant and use a validation dataset. For each transform job, specify a unique model name and location in Amazon S3 for the output file. To analyze the results, use Inference Pipeline Logs and Metrics (p. 1227).

**Batch Transform Errors**

SageMaker uses the Amazon S3 Multipart Upload API to upload results from a batch transform job to Amazon S3. If an error occurs, the uploaded results are removed from Amazon S3. In some cases, such as when a network outage occurs, an incomplete multipart upload might remain in Amazon S3. To avoid incurring storage charges, we recommend that you add the S3 bucket policy to the S3 bucket lifecycle rules. This policy deletes incomplete multipart uploads that might be stored in the S3 bucket. For more information, see Object Lifecycle Management.

If a batch transform job fails to process an input file because of a problem with the dataset, SageMaker marks the job as failed. If an input file contains a bad record, the transform job doesn't create an output file for that input file because doing so prevents it from maintaining the same order in the transformed data as in the input file. When your dataset has multiple input files, a transform job continues to process input files even if it fails to process one. The processed files still generate useable results.

Exceeding the `MaxPayloadInMB` limit causes an error. This might happen with a large dataset if it can't be split, the `SplitType` parameter is set to `none`, or individual records within the dataset exceed the limit.

If you are using your own algorithms, you can use placeholder text, such as `ERROR`, when the algorithm finds a bad record in an input file. For example, if the last record in a dataset is bad, the algorithm places the placeholder text for that record in the output file.

**Batch Transform Sample Notebooks**

For a sample notebook that uses batch transform with a principal component analysis (PCA) model to reduce data in a user-item review matrix, followed by the application of a density-based spatial clustering of applications with noise (DBSCAN) algorithm to cluster movies, see Batch Transform with PCA and DBSCAN Movie Clusters. For instructions on creating and accessing Jupyter notebook...
instances that you can use to run the example in SageMaker, see Use Amazon SageMaker Notebook Instances (p. 124). After creating and opening a notebook instance, choose the SageMaker Examples tab to see a list of all the SageMaker examples. The topic modeling example notebooks that use the NTM algorithms are located in the Advanced functionality section. To open a notebook, choose its Use tab, then choose Create copy.

**Associate Prediction Results with Input Records**

When making predictions on a large dataset, you can exclude attributes that aren’t needed for prediction. After the predictions have been made, you can associate some of the excluded attributes with those predictions or with other input data in your report. By using batch transform to perform these data processing steps, you can often eliminate additional preprocessing or postprocessing. You can use input files in JSON and CSV format only.

**Topics**
- Workflow for Associating Inferences with Input Records (p. 1326)
- Use Data Processing in Batch Transform Jobs (p. 1327)
- Supported JSONPath Operators (p. 1327)
- Batch Transform Examples (p. 1328)

**Workflow for Associating Inferences with Input Records**

The following diagram shows the workflow for associating inferences with input records.

To associate inferences with input data, there are three main steps:

1. Filter the input data that is not needed for inference before passing the input data to the batch transform job. Use the InputFilter parameter to determine which attributes to use as input for the model.

2. Associate the input data with the inference results. Use the JoinSource parameter to combine the input data with the inference.
3. Filter the joined data to retain the inputs that are needed to provide context for interpreting the predictions in the reports. Use `OutputFilter` to store the specified portion of the joined dataset in the output file.

### Use Data Processing in Batch Transform Jobs

When creating a batch transform job with `CreateTransformJob` to process data:

1. Specify the portion of the input to pass to the model with the `InputFilter` parameter in the `DataProcessing` data structure.
2. Join the raw input data with the transformed data with the `JoinSource` parameter.
3. Specify which portion of the joined input and transformed data from the batch transform job to include in the output file with the `OutputFilter` parameter.
4. Choose either JSON- or CSV-formatted files for input:
   - For JSON- or JSON Lines-formatted input files, SageMaker either adds the `SageMakerOutput` attribute to the input file or creates a new JSON output file with the `SageMakerInput` and `SageMakerOutput` attributes. For more information, see `DataProcessing`.
   - For CSV-formatted input files, the joined input data is followed by the transformed data and the output is a CSV file.

If you use an algorithm with the `DataProcessing` structure, it must support your chosen format for both input and output files. For example, with the `TransformOutput` field of the `CreateTransformJob` API, you must set both the `Content Type` and `Accept` parameters to one of the following values: `text/csv`, `application/json`, or `application/jsonlines`. The syntax for specifying columns in a CSV file and specifying attributes in a JSON file are different. Using the wrong syntax causes an error. For more information, see Batch Transform Examples (p. 1328). For more information about input and output file formats for built-in algorithms, see Use Amazon SageMaker Built-in Algorithms (p. 635).

The record delimiters for the input and output must also be consistent with your chosen file input. The `SplitType` parameter indicates how to split the records in the input dataset. The `AssembleWith` parameter indicates how to reassemble the records for the output. If you set input and output formats to `text/csv`, you must also set the `SplitType` and `AssemblyType` parameters to `line`. If you set the input and output formats to `application/jsonlines`, you can set both `SplitType` and `AssemblyType` to `line`.

For JSON files, the attribute name `SageMakerOutput` is reserved for output. The JSON input file can't have an attribute with this name. If it does, the data in the input file might be overwritten.

### Supported JSONPath Operators

To filter and join the input data and inference, use a JSONPath subexpression. SageMaker supports only a subset of the defined JSONPath operators. The following table lists the supported JSONPath operators. For CSV data, each row is taken as a JSON array, so only index based JSONPaths can be applied, e.g. `@[0], `[1:]`. CSV data should also follow RFC format.

<table>
<thead>
<tr>
<th>JSONPath Operator</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>The root element to a query. This operator is required at the beginning of all path expressions.</td>
<td>$</td>
</tr>
<tr>
<td><code>&lt;name&gt;</code></td>
<td>A dot-notated child element.</td>
<td>$.id</td>
</tr>
<tr>
<td>JSONPath Operator</td>
<td>Description</td>
<td>Example</td>
</tr>
<tr>
<td>-------------------</td>
<td>-------------</td>
<td>---------</td>
</tr>
<tr>
<td>*</td>
<td>A wildcard. Use in place of an attribute name or numeric value.</td>
<td>$.id.*</td>
</tr>
<tr>
<td>['&lt;name&gt;' , '&lt;name&gt;']</td>
<td>A bracket-notated element or multiple child elements.</td>
<td>$['id','SageMakerOutput']</td>
</tr>
<tr>
<td>&lt;number&gt; ( , &lt;number&gt;)</td>
<td>An index or array of indexes. Negative index values are also supported. A -1 index refers to the last element in an array.</td>
<td>$[1], $[1,3,5]</td>
</tr>
<tr>
<td>&lt;start&gt;:&lt;end&gt;</td>
<td>An array slice operator. The array slice() method extracts a section of an array and returns a new array. If you omit &lt;start&gt;, SageMaker uses the first element of the array. If you omit &lt;end&gt;, SageMaker uses the last element of the array.</td>
<td>$[2:5], $[:5], $[2:]</td>
</tr>
</tbody>
</table>

When using the bracket-notation to specify multiple child elements of a given field, additional nesting of children within brackets is not supported. For example, $.field1.'child1','child2' is supported while $.field1.'child1','child2.grandchild' is not.

For more information about JSONPath operators, see JsonPath on GitHub.

**Batch Transform Examples**

The following examples show some common ways to join input data with prediction results.

**Topics**
- Example: Output Only Inferences (p. 1328)
- Example: Output Input Data and Inferences (p. 1329)
- Example: Output an ID Column with Results and Exclude the ID Column from the Input (CSV) (p. 1329)
- Example: Output an ID Attribute with Results and Exclude the ID Attribute from the Input (JSON) (p. 1330)

**Example: Output Only Inferences**

By default, the DataProcessing parameter doesn't join inference results with input. It outputs only the inference results.

If you want to explicitly specify to not join results with input, use the Amazon SageMaker Python SDK and specify the following settings in a transformer call.

```python
sm_transformer = sagemaker.transformer.Transformer...
sm_transformer.transform(..., input_filter="$", join_source="None", output_filter="$")
```

To output inferences using the AWS SDK for Python, add the following code to your CreateTransformJob request. The following code mimics the default behavior.

```python
{
    "DataProcessing": {
        "InputFilter": ",",
        "JoinSource": "None",
```
Example: Output Input Data and Inferences

If you're using the Amazon SageMaker Python SDK, to combine the input data with the inferences in the output file, specify "Input" for the JoinSource parameter in a transformer call.

```python
sm_transformer = sagemaker.transformer.Transformer(…)
sm_transformer.transform(…, join_source= "Input")
```

If you're using the AWS SDK for Python (Boto 3), join all input data with the inference by adding the following code to your CreateTransformJob request.

```json
{
    "DataProcessing": {
        "JoinSource": "Input"
    }
}
```

For JSON or JSON Lines input files, the results are in the SageMakerOutput key in the input JSON file. For example, if the input is a JSON file that contains the key-value pair ("key":1), the data transform result might be ("label":1).

SageMaker stores both in the input file in the SageMakerInput key.

```json
{
    "key":1,
    "SageMakerOutput":{"label":1}
}
```

**Note**
The joined result for JSON must be a key-value pair object. If the input isn't a key-value pair object, SageMaker creates a new JSON file. In the new JSON file, the input data is stored in the SageMakerInput key and the results are stored as the SageMakerOutput value.

For a CSV file, for example, if the record is [1,2,3] and the label result is [1], then the output file would contain [1,2,3,1].

Example: Output an ID Column with Results and Exclude the ID Column from the Input (CSV)

If you are using the Amazon SageMaker Python SDK, to include results or an ID column in the output, specify indexes of the joined dataset in a transformer call. For example, if your data includes five columns and the first one is the ID column, use the following transformer request.

```python
sm_transformer = sagemaker.transformer.Transformer(…)
sm_transformer.transform(…, input_filter="$[1:]", join_source= "Input", output_filter="$")
```

If you are using the AWS SDK for Python (Boto 3), add the following code to your CreateTransformJob request.

```json
{
    "DataProcessing": {
        "InputFilter": "$[1:]",
        "Input": "$[1:]",
        "OutputFilter": "$"}
}
```
To specify columns in SageMaker, use the index of the array elements. The first column is index 0, the second column is index 1, and the sixth column is index 5.

To exclude the first column from the input, set `InputFilter` to `"$[1:]"`. The colon (:) tells SageMaker to include all of the elements between two values, inclusive. For example, `$[1:4]` specifies the second through fifth columns.

If you omit the number after the colon, for example, `[5:]`, the subset includes all columns from the 6th column through the last column. If you omit the number before the colon, for example, `[5]`, the subset includes all columns from the first column (index 0) through the sixth column.

**Example: Output an ID Attribute with Results and Exclude the ID Attribute from the Input (JSON)**

If you are using the Amazon SageMaker Python SDK, include results of an ID attribute in the output by specifying it in a transformer call. For example, if you store data in the `features` attribute and the record ID in the `ID` attribute, you would use the following transformer request.

```python
sm_transformer = sagemaker.transformer.Transformer(...)  
sm_transformer.transform(..., input_filter="$.features", join_source= "Input", output_filter="$['id','SageMakerOutput']")
```

If you are using the AWS SDK for Python (Boto 3), join all input data with the inference by adding the following code to your `CreateTransformJob` request.

```json
{
   "DataProcessing": {
      "InputFilter": "$.features",
      "JoinSource": "Input",
      "OutputFilter": "$['id','SageMakerOutput']"
   }
}
```

**Warning**

If you are using a JSON-formatted input file, the file can't contain the attribute name `SageMakerOutput`. This attribute name is reserved for the output file. If your JSON-formatted input file contains an attribute with this name, values in the input file might be overwritten with the inference.

### Automatically Scale Amazon SageMaker Models

Amazon SageMaker supports automatic scaling (autoscaling) for your hosted models. Autoscaling dynamically adjusts the number of instances provisioned for a model in response to changes in your workload. When the workload increases, autoscaling brings more instances online. When the workload decreases, autoscaling removes unnecessary instances so that you don't pay for provisioned instances that you aren't using.

**Topics**
- Prerequisites (p. 1331)
- Configure model autoscaling with the console (p. 1334)
Prerequisites

Before you can use autoscaling, you must have already created an Amazon SageMaker model deployment. Deployed models are referred to as a production variant. This includes information about the model and the resources used to host it.

For more information about deploying a model endpoint, see Step 6.1: Deploy the Model to SageMaker Hosting Services (p. 68).

To enable autoscaling for a model, you can use the console, the AWS CLI, or the Application Auto Scaling API. It is recommended to try to Configure model autoscaling with the console (p. 1334) to get familiar with the requirements and to test your first autoscaling configuration. When using the AWS CLI or Application Auto Scaling the flow is to register the model, define the scaling policy, then apply it. The following overview provides further details on the prerequisites and components used with autoscaling.

Autoscaling policy overview

To use automatic scaling, you define and apply a scaling policy that uses Amazon CloudWatch metrics and target values that you assign. Automatic scaling uses the policy to increase or decrease the number of instances in response to actual workloads.

You can use the AWS Management Console to apply a scaling policy based on a predefined metric. A predefined metric is defined in an enumeration so that you can specify it by name in code or use it in the AWS Management Console. Alternatively, you can use either the AWS Command Line Interface (AWS CLI) or the Application Auto Scaling API to apply a scaling policy based on a predefined or custom metric.

There are two types of supported scaling policies: target-tracking scaling and step scaling. It is recommended to use target-tracking scaling policies for your autoscaling configuration. You configure the target-tracking scaling policy by specifying a predefined or custom metric and a target value for the metric. For more information about using Application Auto Scaling target-tracking scaling policies, see Target Tracking Scaling Policies.

You can use step scaling when you require an advanced configuration, such as specifying how many instances to deploy under what conditions. Otherwise, using target-tracking scaling is preferred as it will be fully automated. For more information about using Application Auto Scaling step scaling policies, see Step Scaling Policies.

A scaling policy has the following components:

- A target metric—The Amazon CloudWatch metric that SageMaker automatic scaling uses to determine when and how much to scale.
- Minimum and maximum capacity—The minimum and maximum number of instances to use for scaling.
- A cool down period—The amount of time, in seconds, after a scale-in or scale-out activity completes before another scale-out activity can start.
Required permissions—Permissions that are required to perform automatic scaling actions.

A service-linked role—An AWS Identity and Access Management (IAM) role that is linked to a specific AWS service. A service-linked role includes all of the permissions that the service requires to call other AWS services on your behalf. SageMaker automatic scaling automatically generates this role, AWSServiceRoleForApplicationAutoScaling_SageMakerEndpoint, for you.

Target metric for autoscaling

Amazon CloudWatch alarms trigger the scaling policy, which calculate how to adjust scaling based on the metric and target value that you set. The scaling policy adds or removes endpoint instances as required to keep the metric at, or close to, the specified target value. In addition, a scaling policy also adjusts to fluctuations in the metric when a workload changes. The scaling policy minimizes rapid fluctuations in the number of available instances for your model.

For example, a scaling policy that uses the predefined InvocationsPerInstance metric with a target value of 70 can keep InvocationsPerInstance at, or close to 70.

Minimum and maximum capacity

You may specify the maximum number of endpoint instances for the model. The maximum value must be equal to or greater than the value specified for the minimum number of endpoint instances. SageMaker automatic scaling does not enforce a limit for this value.

You must also specify the minimum number of instances for the model. This value must be at least 1, and equal to or less than the value specified for the maximum number of endpoint instances.

To determine the minimum and maximum number of instances that you need for typical traffic, test your autoscaling configuration with the expected rate of traffic to your model.

Important

Scaling-in does not occur when there is no traffic: if a variant’s traffic becomes zero, SageMaker automatic scaling doesn’t scale in. This is because SageMaker doesn’t emit metrics with a value of zero.

Cooldown period

Tune the responsiveness of your scaling policy by adding a cooldown period. A cooldown period controls when your model is scaled-in (by reducing instances) or scaled-out (by increasing instances). It does this by blocking subsequent scale-in or scale-out requests until the period expires. This slows the deletion of instances for scale-in requests, and the creation of instances for scale-out requests. A cooldown period helps to ensure that the scaling policy doesn't launch or terminate additional instances before the previous scaling activity takes effect. After automatic scaling dynamically scales using a scaling policy, it waits for the cooldown period to complete before resuming scaling activities.

You configure the cooldown period in your automatic scaling policy. You can specify the following cooldown periods:

- A scale-in activity reduces the number of instances. A scale-in cooldown period specifies the amount of time, in seconds, after a scale-in activity completes before another scale-in activity can start.
- A scale-out activity increases the number of instances. A scale-out cooldown period specifies the amount of time, in seconds, after a scale-out activity completes before another scale-out activity can start.

If you don't specify a scale-in or a scale-out cooldown period automatic scaling use the default, which is 300 seconds for each.
If instances are being added or removed too quickly when you test your automatic scaling configuration, consider increasing this value. You can see this behavior if the traffic to your model has a lot of spikes, or if you have multiple automatic scaling policies defined for a variant.

If instances are not being added quickly enough to address increased traffic, consider decreasing this value.

**Permissions**

The SagemakerFullAccessPolicy IAM policy has all of the IAM permissions required to perform autoscaling. For more information about SageMaker IAM permissions, see SageMaker Roles (p. 1647).

If you are using a custom permission policy, you must include the following permissions:

```json
{
  "Effect": "Allow",
  "Action": [
    "sagemaker:DescribeEndpoint",
    "sagemaker:DescribeEndpointConfig",
    "sagemaker:UpdateEndpointWeightsAndCapacities"
  ],
  "Resource": "*"
}
{
  "Action": [
    "application-autoscaling:*
  ],
  "Effect": "Allow",
  "Resource": "*"
}
{
  "Action": "iam:CreateServiceLinkedRole",
  "Effect": "Allow",
  "Resource": "arn:aws:iam::*:role/aws-service-role/sagemaker.application-autoscaling.amazonaws.com/AWSServiceRoleForApplicationAutoScaling_SageMakerEndpoint",
  "Condition": {
    "StringLike": { "iam:AWSServiceName": "sagemaker.application-autoscaling.amazonaws.com" }
  }
}
{
  "Effect": "Allow",
  "Action": [
    "cloudwatch:PutMetricAlarm",
    "cloudwatch:DescribeAlarms",
    "cloudwatch:DeleteAlarms"
  ],
  "Resource": "*"
}
```

**Service-linked role**

Autoscaling uses the AWSServiceRoleForApplicationAutoScaling_SageMakerEndpoint service-linked role; it created for you automatically. A service-linked role is a unique type of IAM role that is linked directly to an AWS service. Service-linked roles are predefined by the service and include all of the permissions that the service requires to call other AWS services on your behalf. For more information, see Service-Linked Roles.
Configure model autoscaling with the console

To configure autoscaling for a model using the console

1. Open the Amazon SageMaker console at https://console.aws.amazon.com/sagemaker/.
2. In the navigation pane, choose Endpoints.
3. Choose the endpoint that you want to configure.
4. For Endpoint runtime settings, choose the model variant that you want to configure.
5. For Endpoint runtime settings, choose Configure autoscaling.

The Configure variant automatic scaling page appears.

6. For Minimum capacity, type the minimum number of instances that you want the scaling policy to maintain. At least 1 instance is required.
7. For Maximum capacity, type the maximum number of instances that you want the scaling policy to maintain.
8. For the target value, type the average number of invocations per instance per minute for the model. To determine this value, follow the guidelines in Load testing (p. 1344).

Application Auto Scaling adds or removes instances to keep the metric close to the value that you specify.

9. For Scale-in cool down (seconds) and Scale-out cool down (seconds), type the number seconds for each cool down period. Assuming that the order in the list is based on either most important to less important of first applied to last applied.
10. Select Disable scale in to prevent the scaling policy from deleting variant instances if you want to ensure that your variant scales out to address increased traffic, but are not concerned with removing instances to reduce costs when traffic decreases, disable scale-in activities.

Scale-out activities are always enabled so that the scaling policy can create endpoint instances as needed.

11. Choose Save.

This procedure registers a model as a scalable target with Application Auto Scaling. When you register a model, Application Auto Scaling performs validation checks to ensure the following:

- The model exists
- The permissions are sufficient
- You aren't registering a variant with an instance that is a burstable performance instance such as T2

Note
SageMaker doesn't support autoscaling for burstable instances such as T2, because they already allow for increased capacity under increased workloads. For information about burstable performance instances, see Amazon EC2 Instance Types.

Register a model

You can add autoscaling for a model with the AWS CLI or the Application Auto Scaling API. You first must register the model, then you must define an autoscaling policy.

Register a model with the AWS CLI

With the AWS CLI, you can configure autoscaling based on either a predefined or a custom metric.
To register your endpoint, use the `register-scalable-target` AWS CLI command with the following parameters:

- `--service-namespace`—Set this value to `sagemaker`.
- `--resource-id`—The resource identifier for the model (specifically, the production variant). For this parameter, the resource type is `endpoint` and the unique identifier is the name of the production variant. For example, `endpoint/MyEndPoint/variant/MyVariant`.
- `--scalable-dimension`—Set this value to `sagemaker:variant:DesiredInstanceCount`.
- `--min-capacity`—The minimum number of instances that for this model. Set `min-capacity` to at least 1. It must be equal to or less than the value specified for `max-capacity`.
- `--max-capacity`—The maximum number of instances that Application Auto Scaling should manage. Set `max-capacity` to a minimum of 1, it must be equal to or greater than the value specified for `min-capacity`.

**Example**

The following example shows how to register a model named `MyVariant` that is dynamically scaled to have one to eight instances:

```
aws application-autoscaling register-scalable-target \
    --service-namespace sagemaker \
    --resource-id endpoint/MyEndPoint/variant/MyVariant \
    --scalable-dimension sagemaker:variant:DesiredInstanceCount \
    --min-capacity 1 \n    --max-capacity 8
```

**Register a model with the Application Auto Scaling API**

To define the scaling limits for the model, register your model with Application Auto Scaling. Application Auto Scaling dynamically scales the number of production variant instances.

To register your model with Application Auto Scaling, use the `RegisterScalableTarget` Application Auto Scaling API action with the following parameters:

- `ServiceNamespace`—Set this value to `sagemaker`.
- `ResourceId`—The resource identifier for the production variant. For this parameter, the resource type is `endpoint` and the unique identifier is the name of the variant, for example `endpoint/MyEndPoint/variant/MyVariant`.
- `ScalableDimension`—Set this value to `sagemaker:variant:DesiredInstanceCount`.
- `MinCapacity`—The minimum number of instances to be managed by Application Auto Scaling. This value must be set to at least 1 and must be equal to or less than the value specified for `MaxCapacity`.
- `MaxCapacity`—The maximum number of instances to be managed by Application Auto Scaling. This value must be set to at least 1 and must be equal to or greater than the value specified for `MinCapacity`.

**Example**

The following example shows how to register a SageMaker production variant that is dynamically scaled to use one to eight instances:

```
POST / HTTP/1.1
Host: autoscaling.us-east-2.amazonaws.com
Accept-Encoding: identity
```
Define a scaling policy

To specify the metrics and target values for a scaling policy, you configure a target-tracking scaling policy. You can use either a predefined metric or a custom metric.

Scaling policy configuration is represented by a JSON block. You save your scaling policy configuration as a JSON block in a text file. You use that file text when invoking the AWS CLI or the Application Auto Scaling API. For more information about policy configuration syntax, see TargetTrackingScalingPolicyConfiguration in the Application Auto Scaling API Reference.

The following options are available for defining a target-tracking scaling policy configuration.

Use a predefined metric

To quickly define a target-tracking scaling policy for a variant, use the SageMakerVariantInvocationsPerInstance predefined metric. SageMakerVariantInvocationsPerInstance is the average number of times per minute that each instance for a variant is invoked. We strongly recommend using this metric.

To use a predefined metric in a scaling policy, create a target tracking configuration for your policy. In the target tracking configuration, include a PredefinedMetricSpecification for the predefined metric and a TargetValue for the target value of that metric.

Example

The following example is a typical policy configuration for target-tracking scaling for a variant. In this configuration, we use the SageMakerVariantInvocationsPerInstance predefined metric to adjust the number of variant instances so that each instance has an InvocationsPerInstance metric of 70.

```
{
  "TargetValue": 70.0,
  "PredefinedMetricSpecification":
  {
    "PredefinedMetricType": "SageMakerVariantInvocationsPerInstance"
  }
}
```

Use a custom metric

If you need to define a target-tracking scaling policy that meets your custom requirements, define a custom metric. You can define a custom metric based on any production variant metric that changes in proportion to scaling.
Not all SageMaker metrics work for target tracking. The metric must be a valid utilization metric, and it must describe how busy an instance is. The value of the metric must increase or decrease in inverse proportion to the number of variant instances. That is, the value of the metric should decrease when the number of instances increases.

**Important**
Before deploying automatic scaling in production, you must test automatic scaling with your custom metric.

**Example**
The following example is a target-tracking configuration for a scaling policy. In this configuration, for a variant named `my-variant`, a custom metric adjusts the variant based on an average CPU utilization of 50 percent across all instances.

```json
{
    "TargetValue": 50,
    "CustomizedMetricSpecification": {
        "MetricName": "CPUUtilization",
        "Namespace": "/aws/sagemaker/Endpoints",
        "Dimensions": [
            {"Name": "EndpointName", "Value": "my-endpoint" },
            {"Name": "VariantName","Value": "my-variant"}
        ],
        "Statistic": "Average",
        "Unit": "Percent"
    }
}
```

**Add a cooldown period**
To add a cooldown period for scaling-out your model, specify a value, in seconds, for `ScaleOutCooldown`. Similarly, to add a cooldown period for scaling-in your model, add a value, in seconds, for `ScaleInCooldown`. For more information about `ScaleInCooldown` and `ScaleOutCooldown`, see [TargetTrackingScalingPolicyConfiguration](https://docs.aws.amazon.com/sagemaker/latest/dg/api-application-auto-scaling.html) in the Application Auto Scaling API Reference.

**Example**
The following is an example target-tracking configuration for a scaling policy. In this configuration, the `SageMakerVariantInvocationsPerInstance` predefined metric is used to adjust scaling based on an average of 70 across all instances of that variant. The configuration provides a scale-in cooldown period of 10 minutes and a scale-out cooldown period of 5 minutes.

```json
{
    "TargetValue": 70.0,
    "PredefinedMetricSpecification": {
        "PredefinedMetricType": "SageMakerVariantInvocationsPerInstance"
    },
    "ScaleInCooldown": 600,
    "ScaleOutCooldown": 300
}
```

**Apply a scaling policy**
After registering your model and defining a scaling policy, apply the scaling policy to the registered model. To apply a scaling policy, you can use the AWS CLI or the Application Auto Scaling API.
Apply a scaling policy (AWS CLI)

To apply a scaling policy to your model, use the `put-scaling-policy` AWS CLI command with the following parameters:

- `--policy-name`—The name of the scaling policy.
- `--policy-type`—Set this value to `TargetTrackingScaling`.
- `--resource-id`—The resource identifier for the variant. For this parameter, the resource type is `endpoint` and the unique identifier is the name of the variant. For example, `endpoint/MyEndpoint/variant/MyVariant`.
- `--service-namespace`—Set this value to `sagemaker`.
- `--scalable-dimension`—Set this value to `sagemaker:variant:DesiredInstanceCount`.
- `--target-tracking-scaling-policy-configuration`—The target-tracking scaling policy configuration to use for the model.

Example

The following example uses Application Auto Scaling to apply a target-tracking scaling policy named `myscalablepolicy` to a model (variant) named `myscalablevariant`. The policy configuration is saved in a file named `config.json`.

```
aws application-autoscaling put-scaling-policy
  --policy-name myscalablepolicy
  --policy-type TargetTrackingScaling
  --resource-id endpoint/MyEndpoint/variant/MyVariant
  --service-namespace sagemaker
  --scalable-dimension sagemaker:variant:DesiredInstanceCount
  --target-tracking-scaling-policy-configuration file://config.json
```

Apply a scaling policy (Application Auto Scaling API)

To apply a scaling policy to a variant with the Application Auto Scaling API, use the `PutScalingPolicy` Application Auto Scaling API action with the following parameters:

- `PolicyName`—The name of the scaling policy.
- `ServiceNamespace`—Set this value to `sagemaker`.
- `ResourceID`—The resource identifier for the variant. For this parameter, the resource type is `endpoint` and the unique identifier is the name of the variant. For example, `endpoint/MyEndpoint/variant/MyVariant`.
- `ScalableDimension`—Set this value to `sagemaker:variant:DesiredInstanceCount`.
- `PolicyType`—Set this value to `TargetTrackingScaling`.
- `TargetTrackingScalingPolicyConfiguration`—The target-tracking scaling policy configuration to use for the variant.

Example

The following example uses Application Auto Scaling to apply a target-tracking scaling policy named `myscalablepolicy` to a variant named `myscalablevariant`. It uses a policy configuration based on the `SageMakerVariantInvocationsPerInstance` predefined metric.

```
POST / HTTP/1.1
Host: autoscaling.us-east-2.amazonaws.com
Accept-Encodint: identity
X-Amz-Target: AnyScaleFrontendService.
```
Edit a scaling policy

You can edit an autoscaling policy with the AWS Management Console, the AWS CLI, or the Application Auto Scaling API.

Scale-in

If a model's traffic becomes zero, Amazon SageMaker automatic scaling doesn't scale-in. This is because SageMaker doesn't emit metrics with a value of zero, and without a metric, the scaling policy will never be triggered.

As a workaround, do either of the following:

- Send requests to the model variant until autoscaling scales-in to the minimum capacity
- Change the policy to reduce the maximum provisioned capacity to match the minimum provisioned capacity

Disable scale-in activity

You can prevent the target-tracking scaling policy configuration from scaling in your variant by disabling scale-in activity. Disabling scale-in activity prevents the scaling policy from deleting instances, while still allowing it to create them as needed.

To enable or disable scale-in activity for your model, specify a Boolean value for DisableScaleIn. For more information about DisableScaleIn, see TargetTrackingScalingPolicyConfiguration in the Application Auto Scaling API Reference.

Example

The following is an example of a target-tracking configuration for a scaling policy where it will scale-out, but not scale-in. In this configuration, the SageMakerVariantInvocationsPerInstance predefined metric will scale-out based on an average of 70 invocations (inference requests) across all instances the model is on. The configuration also disables scale-in activity for the scaling policy.

```json
{
  "TargetValue": 70.0,
  "PredefinedMetricSpecification":
  {
    "PredefinedMetricType": "SageMakerVariantInvocationsPerInstance"
  }
}
```
Delete a scaling policy

You can delete a scaling policy with the AWS Management Console, the AWS CLI, or the Application Auto Scaling API. You must delete a scaling policy if you wish to update a model’s endpoint.

Delete a scaling policy (Console)

To delete an automatic scaling policy (console)
1. Open the Amazon SageMaker console at https://console.aws.amazon.com/sagemaker/.
2. In the navigation pane, choose **Endpoints**.
3. Choose the endpoint for which you want to delete automatic scaling.
4. For **Endpoint runtime settings**, choose the variant that you want to configure.
5. Choose **Configure auto scaling**.
6. Choose **Deregister auto scaling**.

Delete a scaling policy (AWS CLI or Application Auto Scaling API)

You can use the AWS CLI or the Application Auto Scaling API to delete a scaling policy from a variant.
Delete a scaling policy (AWS CLI)

To delete a scaling policy from a variant, use the `delete-scaling-policy` AWS CLI command with the following parameters:

- `--policy-name`—The name of the scaling policy.
- `--resource-id`—The resource identifier for the variant. For this parameter, the resource type is endpoint and the unique identifier is the name of the variant. For example, `endpoint/MyEndpoint/variant/MyVariant`.
- `--service-namespace`—Set this value to `sagemaker`.
- `--scalable-dimension`—Set this value to `sagemaker:variant:DesiredInstanceCount`.

Example

The following example deletes a target-tracking scaling policy named `myscalablepolicy` from a variant named `myscalablevariant`.

```bash
aws application-autoscaling delete-scaling-policy \
--policy-name myscalablepolicy \
--resource-id endpoint/MyEndpoint/variant/MyVariant \
--service-namespace sagemaker \
--scalable-dimension sagemaker:variant:DesiredInstanceCount
```

Delete a scaling policy (Application Auto Scaling API)

To delete a scaling policy from your variant, use the `DeleteScalingPolicy` Application Auto Scaling API action with the following parameters:

- `PolicyName`—The name of the scaling policy.
- `ServiceNamespace`—Set this value to `sagemaker`.
- `ResourceId`—The resource identifier for the variant. For this parameter, the resource type is endpoint and the unique identifier is the name of the variant. For example, `endpoint/MyEndpoint/variant/MyVariant`.
- `ScalableDimension`—Set this value to `sagemaker:variant:DesiredInstanceCount`.

Example

The following example uses the Application Auto Scaling API to delete a target-tracking scaling policy named `myscalablepolicy` from a variant named `myscalablevariant`.

```json
POST / HTTP/1.1
Host: autoscaling.us-east-2.amazonaws.com
Accept-Encoding: identity
X-Amz-Target: AnyScaleFrontendService.DeleteScalingPolicy
X-Amz-Date: 20160506T182145Z
User-Agent: aws-cli/1.10.23 Python/2.7.11 Darwin/15.4.0 botocore/1.4.8
Content-Type: application/x-amz-json-1.1
Authorization: AUTHPARAMS

{
  "PolicyName": "myscalablepolicy",
  "ServiceNamespace": "sagemaker",
  "ResourceId": "endpoint/MyEndpoint/variant/MyVariant",
  "ScalableDimension": "sagemaker:variant:DesiredInstanceCount"
}
```
Query Endpoint Autoscaling History

You can view the status of scaling activities from your endpoint using `DescribeScalingActivities`. `DescribeScalingActivities` provides descriptive information about the scaling activities in the specified namespace from the previous six weeks.

How To Query Endpoint Autoscaling Actions

Query your autoscaling endpoints with `DescribeScalingActivities`. To do so, specify `ServiceNameSpace` parameter. `ServiceNameSpace` is the name of the AWS service that provides the resource.

Valid service name values include the following:

ecs | elasticmapreduce | ec2 | appstream | dynamodb | rds | sagemaker | custom-resource | comprehend | lambda | cassandra

In this situation you need to set `ServiceNameSpace` to `sagemaker`.

Use the following AWS CLI command to view details about all of your `sagemaker` endpoints that have a scaling policy:

```bash
aws application-autoscaling describe-scaling-activities \
    --service-namespace sagemaker
```

You can search for a specific endpoint using `ResourceId`:

```bash
aws application-autoscaling describe-scaling-activities \
    --service-namespace sagemaker \
    --resource-id endpoint/<endpoint_name>/variant/<variant_name>
```

When you run this command, it returns the following output:

```json
{
    "ActivityId": "activity-id",
    "ServiceNamespace": "sagemaker",
    "ResourceId": "endpoint/<endpoint_name>/variant/<variant_name>",
    "ScalableDimension": "sagemaker:variant:DesiredInstanceCount",
    "Description": "string",
    "Cause": "string",
    "StartTime": timestamp,
    "EndTime": timestamp,
    "StatusCode": "string",
    "StatusMessage": "string"
}
```

How to Identify Blocked AutoScaling Due to Instance Quotas

When you scale out or add more instances, you might reach your account-level instance quota. You can use `DescribeScalingActivities` to check whether you have reached your instance quota. When you exceed your quota, automatic scaling is blocked.

To check if you have reached your instance quota, use the AWS CLI command as shown in the proceeding example where you specified the `ResourceId`:

```bash
aws application-autoscaling describe-scaling-activities \
    --service-namespace sagemaker \
    --resource-id endpoint/<endpoint_name>/variant/<variant_name>
```
Update or delete endpoints that use automatic scaling

Topics

• Update endpoints that use automatic scaling (p. 1343)
• Delete endpoints configured for automatic scaling (p. 1344)

Update endpoints that use automatic scaling

When you update an endpoint, Application Auto Scaling checks to see whether any of the models on that endpoint are targets for automatic scaling. If the update would change the instance type for any model that is a target for automatic scaling, the update fails.

In the AWS Management Console, you see a warning that you must deregister the model from automatic scaling before you can update it. If you are trying to update the endpoint by calling the UpdateEndpoint API, the call fails. Before you update the endpoint, delete any scaling policies configured for it by calling the DeleteScalingPolicy Application Auto Scaling API action, then call DeregisterScalableTarget to deregister the variant as a scalable target. After you update the endpoint, you can register the variant as a scalable target and attach an automatic scaling policy to the updated variant.

There is one exception. If you change the model for a variant that is configured for automatic scaling, Amazon SageMaker automatic scaling allows the update. This is because changing the model doesn’t typically affect performance enough to change automatic scaling behavior. If you do update a model for a variant configured for automatic scaling, ensure that the change to the model doesn’t significantly affect performance and automatic scaling behavior.

When you update SageMaker endpoints that have automatic scaling applied, complete the following steps:

To update an endpoint that has automatic scaling applied

1. Deregister the endpoint as a scalable target by calling DeregisterScalableTarget.
2. Because automatic scaling is blocked while the update operation is in progress (or if you turned off automatic scaling in the previous step), you might want to take the additional precaution of increasing the number of instances for your endpoint during the update. To do this, update the instance counts for the production variants hosted at the endpoint by calling `UpdateEndpointWeightsAndCapacities`.

3. Call `DescribeEndpoint` repeatedly until the value of the `EndpointStatus` field of the response is `InService`.

4. Call `DescribeEndpointConfig` to get the values of the current endpoint config.

5. Create a new endpoint config by calling `CreateEndpointConfig`. For the production variants where you want to keep the existing instance count or weight, use the same variant name from the response from the call to `DescribeEndpointConfig` in the previous step. For all other values, use the values that you got as the response when you called `DescribeEndpointConfig` in the previous step.

6. Update the endpoint by calling `UpdateEndpoint`. Specify the endpoint config you created in the previous step as the `EndpointConfig` field. If you want to retain the variant properties like instance count or weight, set the value of the `RetainAllVariantProperties` parameter to `True`. This specifies that production variants with the same name will are updated with the most recent `DesiredInstanceCount` from the response from the call to `DescribeEndpoint`, regardless of the values of the `InitialInstanceCount` field in the new `EndpointConfig`.

7. (Optional) Re-enable automatic scaling by calling `RegisterScalableTarget`.

   **Note**
   Steps 1 and 7 are required only if you are updating an endpoint with the following changes:
   - Changing the instance type for a production variant that has automatic scaling configured
   - Removing a production variant that has automatic scaling configured.

### Delete endpoints configured for automatic scaling

If you delete an endpoint, Application Auto Scaling checks to see whether any of the models on that endpoint are targets for automatic scaling. If any are and you have permission to deregister the model, Application Auto Scaling deregisters those models as scalable targets without notifying you. If you use a custom permission policy that doesn't provide permission for the `DeleteScalingPolicy` and `DeregisterScalableTarget` actions, you must delete automatic scaling policies and deregister scalable targets and before deleting the endpoint.

   **Note**
   You, as an IAM user, might not have sufficient permission to delete an endpoint if another IAM user configured automatic scaling for a variant on that endpoint.

### Load testing your autoscaling configuration

Perform load tests to choose an automatic scaling configuration that works the way you want.

For an example of load testing to optimize automatic scaling for an Amazon SageMaker endpoint, see [Load test and optimize an Amazon SageMaker endpoint using automatic scaling](#).

The following guidelines for load testing assume you are using an automatic scaling policy that uses the predefined target metric `SageMakerVariantInvocationsPerInstance`.

**Topics**
- Determine the performance characteristics (p. 1345)
- Calculate the target load (p. 1345)
Determine the performance characteristics

Perform load testing to find the peak $\text{InvocationsPerInstance}$ that your model’s production variant can handle, and the latency of requests, as concurrency increases.

This value depends on the instance type chosen, payloads that clients of your model typically send, and the performance of any external dependencies your model has.

To find the peak requests-per-second (RPS) your model's production variant can handle and latency of requests

1. Set up an endpoint with your model using a single instance. For information about how to set up an endpoint, see Step 6.1: Deploy the Model to SageMaker Hosting Services (p. 68).
2. Use a load testing tool to generate an increasing number of parallel requests, and monitor the RPS and model latency in the output of the load testing tool.

   **Note**
   You can also monitor requests-per-minute instead of RPS. In that case don't multiply by 60 in the equation to calculate $\text{SageMakerVariantInvocationsPerInstance}$ shown below.

   When the model latency increases or the proportion of successful transactions decreases, this is the peak RPS that your model can handle.

Calculate the target load

After you find the performance characteristics of the variant, you can determine the maximum RPS we should allow to be sent to an instance. The threshold used for scaling must be less than this maximum value. Use the following equation in combination with load testing to determine the correct value for the $\text{SageMakerVariantInvocationsPerInstance}$ target metric in your automatic scaling configuration.

$$\text{SageMakerVariantInvocationsPerInstance} = (\text{MAX\_RPS} \times \text{SAFETY\_FACTOR}) \times 60$$

Where $\text{MAX\_RPS}$ is the maximum RPS that you determined previously, and $\text{SAFETY\_FACTOR}$ is the safety factor that you chose to ensure that your clients don’t exceed the maximum RPS. Multiply by 60 to convert from RPS to invocations-per-minute to match the per-minute CloudWatch metric that SageMaker uses to implement automatic scaling (you don’t need to do this if you measured requests-per-minute instead of requests-per-second).

   **Note**
   SageMaker recommends that you start testing with a $\text{SAFETY\_FACTOR}$ of 0.5. Test your automatic scaling configuration to ensure it operates in the way you expect with your model for both increasing and decreasing customer traffic on your endpoint.

Use AWS CloudFormation to update autoscaling policies

The following is an example for how to enable autoscaling on an endpoint using AWS CloudFormation.

```json
Endpoint:
  Type: "AWS::SageMaker::Endpoint"
Properties:
  EndpointName: yourEndpointName
  EndpointConfigName: yourEndpointConfigName
```
Test models in production

In production ML workflows, data scientists and engineers frequently try to improve their models in various ways, such as by performing Perform Automatic Model Tuning (p. 1023), training on additional or more-recent data, and improving feature selection. Performing A/B testing between a new model and an old model with production traffic can be an effective final step in the validation process for a new model. In A/B testing, you test different variants of your models and compare how each variant performs. If the newer version of the model delivers better performance than the previously-existing version, replace the old version of the model with the new version in production.

Amazon SageMaker enables you to test multiple models or model versions behind the same endpoint using production variants. Each production variant identifies a machine learning (ML) model and the resources deployed for hosting the model. By using production variants, you can test ML models that have been trained using different datasets, trained using different algorithms and ML frameworks, or are deployed to different instance type, or any combination of all of these. You can distribute endpoint invocation requests across multiple production variants by providing the traffic distribution for each variant, or you can invoke a specific variant directly for each request. In this topic, we look at both methods for testing ML models.

Topics
- Test models by specifying traffic distribution (p. 1346)
- Test models by invoking specific variants (p. 1347)
- Model A/B test example (p. 1348)

Test models by specifying traffic distribution

To test multiple models by distributing traffic between them, specify the percentage of the traffic that gets routed to each model by specifying the weight for each production variant in the endpoint configuration. For information, see CreateEndpointConfig. The following diagram shows how this works in more detail.
Test models by invoking specific variants

To test multiple models by invoking specific models for each request, specify the specific version of the model you want to invoke by providing a value for the TargetVariant parameter when you call `InvokeEndpoint`. SageMaker ensures that the request is processed by the production variant you specify. If you have already provided traffic distribution and specify a value for the TargetVariant parameter, the targeted routing overrides the random traffic distribution. The following diagram shows how this works in more detail.
Model A/B test example

The following example shows how to perform A/B model testing. For a sample notebook that implements this example, see "A/B Testing ML models in production."

Step 1: Create and deploy models

First, we define where our models are located in Amazon S3. These locations are used when we deploy our models in subsequent steps:

```python
model_url = f"s3://{path_to_model_1}"
model_url2 = f"s3://{path_to_model_2}"
```

Next, we create the model objects with the image and model data. These model objects are used to deploy production variants on an endpoint. The models are developed by training ML models on different data sets, different algorithms or ML frameworks, and different hyperparameters:

```python
from sagemaker.amazon.amazon_estimator import get_image_uri

model_name = f"DEMO-xgb-churn-pred-{datetime.now():%Y-%m-%d-%H-%M-%S}" 
model_name2 = f"DEMO-xgb-churn-pred2-{datetime.now():%Y-%m-%d-%H-%M-%S}" 
```
image_uri = get_image_uri(boto3.Session().region_name, 'xgboost', '0.90-1')
image_uri2 = get_image_uri(boto3.Session().region_name, 'xgboost', '0.90-2')

sm_session.create_model(name=model_name, role=role, container_defs={
    'Image': image_uri,
    'ModelDataUrl': model_url
})

sm_session.create_model(name=model_name2, role=role, container_defs={
    'Image': image_uri2,
    'ModelDataUrl': model_url2
})

We now create two production variants, each with its own different model and resource requirements (instance type and counts). This enables you to also test models on different instance types.

We set an initial_weight of 1 for both variants. This means that 50% of requests go to Variant1, and the remaining 50% of requests to Variant2. The sum of weights across both variants is 2 and each variant has weight assignment of 1. This means that each variant receives 1/2, or 50%, of the total traffic.

from sagemaker.session import production_variant

variant1 = production_variant(model_name=model_name,
    instance_type="ml.m5.xlarge",
    initial_instance_count=1,
    variant_name='Variant1',
    initial_weight=1)

variant2 = production_variant(model_name=model_name2,
    instance_type="ml.m5.xlarge",
    initial_instance_count=1,
    variant_name='Variant2',
    initial_weight=1)

Finally we’re ready to deploy these production variants on a SageMaker endpoint.

endpoint_name = f"DEMO-xgb-churn-pred-{datetime.now():%Y-%m-%d-%H-%M-%S}" 
print(f"EndpointName={endpoint_name}"

sm_session.endpoint_from_production_variants(
    name=endpoint_name,
    production_variants=[variant1, variant2]
)

**Step 2: Invoke the deployed models**

Now we send requests to this endpoint to get inferences in real time. We use both traffic distribution and direct targeting.

First, we use traffic distribution that we configured in the previous step. Each inference response contains the name of the production variant that processes the request, so we can see that traffic to the two production variants is roughly equal.

# get a subset of test data for a quick test
!tail -120 test_data/test-dataset-input-cols.csv > test_data/
test_sample_tail_input_cols.csv
print(f"Sending test traffic to the endpoint {endpoint_name}. \nPlease wait...")
with open('test_data/test_sample_tail_input_cols.csv', 'r') as f:
    for row in f:
        print('.', end='', flush=True)
        payload = row.rstrip('
')
        sm_runtime.invoke_endpoint(EndpointName=endpoint_name,
            ContentType="text/csv",
            Body=payload)
        time.sleep(0.5)
print("Done!")

SageMaker emits metrics such as Latency and Invocations for each variant in Amazon CloudWatch. For a complete list of metrics that SageMaker emits, see Monitor Amazon SageMaker with Amazon CloudWatch (p. 1704). Let's query CloudWatch to get the number of invocations per variant, to show how invocations are split across variants by default:

<table>
<thead>
<tr>
<th>Variant1</th>
<th>Variant2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Timestamp</td>
<td></td>
</tr>
<tr>
<td>2020-06-05 15:34:00-00:00</td>
<td>44.0</td>
</tr>
<tr>
<td>2020-06-05 15:35:00-00:00</td>
<td>7.0</td>
</tr>
</tbody>
</table>

Now let's invoke a specific version of the model by specifying Variant1 as the TargetVariant in the call to invoke_endpoint.

print(f"Sending test traffic to the endpoint {endpoint_name}. \nPlease wait...")
with open('test_data/test_sample_tail_input_cols.csv', 'r') as f:
    for row in f:
        print('.', end='', flush=True)
        payload = row.rstrip('
')
        sm_runtime.invoke_endpoint(EndpointName=endpoint_name,
            ContentType="text/csv",
            Body=payload,
            TargetVariant="Variant1") # Notice the new parameter
        time.sleep(0.5)

To confirm that all new invocations were processed by Variant1, we can query CloudWatch to get the number of invocations per variant. We see that for the most recent invocations (latest timestamp), all requests were processed by Variant1, as we had specified. There were no invocations made for Variant2.
Step 3: Evaluate model performance

To see which model version performs better, let's evaluate the accuracy, precision, recall, F1 score, and Receiver operating characteristic/Area under the curve for each variant. First, let's look at these metrics for Variant1:

- Accuracy: 0.958333333333334
- Precision: 0.9411764705882353
- Recall: 0.8
- F1 Score: 0.8648484848484848
- AUC is 0.895

Now let's look at the metrics for Variant2:
For most of our defined metrics, Variant2 is performing better, so this is the one that we want to use in production.

**Step 4: Increase traffic to the best model**

Now that we have determined that Variant2 performs better than Variant1, we shift more traffic to it. We can continue to use TargetVariant to invoke a specific model variant, but a simpler approach is to update the weights assigned to each variant by calling UpdateEndpointWeightsAndCapacities. This changes the traffic distribution to your production variants without requiring updates to your endpoint. Recall from the setup section that we set variant weights to split traffic 50/50. The CloudWatch metrics for the total invocations for each variant below show us the invocation patterns for each variant:

Now we shift 75% of the traffic to Variant2 by assigning new weights to each variant using UpdateEndpointWeightsAndCapacities. SageMaker now sends 75% of the inference requests to Variant2 and remaining 25% of requests to Variant1.
The CloudWatch metrics for total invocations for each variant shows us higher invocations for Variant2 than for Variant1:

We can continue to monitor our metrics, and when we're satisfied with a variant's performance, we can route 100% of the traffic to that variant. We use `UpdateEndpointWeightsAndCapacities` to update the traffic assignments for the variants. The weight for Variant1 is set to 0 and the weight for Variant2 is set to 1. SageMaker now sends 100% of all inference requests to Variant2.

```python
sm.update_endpoint_weights_and_capacities(
    EndpointName=endpoint_name,
    DesiredWeightsAndCapacities=[
        {
            "DesiredWeight": 0,
            "VariantName": variant1["VariantName"]
        },
        {
            "DesiredWeight": 1,
            "VariantName": variant2["VariantName"]
        }
    ]
)
```

The CloudWatch metrics for the total invocations for each variant show that all inference requests are being processed by Variant2 and there are no inference requests processed by Variant1.

You can now safely update your endpoint and delete Variant1 from your endpoint. You can also continue testing new models in production by adding new variants to your endpoint and following steps 2 - 4.
Troubleshoot Amazon SageMaker Model Deployments

If you encounter an issue when deploying machine learning models in Amazon SageMaker, see the following guidance.

**Topics**
- Detection Errors in the Active CPU Count (p. 1354)

### Detection Errors in the Active CPU Count

If you deploy a SageMaker model with a Linux Java Virtual Machine (JVM), you might encounter detection errors that prevent using available CPU resources. This issue affects some JVMs that support Java 8 and Java 9, and most that support Java 10 and Java 11. These JVMs implement a mechanism that detects and handles the CPU count and the maximum memory available when running a model in a Docker container, and, more generally, within Linux `taskset` commands or control groups (cgroups). SageMaker deployments take advantage of some of the settings that the JVM uses for managing these resources. Currently, this causes the container to incorrectly detect the number of available CPUs.

SageMaker doesn’t limit access to CPUs on an instance. However, the JVM might detect the CPU count as 1 when more CPUs are available for the container. As a result, the JVM adjusts all of its internal settings to run as if only 1 CPU core is available. These settings affect garbage collection, locks, compiler threads, and other JVM internals that negatively affect the concurrency, throughput, and latency of the container.

For an example of the misdetection, in a container configured for SageMaker that is deployed with a JVM that is based on Java8_191 and that has four available CPUs on the instance, run the following command to start your JVM:

```bash
java -XX:+UnlockDiagnosticVMOptions -XX:+PrintActiveCpus -version
```

This generates the following output:

```
active_processor_count: sched_getaffinity processor count: 4
active_processor_count: determined by OSContainer: 1
active_processor_count: sched_getaffinity processor count: 4
active_processor_count: determined by OSContainer: 1
active_processor_count: sched_getaffinity processor count: 4
active_processor_count: determined by OSContainer: 1
active_processor_count: sched_getaffinity processor count: 4
active_processor_count: determined by OSContainer: 1
openjdk version "1.8.0_191"
OpenJDK Runtime Environment (build 1.8.0_191-b12-2ubuntu0.16.04.1-b12)
OpenJDK 64-Bit Server VM (build 25.191-b12, mixed mode)
```

Many of the JVMs affected by this issue have an option to disable this behavior and reestablish full access to all of the CPUs on the instance. Disable the unwanted behavior and establish full access to all instance CPUs by including the `-XX:-UseContainerSupport` parameter when starting Java applications. For example, run the `java` command to start your JVM as follows:

```bash
java -XX:-UseContainerSupport -XX:+UnlockDiagnosticVMOptions -XX:+PrintActiveCpus -version
```

This generates the following output:

```
active_processor_count: sched_getaffinity processor count: 4
```
Check whether the JVM used in your container supports the `-XX:-UseContainerSupport` parameter. If it does, always pass the parameter when you start your JVM. This provides access to all of the CPUs in your instances.

You might also encounter this issue when indirectly using a JVM in SageMaker containers. For example, when using a JVM to support SparkML Scala. The `-XX:-UseContainerSupport` parameter also affects the output returned by the Java `Runtime.getRuntime().availableProcessors()` API.

**Deployment Best Practices**

This topic provides guidance on best practices for deploying machine learning models in Amazon SageMaker.

### Deploy Multiple Instances Across Avaliability Zones

**Create robust endpoints when hosting your model.** SageMaker endpoints can help protect your application from Availability Zone outages and instance failures. If an outage occurs or an instance fails, SageMaker automatically attempts to distribute your instances across Availability Zones. For this reason, we strongly recommended that you deploy multiple instances for each production endpoint.

If you are using an Amazon Virtual Private Cloud (VPC), configure the VPC with at least two Subnets, each in a different Availability Zone. If an outage occurs or an instance fails, Amazon SageMaker automatically attempts to distribute your instances across Availability Zones.

In general, to achieve more reliable performance, use more small Instance Types in different Availability Zones to host your endpoints.

### Host Instance Storage Volumes

When you create an endpoint, Amazon SageMaker attaches an Amazon Elastic Block Store (Amazon EBS) storage volume to Amazon EC2 instances that hosts the endpoint. The size of the storage volume is scalable, and storage options are divided into two categories: SSD-backed storage and HDD-backed storage.

For more information about Amazon EBS storages and features, see the following pages.

- Amazon EBS Features
- Amazon EBS User Guide

For a full list of the host instance storage volumes, see Host Instance Storage Volumes Table
Using Docker containers with SageMaker

Amazon SageMaker makes extensive use of Docker containers for build and runtime tasks. SageMaker provides prebuilt Docker images for its built-in algorithms and the supported deep learning frameworks used for training and inference. Using containers, you can train machine learning algorithms and deploy models quickly and reliably at any scale. The topics in this section show how to deploy these containers for your own use cases.

Topics
- Scenarios for Running Scripts, Training Algorithms, or Deploying Models with SageMaker (p. 1356)
- Docker Container Basics (p. 1357)
- Use Prebuilt SageMaker Docker images (p. 1357)
- Adapting Your Own Docker Container to Work with SageMaker (p. 1371)
- Create a container with your own algorithms and models (p. 1383)
- Example Notebooks: Use Your Own Algorithm or Model (p. 1394)

Scenarios for Running Scripts, Training Algorithms, or Deploying Models with SageMaker

Amazon SageMaker always uses Docker containers when running scripts, training algorithms, and deploying models. However, your level of engagement with containers depends on your use case.

- **Use a built-in SageMaker algorithm or framework.** For most use cases, you can use the built-in algorithms and frameworks without worrying about containers. You can train and deploy these algorithms from the SageMaker console, the AWS Command Line Interface (AWS CLI), a Python notebook, or the Amazon SageMaker Python SDK by specifying the algorithm or framework version when creating your Estimator. The built-in algorithms available are itemized and described in the Use Amazon SageMaker Built-in Algorithms (p. 635) topic. For more information on the available frameworks, see ML Frameworks, Python & R (p. 17). For an example of how to train and deploy a built-in algorithm using a Jupyter notebook running in a SageMaker notebook instance, see the Get Started with Amazon SageMaker (p. 33) topic.

- **Use prebuilt SageMaker container images.** Alternatively, you can use the built-in algorithms and frameworks using Docker containers. SageMaker provides containers for its built-in algorithms and prebuilt Docker images for some of the most common machine learning frameworks, such as Apache MXNet, TensorFlow, PyTorch, and Chainer. For a full list of the available SageMaker Images, see Available Deep Learning Containers Images. It also supports machine learning libraries such as scikit-learn and SparkML. If you use the Amazon SageMaker Python SDK, you can deploy the containers by passing the full container URI to their respective SageMaker SDK Estimator class. For the full list of deep learning frameworks currently supported by SageMaker, see Prebuilt SageMaker Docker Images for TensorFlow, MXNet, Chainer, and PyTorch (p. 1358). For information on the scikit-learn and SparkML prebuilt container images, see Prebuilt Amazon SageMaker Docker Images for Scikit-learn and Spark ML (p. 1358). For more information about using frameworks with the Amazon SageMaker...
• **Extend a prebuilt SageMaker container image.** If you would like to extend a prebuilt SageMaker algorithm or model Docker image, you can modify the SageMaker image to satisfy your needs. For an example, see Extending our PyTorch containers.

• **Adapt an existing container image:** If you would like to adapt a pre-existing container image to work with SageMaker, you need to modify the Docker container to enable either the SageMaker Training or Inference toolkit. For an example that shows how to build your own containers to train and host an algorithm, see Bring Your Own R Algorithm.

---

**Docker Container Basics**

Docker is a program that performs operating system-level virtualization for installing, distributing, and managing software. It packages applications and their dependencies into virtual containers that provide isolation, portability, and security. With Docker, you can ship code faster, standardize application operations, seamlessly move code, and economize by improving resource utilization. For more general information about Docker, see Docker overview.

The following information outlines the most significant aspects of using Docker containers with Amazon SageMaker.

**SageMaker Functions**

SageMaker uses Docker containers in the backend to manage training and inference processes. SageMaker abstracts away from this process, so it happens automatically when an estimator is used. While you don't need to use Docker containers explicitly with SageMaker for most use cases, you can use Docker containers to extend and customize SageMaker functionality.

**Containers with SageMaker Studio**

SageMaker Studio runs from a Docker container and uses it to manage functionality. As a result, you cannot create and upload a Docker container from a SageMaker Studio instance. However, you can use a prebuilt SageMaker container as long as that container was created outside of Studio.

**Use Prebuilt SageMaker Docker images**

Amazon SageMaker provides containers for its built-in algorithms and prebuilt Docker images for some of the most common machine learning frameworks, such as Apache MXNet, TensorFlow, PyTorch, and Chainer. It also supports machine learning libraries such as scikit-learn and SparkML.

You can use these images from your SageMaker notebook instance or SageMaker Studio. You can also extend the prebuilt SageMaker images to include libraries and needed functionality. The following topics give information about the available images and how to use them.

**Note**

For information on Docker images for developing reinforcement learning (RL) solutions in SageMaker, see SageMaker RL Containers.

**Topics**

- Prebuilt SageMaker Docker Images for TensorFlow, MXNet, Chainer, and PyTorch (p. 1358)
- Prebuilt Amazon SageMaker Docker Images for Scikit-learn and Spark ML (p. 1358)
- Train a Deep Graph Network (p. 1360)
- Extend a Prebuilt Container (p. 1363)
Prebuilt SageMaker Docker Images for TensorFlow, MXNet, Chainer, and PyTorch

SageMaker provides prebuilt Docker images that include deep learning framework libraries and other dependencies needed for training and inference. For a complete list of the available pre-built Docker images, see Deep Learning Containers Images.

If you are not using the Amazon SageMaker Python SDK and one of its estimators to retrieve the pre-built images, you have to retrieve them yourself.

Using the SageMaker Python SDK

With the SageMaker Python SDK, you can train and deploy models using these popular deep learning frameworks. For instructions on installing and using the SDK, see Amazon SageMaker Python SDK. The following table lists the available frameworks and instructions on how to use them with the SageMaker Python SDK:

<table>
<thead>
<tr>
<th>Framework</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>TensorFlow</td>
<td>Using TensorFlow with the SageMaker Python SDK</td>
</tr>
<tr>
<td>MXNet</td>
<td>Using MXNet with the SageMaker Python SDK</td>
</tr>
<tr>
<td>PyTorch</td>
<td>Using PyTorch with the SageMaker Python SDK</td>
</tr>
<tr>
<td>Chainer</td>
<td>Using Chainer with the SageMaker Python SDK</td>
</tr>
</tbody>
</table>

Extending Prebuilt SageMaker Docker Images

You can customize these prebuilt containers or extend them to handle any additional functional requirements for your algorithm or model that the prebuilt SageMaker Docker image doesn't support. For an example, see Extending Our PyTorch Containers.

You can also use prebuilt containers to deploy your custom models or models that have been trained in a framework other than SageMaker. For an overview of the process of bringing the trained model artifacts into SageMaker and hosting them at an endpoint, see Bring Your Own Pretrained MXNet or TensorFlow Models into Amazon SageMaker.

Prebuilt Amazon SageMaker Docker Images for Scikit-learn and Spark ML

SageMaker provides prebuilt Docker images that install the scikit-learn and Spark ML libraries. These libraries also include the dependencies needed to build Docker images that are compatible with SageMaker using the Amazon SageMaker Python SDK. With the SDK, you can use scikit-learn for machine learning tasks and use Spark ML to create and tune machine learning pipelines. For instructions on installing and using the SDK, see SageMaker Python SDK.

Using the SageMaker Python SDK

The following table contains links to the GitHub repositories with the source code for the scikit-learn and Spark ML containers. The table also contains links to instructions that show how use these containers with Python SDK estimators to run your own training algorithms and hosting your own models.
### Specifying the Prebuilt Images Manually

If you are not using the SageMaker Python SDK and one of its estimators to manage the container, you have to retrieve the relevant prebuilt container manually. The SageMaker prebuilt Docker images are stored in Amazon Elastic Container Registry (Amazon ECR). You can push or pull them using their fullname registry addresses. SageMaker uses the following Docker Image URL patterns for scikit-learn and Spark ML:

- `<ACCOUNT_ID>.dkr.ecr.<REGION_NAME>.amazonaws.com/sagemaker-scikit-learn:<SCIKIT-LEARN_VERSION>-cpu-py<PYTHON_VERSION>`
  
  For example, `746614075791.dkr.ecr.us-west-1.amazonaws.com/sagemaker-scikit-learn:0.23-1-cpu-py3`

- `<ACCOUNT_ID>.dkr.ecr.<REGION_NAME>.amazonaws.com/sagemaker-sparkml-serving:<SPARK-ML_VERSION>`

  For example, `341280168497.dkr.ecr.ca-central-1.amazonaws.com/sagemaker-sparkml-serving:2.4`

The following table lists the supported values for account IDs and corresponding AWS Region names.

<table>
<thead>
<tr>
<th>ACCOUNT_ID</th>
<th>REGION_NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>746614075791</td>
<td>us-west-1</td>
</tr>
<tr>
<td>246618743249</td>
<td>us-west-2</td>
</tr>
<tr>
<td>683313688378</td>
<td>us-east-1</td>
</tr>
<tr>
<td>257758044811</td>
<td>us-east-2</td>
</tr>
<tr>
<td>354813040037</td>
<td>ap-northeast-1</td>
</tr>
<tr>
<td>366743142698</td>
<td>ap-northeast-2</td>
</tr>
<tr>
<td>121021644041</td>
<td>ap-southeast-1</td>
</tr>
<tr>
<td>783357654285</td>
<td>ap-southeast-2</td>
</tr>
<tr>
<td>720646828776</td>
<td>ap-south-1</td>
</tr>
<tr>
<td>141502667606</td>
<td>eu-west-1</td>
</tr>
<tr>
<td>764974769150</td>
<td>eu-west-2</td>
</tr>
<tr>
<td>492215442770</td>
<td>eu-central-1</td>
</tr>
<tr>
<td>341280168497</td>
<td>ca-central-1</td>
</tr>
</tbody>
</table>
ACCOUNT_ID   REGION_NAME
414596584902   us-gov-west-1

Finding Available Images

Use the following commands to find out which versions of the images are available. For example, use the following to find the available sagemaker-sparkml-serving image in the ca-central-1 Region:

```bash
$ aws ecr describe-images
  --region ca-central-1
  --registry-id 341280168497
  --repository-name sagemaker-sparkml-serving
```

Train a Deep Graph Network

In this overview, you learn how to get started with a deep graph network by using one of the DGL containers in Amazon Elastic Container Registry (Amazon ECR). You can also see links to practical examples for deep graph networks.

What Is a Deep Graph Network?

Deep graph networks refer to a type of neural network that is trained to solve graph problems. A deep graph network uses an underlying deep learning framework like PyTorch or MXNet. The potential for graph networks in practical AI applications is highlighted in the Amazon SageMaker tutorials for Deep Graph Library (DGL). Examples for training models on graph datasets include social networks, knowledge bases, biology, and chemistry.
Several examples are provided using Amazon SageMaker’s deep learning containers that are preconfigured with DGL. If you have special modules you want to use with DGL, you can also build your own container. The examples involve heterographs, which are graphs that have multiple types of nodes and edges, and draw on a variety of applications across disparate scientific fields, such as bioinformatics and social network analysis. DGL provides a wide array of graph neural network implementations for different types models. Some of the highlights include:

- Graph convolutional network (GCN)
- Relational graph convolutional network (R-GCN)
- Graph attention network (GAT)
- Deep generative models of graphs (DGMG)
- Junction tree neural network (JTNN)

**Get Started**

DGL is available as a deep learning container in Amazon ECR. You can select deep learning containers when you write your estimator function in an Amazon SageMaker notebook. You can also craft your own custom container with DGL by following the Bring Your Own Container guide. The easiest way to get started with a deep graph network uses one of the DGL containers in Amazon ECR.

**Note**

Backend framework support is limited to PyTorch and MXNet.

**Setup**
If you are using Amazon SageMaker Studio, you need to clone the examples repository first. If you are using a notebook instance, you can find the examples by choosing the SageMaker icon at bottom of the left toolbar.

**To clone the Amazon SageMaker SDK and notebook examples repository**

1. From the **JupyterLab** view in Amazon SageMaker, go to the **File Browser** at the top of the left toolbar. From the **File Browser panel**, you can see a new navigation at the top of the panel.
2. Choose the icon on the far right to clone a Git repository.
3. Add the repository URL: https://github.com/awslabs/amazon-sagemaker-examples.git
4. Browse the newly added folder and its contents. The DGL examples are stored in the **sagemaker-python-sdk** folder.

**Run a Graph Network Training Example**

**To train a deep graph network**

1. From the **JupyterLab** view in Amazon SageMaker, browse the **example notebooks** and look for DGL folders. Several files may be included to support an example. Examine the README for any prerequisites.
2. Run the .ipynb notebook example.
3. Find the estimator function, and note the line where it is using an Amazon ECR container for DGL and a specific instance type. You may want to update this to use a container in your preferred Region.
4. Run the function to launch the instance and use the DGL container for training a graph network. Charges are incurred for launching this instance. The instance self-terminates when the training is complete.

**Examples**

An example of knowledge graph embedding (KGE) is provided. It uses the Freebase dataset, a knowledge base of general facts. An example use case would be to graph the relationships of persons and predict their nationality.

An example implementation of a graph convolutional network (GCN) shows how you can train a graph network to predict toxicity. A physiology dataset, Tox21, provides toxicity measurements for how substances affect biological responses.

Another GCN example shows you how to train a graph network on a scientific publications bibliography dataset, known as Cora. You can use it to find relationships between authors, topics, and conferences.

The last example is a recommender system for movie reviews. It uses a graph convolutional matrix completion (GCMC) network trained on the MovieLens datasets. These datasets consist of movie titles, genres, and ratings by users.

**Use a Deep Learning Container with DGL**

The following examples use preconfigured deep learning containers. These are the easiest to try since they work out of the box on Amazon SageMaker.

- Semi-supervised classification of a knowledge base using a GCN
- Learning embeddings of large-scale knowledge graphs using a dataset of scientific publications
Bring Your Own Container with DGL

The following examples enable you to bring your own container (BYOC). Read the BYOC guide and familiarize yourself with that process before trying these. Configuration is required.

- Molecular property prediction of toxicity using a GCN
- Recommender system for movies using a GCMC implementation

Extend a Prebuilt Container

If a prebuilt SageMaker container doesn't fulfill all of your requirements, you can extend the existing image to accommodate your needs. Even if there is direct support for your environment or framework, you may want to add additional functionality or configure your container environment differently. By extending a prebuilt image, you can leverage the included deep learning libraries and settings without having to create an image from scratch. You can extend the container to add libraries, modify settings, and install additional dependencies.

The following tutorial shows how to extend a prebuilt SageMaker image and publish it to Amazon ECR.

Topics
- Requirements to Extend a Prebuilt Container (p. 1363)
- Extend SageMaker Containers to Run a Python Script (p. 1363)

Requirements to Extend a Prebuilt Container

To extend a pre-built SageMaker image, you need to set the following environment variables within your Dockerfile. For more information on environment variables with SageMaker containers, see the SageMaker Training Toolkit GitHub repo.

- `SAGEMAKER_SUBMIT_DIRECTORY`: The directory within the container in which the Python script for training is located.
- `SAGEMAKER_PROGRAM`: The Python script that should be invoked and used as the entry point for training.

You can also install additional libraries by including the following in your Dockerfile:

```
RUN pip install <library>
```

The following tutorial shows how to use these environment variables.

Extend SageMaker Containers to Run a Python Script

In this tutorial, you learn how to extend the SageMaker PyTorch container with a Python file that uses the CIFAR-10 dataset. By extending the SageMaker PyTorch container, you utilize the existing training solution made to work with SageMaker. This tutorial extends a training image, but the same steps can be taken to extend an inference image. For a full list of the available images, see Available Deep Learning Containers Images.

To run your own training model using the SageMaker containers, build a Docker container through a SageMaker Notebook instance.

Step 1: Create an SageMaker Notebook Instance

1. Open the SageMaker console.
2. In the left navigation pane, choose Notebook, choose Notebook instances, and then choose Create notebook instance.

3. On the Create notebook instance page, provide the following information:
   a. For Notebook instance name, enter RunScriptNotebookInstance.
   b. For Notebook instance type, choose ml.t2.medium.
   c. In the Permissions and encryption section, do the following:
      i. For IAM role, choose Create a new role.
      ii. On the Create an IAM role page, choose Specific S3 buckets, specify an Amazon S3 bucket named sagemaker-run-script, and then choose Create role.
      SageMaker creates an IAM role named AmazonSageMaker-ExecutionRole-YYYYMMDDTHHmmSS, such as AmazonSageMaker-ExecutionRole-20190429T110788. Note that the execution role naming convention uses the date and time when the role was created, separated by a T.
   d. For Root Access, choose Enable.
   e. Choose Create notebook instance.

4. On the Notebook instances page, the Status is Pending. It can take a few minutes for Amazon SageMaker to launch a machine learning compute instance—in this case, it launches a notebook instance—and attach an ML storage volume to it. The notebook instance has a preconfigured Jupyter notebook server and a set of Anaconda libraries. For more information, see CreateNotebookInstance.

5. In the Permissions and encryption section, copy the IAM role ARN number, and paste it into a notepad file to save it temporarily. You use this IAM role ARN number later to configure a local training estimator in the notebook instance. The IAM role ARN number looks like the following: 'arn:aws:iam::111122223333:role/service-role/AmazonSageMaker-ExecutionRole-20190429T110788'

6. After the status of the notebook instance changes to InService, choose Open JupyterLab.

Step 2: Create and Upload the Dockerfile and Python Training Scripts

1. After JupyterLab opens, create a new folder in the home directory of your JupyterLab. In the upper-left corner, choose the New Folder icon, and then enter the folder name docker_test_folder.

2. Create a Dockerfile text file in the docker_test_folder directory.
   a. Choose the New Launcher icon (+) in the upper-left corner.
   b. In the right pane under the Other section, choose Text File.
   c. Paste the following Dockerfile sample code into your text file.

```
# SageMaker PyTorch image
FROM 763104351884.dkr.ecr.us-east-1.amazonaws.com/pytorch-training:1.5.1-cpu-py36-ubuntu16.04
ENV PATH="/opt/ml/code:${PATH}"

# this environment variable is used by the SageMaker PyTorch container to determine our user code directory.
ENV SAGEMAKER_SUBMIT_DIRECTORY /opt/ml/code

# /opt/ml and all subdirectories are utilized by SageMaker, use the /code subdirectory to store your user code.
COPY cifar10.py /opt/ml/code/cifar10.py

# Defines cifar10.py as script entrypoint
```
ENV SAGEMAKER_PROGRAM cifar10.py

The Dockerfile script performs the following tasks:

• FROM 763104351884.dkr.ecr.us-east-1.amazonaws.com/pytorch-training:1.5.1-cpu-py36-ubuntu16.04 – Downloads the SageMaker PyTorch base image. You can replace this with any SageMaker base image you want to bring to build containers.

• ENV SAGEMAKER_SUBMIT_DIRECTORY /opt/ml/code – Sets /opt/ml/code as the training script directory.

• COPY cifar10.py /opt/ml/code/cifar10.py –Copies the script to the location inside the container that is expected by SageMaker. The script must be located in this folder.

• ENV SAGEMAKER_PROGRAM cifar10.py – Sets your cifar10.py training script as the entrypoint script.

d. On the left directory navigation pane, the text file name might automatically be named untitled.txt. To rename the file, right-click the file, choose Rename, rename the file as Dockerfile without the .txt extension, and then press Ctrl+s or Command+s to save the file.

3. Create or upload a training script cifar10.py in the docker_test_folder. You can use the following example script for this exercise.

```python
import ast
import argparse
import logging
import os
import torch
import torch.distributed as dist
import torch.nn as nn
import torch.nn.parallel
import torch.optim
import torch.utils.data
import torch.utils.data.distributed
import torchvision
import torchvision.models
import torchvision.transforms as transforms
import torch.nn.functional as F

logger = logging.getLogger(__name__)
logger.setLevel(logging.DEBUG)

classes = ('plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck')

# https://github.com/pytorch/tutorials/blob/master/beginner_source/blitz/cifar10_tutorial.py#L118
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
```

1365
x = self.pool(F.relu(self.conv2(x)))
x = x.view(-1, 16 * 5 * 5)
x = F.relu(self.fc1(x))
x = F.relu(self.fc2(x))
x = self.fc3(x)
return x

def _train(args):
    is_distributed = len(args.hosts) > 1 and args.dist_backend is not None
    logger.debug("Distributed training - {}".format(is_distributed))

    if is_distributed:
        # Initialize the distributed environment.
        world_size = len(args.hosts)
        os.environ["WORLD_SIZE"] = str(world_size)
        host_rank = args.hosts.index(args.current_host)
        dist.init_process_group(backend=args.dist_backend, rank=host_rank,
            world_size=world_size)
        logger.info(
            'Initialized the distributed environment: \'{\}\' backend on {} nodes.
            '.format(
                args.dist_backend,
                dist.get_world_size()) + 'Current host rank is {}. Using cuda: {}.
                Number of gpus: {}').format(
                dist.get_rank(), torch.cuda.is_available(), args.num_gpus)

        device = 'cuda' if torch.cuda.is_available() else 'cpu'
        logger.info("Device Type: {}".format(device))

    logger.info("Loading Cifar10 dataset")
    transform = transforms.Compose(
        [transforms.ToTensor(),
         transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

    trainset = torchvision.datasets.CIFAR10(root=args.data_dir, train=True,
        download=False, transform=transform)
    train_loader = torch.utils.data.DataLoader(trainset, batch_size=args.batch_size,
        shuffle=True, num_workers=args.workers)

    testset = torchvision.datasets.CIFAR10(root=args.data_dir, train=False,
        download=False, transform=transform)
    test_loader = torch.utils.data.DataLoader(testset, batch_size=args.batch_size,
        shuffle=False, num_workers=args.workers)

    logger.info("Model loaded")
    model = Net()

    if torch.cuda.device_count() > 1:
        logger.info("Gpu count: {}".format(torch.cuda.device_count()))
        model = nn.DataParallel(model)

    model = model.to(device)

    criterion = nn.CrossEntropyLoss().to(device)
    optimizer = torch.optim.SGD(model.parameters(), lr=args.lr, momentum=args.momentum)

    for epoch in range(0, args.epochs):
        running_loss = 0.0
        for i, data in enumerate(train_loader):
            # get the inputs
            inputs, labels = data
            inputs, labels = inputs.to(device), labels.to(device)

            # zero the parameter gradients
            optimizer.zero_grad()
# forward + backward + optimize
outputs = model(inputs)
loss = criterion(outputs, labels)
optimizer.backward()

# print statistics
running_loss += loss.item()
if i % 2000 == 1999:  # print every 2000 mini-batches
    print('[%d, %5d] loss: %.3f' % (epoch + 1, i + 1, running_loss / 2000))
running_loss = 0.0
print('Finished Training')
return _save_model(model, args.model_dir)

def _save_model(model, model_dir):
    logger.info("Saving the model.")
    path = os.path.join(model_dir, 'model.pth')
    # recommended way from http://pytorch.org/docs/master/notes/serialization.html
    torch.save(model.cpu().state_dict(), path)

def model_fn(model_dir):
    logger.info('model_fn')
    device = "cuda" if torch.cuda.is_available() else "cpu"
    model = Net()
    if torch.cuda.device_count() > 1:
        logger.info("Gpu count: {}").format(torch.cuda.device_count()))
        model = nn.DataParallel(model)
    with open(os.path.join(model_dir, 'model.pth'), 'rb') as f:
        model = torch.load(f)
    return model.to(device)

if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('--workers', type=int, default=2, metavar='W',
                        help='number of data loading workers (default: 2)')
    parser.add_argument('--epochs', type=int, default=2, metavar='E',
                        help='number of total epochs to run (default: 2)')
    parser.add_argument('--batch-size', type=int, default=4, metavar='BS',
                        help='batch size (default: 4)')
    parser.add_argument('--lr', type=float, default=0.001, metavar='LR',
                        help='initial learning rate (default: 0.001)')
    parser.add_argument('--momentum', type=float, default=0.9, metavar='M',
                        help='momentum (default: 0.9)')
    parser.add_argument('--dist-backend', type=str, default='gloo',
                        help='distribution backend (default: gloo)')

    # The parameters below retrieve their default values from SageMaker environment
    # variables, which are
    # instantiated by the SageMaker containers framework.
    # https://github.com/aws/sagemaker-containers#how-a-script-is-executed-inside-the-containers
    parser.add_argument('--hosts', type=str,
                        default=ast.literal_eval(os.environ['SM_HOSTS']))
    parser.add_argument('--current-host', type=str,
                        default=os.environ['SM_CURRENT_HOST']
                        parser.add_argument('--model-dir', type=str,
                        default=os.environ['SM_MODEL_DIR']
                        parser.add_argument('--data-dir', type=str,
                        default=os.environ['SM_CHANNEL_TRAINING']
                        parser.add_argument('--num-gpus', type=int, default=os.environ['SM_NUM_GPUS'])
Step 3: Build the Container

1. In the JupyterLab home directory, open a Jupyter notebook. To open a new notebook, choose the New Launch icon and then choose conda_pytorch_p36 in the Notebook section.
2. Run the following command in the first notebook cell to change to the docker_test_folder directory:

   ```bash
   % cd ~/SageMaker/docker_test_folder
   ```

   This returns your current directory as follows:

   ```bash
   ! pwd
   ```

   ```output
   output: /home/ec2-user/SageMaker/docker_test_folder
   ```

3. Log in to Docker to access the base container:

   ```bash
   ! aws ecr get-login-password --region us-east-1 | docker login --username AWS --password-stdin 763104351884.dkr.ecr.us-east-1.amazonaws.com
   ```

4. To build the Docker container, run the following Docker build command, including the space followed by a period at the end:

   ```bash
   ! docker build -t pytorch-extended-container-test .
   ```

   The Docker build command must be run from the Docker directory you created, in this case docker_test_folder.

   **Note**
   If you get the following error message that Docker cannot find the Dockerfile, make sure the Dockerfile has the correct name and has been saved to the directory.

   ```bash
   unable to prepare context: unable to evaluate symlinks in Dockerfile path:
   lstat /home/ec2-user/SageMaker/docker/Dockerfile: no such file or directory
   ```

   Remember that docker looks for a file specifically called Dockerfile without any extension within the current directory. If you named it something else, you can pass in the file name manually with the -f flag. For example, if you named your Dockerfile Dockerfile-text.txt, run the following command:

   ```bash
   ! docker build -t tf-custom-container-test -f Dockerfile-text .
   ```

Step 4: Test the Container

1. To test the container locally in the notebook instance, open a Jupyter notebook. Choose New Launcher and choose Notebook in conda_pytorch_p36 framework. The rest of the code snippets must run from the Jupyter notebook instance.
2. Download the CIFAR-10 dataset.

   ```python
   import torch
   ```
import torchvision
import torchvision.transforms as transforms

def _get_transform():
    return transforms.Compose(
        [transforms.ToTensor(),
         transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

def get_train_data_loader(data_dir='/tmp/pytorch/cifar-10-data'):
    transform = _get_transform()
    trainset = torchvision.datasets.CIFAR10(root=data_dir, train=True,
                                             download=True, transform=transform)
    return torch.utils.data.DataLoader(trainset, batch_size=4,
                                        shuffle=True, num_workers=2)

def get_test_data_loader(data_dir='/tmp/pytorch/cifar-10-data'):
    transform = _get_transform()
    testset = torchvision.datasets.CIFAR10(root=data_dir, train=False,
                                            download=True, transform=transform)
    return torch.utils.data.DataLoader(testset, batch_size=4,
                                         shuffle=False, num_workers=2)

trainloader = get_train_data_loader('/tmp/pytorch-example/cifar-10-data')
testloader = get_test_data_loader('/tmp/pytorch-example/cifar-10-data')

3. Set role to the role used to create your Jupyter notebook. This is used to configure your SageMaker Estimator.

from sagemaker import get_execution_role
role = get_execution_role()

4. Paste the following example script into the notebook code cell to configure a SageMaker Estimator using your extended container.

from sagemaker.estimator import Estimator

hyperparameters = {'epochs': 1}

estimator = Estimator(image_name='pytorch-extended-container-test',
                       role=role,
                       train_instance_count=1,
                       train_instance_type='local',
                       hyperparameters=hyperparameters)

estimator.fit('file:///tmp/pytorch-example/cifar-10-data')

5. Run the code cell. This test outputs the training environment configuration, the values used for the environmental variables, the source of the data, and the loss and accuracy obtained during training.

**Step 5: Push the Container to Amazon Elastic Container Registry (Amazon ECR)**

1. After you successfully run the local mode test, you can push the Docker container to Amazon ECR and use it to run training jobs.

Run the following command lines in a notebook cell.
# Specify an algorithm name

```python
algorithm_name=pytorch-extended-container-test
```

```bash
account=$(aws sts get-caller-identity --query Account --output text)
```

# Get the region defined in the current configuration (default to us-west-2 if none defined)

```bash
region=$(aws configure get region)
```

**fullname=${account}.dkr.ecr.${region}.amazonaws.com/${algorithm_name}:latest**

# If the repository doesn’t exist in ECR, create it.

```bash
aws ecr describe-repositories --repository-names "${algorithm_name}" > /dev/null 2>&1
```

```bash
if [ $? -ne 0 ]
then
aws ecr create-repository --repository-name "${algorithm_name}" > /dev/null
fi
```

# Build the docker image locally with the image name and then push it to ECR with the full name.

```bash
docker build -t ${algorithm_name} .
docker tag ${algorithm_name} ${fullname}
docker push ${fullname}
```

2. After you push the container, you can call the Amazon ECR image from anywhere in the SageMaker environment. Run the following code example in the next notebook cell.

If you want to use this training container with SageMaker Studio to use its visualization features, you can also run the following code in a Studio notebook cell to call the Amazon ECR image of your training container.

```python
import boto3
client = boto3.client('sts')
account = client.get_caller_identity()['Account']
my_session = boto3.session.Session()
region = my_session.region_name
algorithm_name = "pytorch-extended-container-test"
ecr_image = '{}.dkr.ecr.{}.amazonaws.com/{}/latest'.format(account, region, algorithm_name)
ecr_image
```

# This should return something like

# 12-digits-of-your-account.dkr.ecr.us-east-2.amazonaws.com/tf-2.2-test:latest

3. Use the ecr_image retrieved from the previous step to configure a SageMaker estimator object. The following code sample configures a SageMaker PyTorch estimator.

```python
import sagemaker
from sagemaker import get_execution_role
from sagemaker.estimator import Estimator
estimator = Estimator(image_name=ecr_image,
                      role=get_execution_role(),
                      base_job_name='pytorch-extended-container-test',
                      train_instance_count=1,
                      train_instance_type='ml.p2.xlarge')
```
Adapting Your Own Docker Container to Work with SageMaker

You can adapt an existing Docker image to work with SageMaker. You may need to use an existing, external Docker image with SageMaker when you have a container that satisfies feature or safety requirements that are not currently supported by a prebuilt SageMaker image. There are two toolkits that allow you to bring your own container and adapt it to work with SageMaker:

- SageMaker Training Toolkit
- SageMaker Inference Toolkit

The following topics show how to adapt your existing image using the SageMaker Training and Inference toolkits:

Topics
- Individual Framework Libraries (p. 1371)
- Using the SageMaker Training and Inference Toolkits (p. 1372)
- Adapting Your Own Training Container (p. 1373)
- Adapting Your Own Inference Container (p. 1379)

Individual Framework Libraries

In addition to the SageMaker Training Toolkit and SageMaker Inference Toolkit, SageMaker also provides toolkits specialized for TensorFlow, MXNet, PyTorch, and Chainer. The following table provides links to the GitHub repositories that contain the source code for each framework and their respective serving toolkits. The instructions linked are for using the Python SDK to run training algorithms and host models.
on SageMaker. The functionality for these individual libraries is included in the SageMaker Training Toolkit and SageMaker Inference Toolkit.

<table>
<thead>
<tr>
<th>Framework</th>
<th>Toolkit Source Code</th>
</tr>
</thead>
</table>
| TensorFlow | SageMaker TensorFlow Training  
SageMaker TensorFlow Serving |
| MXNet | SageMaker MXNet Training  
SageMaker MXNet Inference |
| PyTorch | SageMaker PyTorch Training  
SageMaker PyTorch Inference |
| Chainer | SageMaker Chainer SageMaker Containers |

### Using the SageMaker Training and Inference Toolkits

The SageMaker Training and SageMaker Inference toolkits implement the functionality that you need to adapt your containers to run scripts, train algorithms, and deploy models on SageMaker. When installed, the library defines the following for users:

- The locations for storing code and other resources.
- The entry point that contains the code to run when the container is started. Your Dockerfile must copy the code that needs to be run into the location expected by a container that is compatible with SageMaker.
- Other information that a container needs to manage deployments for training and inference.

### SageMaker Toolkits Containers Structure

When SageMaker trains a model, it creates the following file folder structure in the container’s `/opt/ml` directory.

```
/opt/ml
  ## input
  #  ## config
  #  #  ### hyperparameters.json  
  #  #  ### resourceConfig.json  
  #  ### data
  #    ### <channel_name>
  #      ### <input data>
  ### model  
  #  ### code
  #  ### output
  #  ### failure
```

When you run a model training job, the SageMaker container uses the `/opt/ml/input/` directory, which contains the JSON files that configure the hyperparameters for the algorithm and the network layout used for distributed training. The `/opt/ml/input/` directory also contains files that specify the channels through which SageMaker accesses the data, which is stored in Amazon Simple Storage Service.
Adapting Your Own Training Container

To run your own training model, build a Docker container using the Amazon SageMaker Training Toolkit through an Amazon SageMaker notebook instance.

**Step 1: Create a SageMaker notebook instance**

1. Open the Amazon SageMaker console at https://console.aws.amazon.com/sagemaker/.
2. In the left navigation pane, choose **Notebook**, choose **Notebook instances**, and then choose **Create notebook instance**.
3. On the **Create notebook instance** page, provide the following information:
   a. For **Notebook instance name**, enter **RunScriptNotebookInstance**.
   b. For **Notebook Instance type**, choose **ml.t2.medium**.
   c. In the **Permissions and encryption** section, do the following:
Adapting Your Own Training Container

i. For IAM role, choose **Create a new role**.

ii. On the **Create an IAM role** page, choose **Specific S3 buckets**, specify an Amazon S3 bucket named `sagemaker-run-script`, and then choose **Create role**.

SageMaker creates an IAM role named `AmazonSageMaker-ExecutionRole-YYYYMMDDTHHmmSS`. For example, `AmazonSageMaker-ExecutionRole-20190429T110788`. Note that the execution role naming convention uses the date and time at which the role was created, separated by a `T`.

d. For **Root Access**, choose Enable.

e. Choose **Create notebook instance**.

4. On the **Notebook instances** page, the **Status** is **Pending**. It can take a few minutes for Amazon SageMaker to launch a machine learning compute instance—in this case, it launches a notebook instance—and attach an ML storage volume to it. The notebook instance has a preconfigured Jupyter notebook server and a set of Anaconda libraries. For more information, see CreateNotebookInstance.

5. In the **Permissions and encryption** section, copy the IAM role ARN number, and paste it into a notepad file to save it temporarily. You use this IAM role ARN number later to configure a local training estimator in the notebook instance. The IAM role ARN number looks like the following: `arn:aws:iam::111122223333:role/service-role/AmazonSageMaker-ExecutionRole-20190429T110788`

6. After the status of the notebook instance changes to **InService**, choose **Open JupyterLab**.

**Step 2: Create and upload the Dockerfile and Python training scripts**

1. After JupyterLab opens, create a new folder in the home directory of your JupyterLab. In the upper-left corner, choose the **New Folder** icon, and then enter the folder name `docker_test_folder`.

2. Create a **Dockerfile** text file in the `docker_test_folder` directory.

   a. Choose the **New Launcher** icon (+) in the upper-left corner.

   b. In the right pane under the **Other** section, choose **Text File**.

   c. Paste the following Dockerfile sample code into your text file.

```
FROM tensorflow/tensorflow:2.2.0rc2-gpu-py3-jupyter

# Install sagemaker-training toolkit to enable SageMaker Python SDK
RUN pip3 install sagemaker-training

# Copies the training code inside the container
COPY train.py /opt/ml/code/train.py

# Defines train.py as script entrypoint
ENV SAGEMAKER_PROGRAM train.py
```

The Dockerfile script performs the following tasks:

- **FROM tensorflow/tensorflow:2.2.0rc2-gpu-py3-jupyter** — Downloads the TensorFlow Docker base image. You can replace this with any Docker base image you want to bring to build containers, as well as with AWS pre-built container base images.

- **RUN pip install sagemaker-training** — Installs SageMaker Training Toolkit that contains the common functionality necessary to create a container compatible with SageMaker.
Adapting Your Own Training Container

- COPY train.py /opt/ml/code/train.py – Copies the script to the location inside the container that is expected by SageMaker. The script must be located in this folder.
- ENV SAGEMaker_PROGRAM train.py – Takes your training script train.py as the entrypoint script copied in the /opt/ml/code folder of the container. This is the only environmental variable that you must specify when you build your own container.

d. On the left directory navigation pane, the text file name might automatically be named untitled.txt. To rename the file, right-click the file, choose Rename, rename the file as Dockerfile without the .txt extension, and then press Ctrl+s or Command+s to save the file.

3. Create or upload a training script train.py in the docker_test_folder. You can use the following example script for this exercise.

```python
import tensorflow as tf

mnist = tf.keras.datasets.mnist

(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0

model = tf.keras.models.Sequential([tf.keras.layers.Flatten(input_shape=(28, 28)),
                                    tf.keras.layers.Dense(128, activation='relu'),
                                    tf.keras.layers.Dropout(0.2),
                                    tf.keras.layers.Dense(10, activation='softmax')])

model.compile(optimizer='adam',
               loss='sparse_categorical_crossentropy',
               metrics=['accuracy'])

model.fit(x_train, y_train, epochs=1)
model.evaluate(x_test, y_test)
```

Step 3: Build the container

1. In the JupyterLab home directory, open a Jupyter notebook. To open a new notebook, choose the New Launch icon and then choose conda_tensorflow2_p36 in the Notebook section.
2. Run the following command in the first notebook cell to change to the docker_test_folder directory:

```bash
% cd ~/SageMaker/docker_test_folder
```

This returns your current directory as follows:

```bash
! pwd
```

output: /home/ec2-user/SageMaker/docker_test_folder

3. To build the Docker container, run the following Docker build command, including the space followed by a period at the end:

```bash
! docker build -t tf-custom-container-test .
```
The Docker build command must be run from the Docker directory you created, in this case docker_test_folder.

**Note**
If you get the following error message that Docker cannot find the Dockerfile, make sure the Dockerfile has the correct name and has been saved to the directory.

```bash
unable to prepare context: unable to evaluate symlinks in Dockerfile path:
lstat /home/ec2-user/SageMaker/docker/Dockerfile: no such file or directory
```

Remember that docker looks for a file specifically called Dockerfile without any extension within the current directory. If you named it something else, you can pass in the file name manually with the `-f` flag. For example, if you named your Dockerfile as Dockerfile-text.txt, run the following command:

```bash
! docker build -t tf-custom-container-test -f Dockerfile-text.txt .
```

### Step 4: Test the container

1. To test the container locally in the notebook instance, open a Jupyter notebook. Choose **New Launcher** and choose **Notebook** in **conda_tensorflow_p36** framework.
2. Paste the following example script into the notebook code cell to configure a SageMaker Estimator.

**SageMaker Python SDK v1**

```python
from sagemaker.estimator import Estimator

estimator = Estimator(image_name='tf-custom-container-test',
    role='arn:aws:iam::111122223333:role/role-name',
    train_instance_count=1,
    train_instance_type='local')

estimator.fit()
```

**SageMaker Python SDK v2**

```python
from sagemaker.estimator import Estimator

estimator = Estimator(image_uri='tf-custom-container-test',
    role='arn:aws:iam::111122223333:role/role-name',
    instance_count=1,
    instance_type='local')

estimator.fit()
```

3. Replace the 'Put_Your_ARN_Here' value with the **IAM role ARN number** you copied to a notepad file when you configured the notebook instance. The ARN should look like the following: 'arn:aws:iam::111122223333:role/service-role/AmazonSageMaker-ExecutionRole-20190429T110788'.
4. Run the code cell. This test outputs the training environment configuration, the values used for the environmental variables, the source of the data, and the loss and accuracy obtained during training.
Step 5: Push the container to Amazon Elastic Container Registry (Amazon ECR)

1. After you successfully run the local mode test, you can push the Docker container to Amazon ECR and use it to run training jobs.

Run the following command lines in a notebook cell.

```
%sh

# Specify an algorithm name
algorithm_name=tf-custom-container-test

account=$(aws sts get-caller-identity --query Account --output text)

# Get the region defined in the current configuration (default to us-west-2 if none defined)
region=$(aws configure get region)
region=${region:-us-west-2}

fullname="${account}.dkr.ecr.${region}.amazonaws.com/${algorithm_name}:latest"

# If the repository doesn't exist in ECR, create it.
aws ecr describe-repositories --repository-names "${algorithm_name}" > /dev/null 2>&1
if [ $? -ne 0 ]
then
    aws ecr create-repository --repository-name "${algorithm_name}" > /dev/null
fi

# Get the login command from ECR and execute it directly
$(aws ecr get-login --region ${region} --no-include-email)

# Build the docker image locally with the image name and then push it to ECR with the full name.
docker build -t ${algorithm_name} .
docker tag ${algorithm_name} ${fullname}
docker push ${fullname}
```

Note
This bash shell script may raise a permission issue similar to the following error message:

"denied: User: [ARN] is not authorized to perform: ecr:InitiateLayerUpload on resource: arn:aws:ecr:us-east-1:[id]:repository/tf-custom-container-test"

If this error occurs, you need to attach the AmazonEC2ContainerRegistryFullAccess policy to your IAM role. Go to the IAM console, choose Roles from the left navigation pane, look up the IAM role you used for the Notebook instance. Under the Permission tab, choose the Attach policies button, and search the AmazonEC2ContainerRegistryFullAccess policy. Mark the check box of the policy, and choose Attach policy to finish.

2. After you push the container, you can call the Amazon ECR image from anywhere in the SageMaker environment. Run the following code example in the next notebook cell.
If you want to use this training container with SageMaker Studio to use its visualization features, you can also run the following code in a Studio notebook cell to call the Amazon ECR image of your training container.

```python
import boto3
account_id = boto3.client('sts').get_caller_identity().get('Account')
ecr_repository = 'sagemaker-byoc-test'
tag = ':latest'
region = boto3.session.Session().region_name
uri_suffix = 'amazonaws.com'
if region in ['cn-north-1', 'cn-northwest-1']:
    uri_suffix = 'amazonaws.com.cn'
byoc_image_uri = '{}.dkr.ecr.{}.{}/{}'.format(account_id, region, uri_suffix, ecr_repository + tag)
byoc_image_uri
# This should return something like
# 111122223333.dkr.ecr.us-east-2.amazonaws.com/sagemaker-byoc-test:latest
```

3. Use the `ecr_image` retrieved from the previous step to configure a SageMaker estimator object. The following code sample configures a SageMaker estimator with the `byoc_image_uri` and initiates a training job on an Amazon EC2 instance.

**SageMaker Python SDK v1**

```python
import sagemaker
from sagemaker import get_execution_role
from sagemaker.estimator import Estimator

estimator = Estimator(image_name=byoc_image_uri,
    role=get_execution_role(),
    base_job_name='tf-custom-container-test-job',
    train_instance_count=1,
    train_instance_type='ml.p2.xlarge')

# start training
estimator.fit()

# deploy the trained model
predictor = estimator.deploy(1, instance_type)
```

**SageMaker Python SDK v2**

```python
import sagemaker
from sagemaker import get_execution_role
from sagemaker.estimator import Estimator

estimator = Estimator(image_uri=byoc_image_uri,
    role=get_execution_role(),
    base_job_name='tf-custom-container-test-job',
    instance_count=1,
    instance_type='ml.p2.xlarge')

# start training
estimator.fit()

# deploy the trained model
```
predictor = estimator.deploy(1, instance_type)

For a full example that shows how to test a custom container locally and push it to an Amazon ECR image, see the Building Your Own TensorFlow Container example notebook.

## Step 6: Clean up resources

### To clean up resources when done with the get started example

1. Open the SageMaker console, choose the notebook instance RunScriptNotebookInstance, choose Actions, and choose Stop. It can take a few minutes for the instance to stop.

2. After the instance Status changes to Stopped, choose Actions, choose Delete, and then choose Delete in the dialog box. It can take a few minutes for the instance to be deleted. The notebook instance disappears from the table when it has been deleted.

3. Open the Amazon S3 console and delete the bucket that you created for storing model artifacts and the training dataset.

4. Open the IAM console and delete the IAM role. If you created permission policies, you can delete them, too.

   **Note**
   The Docker container shuts down automatically after it has run. You don’t need to delete it.

## Adapting Your Own Inference Container

If none of the Amazon SageMaker prebuilt inference containers suffice for your situation, and you want to use your own Docker container, use the SageMaker Inference Toolkit to adapt your container to work with SageMaker hosting. To adapt your container to work with SageMaker hosting, create the inference code in one or more Python script files and a Dockerfile that imports the inference toolkit.

The inference code includes an inference handler, a handler service, and an entrypoint. In this example, they are stored as three separate Python files. All three of these Python files must be in the same directory as your Dockerfile.

For an example Jupyter notebook that shows a complete example of extending a container by using the SageMaker inference toolkit, see Amazon SageMaker Multi-Model Endpoints using your own algorithm container.

## Step 1: Create an Inference Handler

The SageMaker inference toolkit is built on the multi-model server (MMS). MMS expects a Python script that implements functions to load the model, pre-process input data, get predictions from the model, and process the output data in a model handler.

### The model_fn Function

The model_fn function is responsible for loading your model. It takes a model_dir argument that specifies where the model is stored. How you load your model depends on the framework you are using. There is no default implementation for the model_fn function. You must implement it yourself. The following simple example shows an implementation of a model_fn function that loads a PyTorch model:

```python
def model_fn(self, model_dir):
    import torch

    logger.info('model_fn')
    device = "cuda" if torch.cuda.is_available() else "cpu"
```
with open(os.path.join(model_dir, 'model.pth'), 'rb') as f:
    model = torch.jit.load(f)
    return model.to(device)

The \texttt{input\_fn} Function

The \texttt{input\_fn} function is responsible for deserializing your input data so that it can be passed to your model. It takes input data and content type as parameters, and returns deserialized data. The SageMaker inference toolkit provides a default implementation that deserializes the following content types:

- JSON
- CSV
- Numpy array
- NPZ

If your model requires a different content type, or you want to preprocess your input data before sending it to the model, you must implement the \texttt{input\_fn} function. The following example shows a simple implementation of the \texttt{input\_fn} function.

```python
from sagemaker_inference import content_types, decoder
def input_fn(self, input_data, content_type):
    """A default input\_fn that can handle JSON, CSV and NPZ formats.

    Args:
    input_data: the request payload serialized in the content_type format
    content_type: the request content_type

    Returns: input_data deserialized into torch.FloatTensor or torch.cuda.FloatTensor depending if cuda is available.
    ""
    return decoder.decode(input_data, content_type)
```

The \texttt{predict\_fn} Function

The \texttt{predict\_fn} function is responsible for getting predictions from the model. It takes the model and the data returned from \texttt{input\_fn} as parameters, and returns the prediction. There is no default implementation for the \texttt{predict\_fn}. You must implement it yourself. The following is a simple implementation of the \texttt{predict\_fn} function for a PyTorch model.

```python
def predict_fn(self, data, model):
    """A default predict\_fn for PyTorch. Calls a model on data deserialized in input\_fn.

    Runs prediction on GPU if cuda is available.

    Args:
    data: input data (torch.Tensor) for prediction deserialized by input\_fn
    model: PyTorch model loaded in memory by model\_fn

    Returns: a prediction
    ""
    return model(input_data)
```

The \texttt{output\_fn} Function

The \texttt{output\_fn} function is responsible for serializing the data that the \texttt{predict\_fn} function returns as a prediction. The SageMaker inference toolkit implements a default \texttt{output\_fn} function that serializes Numpy arrays, JSON, and CSV. If your model outputs any other content type, or you want to
perform other post-processing of your data before sending it to the user, you must implement your own output_fn function. The following shows a simple output_fn function for a PyTorch model.

```python
from sagemaker_inference import encoder
def output_fn(self, prediction, accept):
    """A default output_fn for PyTorch. Serializes predictions from predict_fn to JSON, CSV or NPY format.

    Args:
        prediction: a prediction result from predict_fn
        accept: type which the output data needs to be serialized

    Returns: output data serialized
    """
    return encoder.encode(prediction, accept)
```

### Step 2: Implement a Handler Service

The handler service is executed by the model server. The handler service implements initialize and handle methods. The initialize method is invoked when the model server starts, and the handle method is invoked for all incoming inference requests to the model server. For more information, see Custom Service in the Multi-model server documentation. The following is an example of a handler service for a PyTorch model server.

```python
from sagemaker_inference.default_handler_service import DefaultHandlerService
from sagemaker_inference.transformer import Transformer
from sagemaker_pytorch_serving_container.default_inference_handler import DefaultPytorchInferenceHandler
class HandlerService(DefaultHandlerService):
    """Handler service that is executed by the model server. Determines specific default inference handlers to use based on model being used.
    This class extends DefaultHandlerService, which define the following:
    - The `handle` method is invoked for all incoming inference requests to the model server.
    - The `initialize` method is invoked at model server start up.
    Based on: https://github.com/awslabs/mxnet-model-server/blob/master/docs/custom_service.md
    """
    def __init__(self):
        transformer = Transformer(default_inference_handler=DefaultPytorchInferenceHandler())
        super(HandlerService, self).__init__(transformer=transformer)
```

### Step 3: Implement an Entrypoint

The entrypoint starts the model server by invoking the handler service. You specify the location of the entrypoint in your Dockerfile. The following is an example of an entrypoint.

```python
from sagemaker_inference import model_server
model_server.start_model_server(handler_service=HANDLER_SERVICE)
```

### Step 4: Write a Dockerfile

In your Dockerfile, copy the model handler from step 2 and specify the Python file from the previous step as the entrypoint in your Dockerfile. The following is an example of the lines you can add to your Dockerfile.
Dockerfile to copy the model handler and specify the entrypoint. For a full example of a Dockerfile for an inference container, see Dockerfile.

```
# Copy the default custom service file to handle incoming data and inference requests
COPY model_handler.py /home/model-server/model_handler.py

# Define an entrypoint script for the docker image
ENTRYPOINT ["python", "/usr/local/bin/entrypoint.py"]
```

Step 5: Build and Register Your Container

Now you can build your container and register it in Amazon Elastic Container Registry (Amazon ECR). The following shell script from the sample notebook builds the container and uploads it to an Amazon ECR repository in your AWS account.

### Note
SageMaker hosting supports using inference containers that are stored in repositories other than Amazon ECR. For information, see Use a Private Docker Registry for Real-Time Inference Containers (p. 1390).

The following shell script shows how to build and register a container.

```
#!/bin/bash

# The name of our algorithm
algorithm_name=demo-sagemaker-multimodel

cd container

account=$(aws sts get-caller-identity --query Account --output text)

# Get the region defined in the current configuration (default to us-west-2 if none defined)
region=$(aws configure get region)
region=${region:-us-west-2}

fullname="${account}.dkr.ecr.${region}.amazonaws.com/${algorithm_name}:latest"

# If the repository doesn't exist in ECR, create it.
aws ecr describe-repositories --repository-names "${algorithm_name}" > /dev/null 2>&1
if [ $? -ne 0 ]
then
    aws ecr create-repository --repository-name "${algorithm_name}" > /dev/null
fi

# Get the login command from ECR and execute it directly
$(aws ecr get-login --region ${region} --no-include-email)

# Build the docker image locally with the image name and then push it to ECR with the full name.
docker build -q -t ${algorithm_name} .
docker tag ${algorithm_name} ${fullname}
docker push ${fullname}
```

You can now use this container to deploy endpoints in SageMaker. For an example of how to deploy an endpoint in SageMaker, see Step 6.1: Deploy the Model to SageMaker Hosting Services (p. 68).
Create a container with your own algorithms and models

If none of the existing SageMaker containers meet your needs and you don’t have an existing container of your own, you may need to create a new Docker container. The following sections show how to create Docker containers with your training and inference algorithms for use with SageMaker.

Topics

• Use Your Own Training Algorithms (p. 1383)
• Use Your Own Inference Code (p. 1387)

Use Your Own Training Algorithms

This section explains how Amazon SageMaker interacts with a Docker container that runs your custom training algorithm. Use this information to write training code and create a Docker image for your training algorithms.

Topics

• How Amazon SageMaker Runs Your Training Image (p. 1383)
• How Amazon SageMaker Provides Training Information (p. 1384)
• How Amazon SageMaker Signals Algorithm Success and Failure (p. 1386)
• How Amazon SageMaker Processes Training Output (p. 1387)

How Amazon SageMaker Runs Your Training Image

To configure a Docker container to run as an executable, use an ENTRYPOINT instruction in a Dockerfile. Note the following:

• For model training, Amazon SageMaker runs the container as follows:

  docker run image train

  SageMaker overrides any default CMD statement in a container by specifying the train argument after the image name. The train argument also overrides arguments that you provide using CMD in the Dockerfile.

• In your dockerfile, use the exec form of the ENTRYPOINT instruction:

  ENTRYPOINT ["executable", "param1", "param2", ...]

  For example:

  ENTRYPOINT ["python", "k-means-algorithm.py"]

  The exec form of the ENTRYPOINT instruction starts the executable directly, not as a child of /bin/sh. This enables it to receive signals like SIGTERM and SIGKILL from SageMaker APIs. Note the following:
Amazon SageMaker Developer Guide
Use Your Own Training Algorithms

- The **CreateTrainingJob** API has a stopping condition that directs SageMaker to stop model training after a specific time.

- The **StopTrainingJob** API issues the equivalent of the `docker stop`, with a 2-minute timeout command to gracefully stop the specified container:

  ```
docker stop -t 120
  ```

  The command attempts to stop the running container by sending a **SIGTERM** signal. After the 2-minute timeout, **SIGKILL** is sent and the containers are forcibly stopped. If the container handles the **SIGTERM** gracefully and exits within 120 seconds from receiving it, no **SIGKILL** is sent.

  **Note**
  If you want access to the intermediate model artifacts after SageMaker stops the training, add code to handle saving artifacts in your **SIGTERM** handler.

- If you plan to use GPU devices for model training, make sure that your containers are **nvidia-docker** compatible. Only the CUDA toolkit should be included on containers; don't bundle NVIDIA drivers with the image. For more information about **nvidia-docker**, see [NVIDIA/nvidia-docker](https://github.com/NVIDIA/nvidia-docker).

- You can't use the **tini** initializer as your entry point in SageMaker containers because it gets confused by the `train` and `serve` arguments.

- `/opt/ml` and all sub-directories are reserved by SageMaker training. When building your algorithm's Docker image, please ensure you don't place any data required by your algorithm under them as the data may no longer be visible during training.

### How Amazon SageMaker Provides Training Information

This section explains how SageMaker makes training information, such as training data, hyperparameters, and other configuration information, available to your Docker container.

When you send a **CreateTrainingJob** request to SageMaker to start model training, you specify the Amazon Elastic Container Registry path of the Docker image that contains the training algorithm. You also specify the Amazon Simple Storage Service (Amazon S3) location where training data is stored and algorithm-specific parameters. SageMaker makes this information available to the Docker container so that your training algorithm can use it. This section explains how we make this information available to your Docker container. For information about creating a training job, see **CreateTrainingJob**. For more information on the way that SageMaker containers organize information, see [Using the SageMaker Training and Inference Toolkits](p. 1372).

**Topics**

- Hyperparameters (p. 1384)
- Environment Variables (p. 1385)
- Input Data Configuration (p. 1385)
- Training Data (p. 1385)
- Distributed Training Configuration (p. 1386)

**Hyperparameters**

SageMaker makes the hyperparameters in a **CreateTrainingJob** request available in the Docker container in the `/opt/ml/input/config/hyperparameters.json` file.
Environment Variables

- **TRAINING_JOB_NAME**—The training job name stored in the `TrainingJobName` parameter in a `CreateTrainingJob` request.
- **TRAINING_JOB_ARN**—The Amazon Resource Name (ARN) of the training job returned as the `TrainingJobArn` response element for `CreateTrainingJob`.

Input Data Configuration

You specify data channel information in the `InputDataConfig` parameter in a `CreateTrainingJob` request. SageMaker makes this information available in the `/opt/ml/input/config/inputdataconfig.json` file in the Docker container.

For example, suppose that you specify three data channels (train, evaluation, and validation) in your request. SageMaker provides the following JSON:

```json
{
    "train" : {
        "ContentType":  "trainingContentType",
        "TrainingInputMode": "File",
        "S3DistributionType": "FullyReplicated",
        "RecordWrapperType": "None"},
    "evaluation" : {
        "ContentType":  "evalContentType",
        "TrainingInputMode": "File",
        "S3DistributionType": "FullyReplicated",
        "RecordWrapperType": "None"},
    "validation" : {
        "TrainingInputMode": "File",
        "S3DistributionType": "FullyReplicated",
        "RecordWrapperType": "None"}
}
```

**Note**
SageMaker provides only relevant information about each data channel (for example, the channel name and the content type) to the container, as shown. `S3DistributionType` will be set as `FullyReplicated` if specify EFS or FSxLustre as input data sources.

Training Data

The `TrainingInputMode` parameter in a `CreateTrainingJob` request specifies how to make data available for model training: in **FILE** mode or **PIPE** mode. Depending on the specified input mode, SageMaker does the following:

- **FILE mode**—SageMaker makes the data for the channel available in the `/opt/ml/input/data/channel_name` directory in the Docker container. For example, if you have three channels named training, validation, and testing, SageMaker makes three directories in the Docker container:
  - `/opt/ml/input/data/training`
  - `/opt/ml/input/data/validation`
  - `/opt/ml/input/data/testing`

  **Note**
  Channels that use file system data sources such as Amazon Elastic File System (Amazon EFS) and Amazon FSx must use FILE mode. Also to utilize an Amazon FSx file server, you must specify a path that begins with `/fsx`. If a file system is specified, the directory path provided in the channel is mounted at `/opt/ml/input/data/channel_name`.

- **PIPE mode**—SageMaker makes data for the channel available from the named pipe: `/opt/ml/input/data/channel_name_epoch_number`. For example, if you have three channels named training, validation, and testing, you will need to read from the following pipes:
• `/opt/ml/input/data/training_0`, `/opt/ml/input/data/training_1`, ...
• `/opt/ml/input/data/validation_0`, `/opt/ml/input/data/validation_1`, ...
• `/opt/ml/input/data/testing_0`, `/opt/ml/input/data/testing_1`, ...

Read the pipes sequentially. For example, if you have a channel called `training`, read the pipes in this sequence:

1. Open `/opt/ml/input/data/training_0` in read mode and read it to end-of-file (EOF) or, if you are done with the first epoch, close the pipe file early.
2. After closing the first pipe file, look for `/opt/ml/input/data/training_1` and read it until you have completed the second epoch, and so on.

If the file for a given epoch doesn't exist yet, your code may need to retry until the pipe is created. There is no sequencing restriction across channel types. That is, you can read multiple epochs for the `training` channel, for example, and only start reading the `validation` channel when you are ready. Or, you can read them simultaneously if your algorithm requires that.

### Distributed Training Configuration

If you're performing distributed training with multiple containers, SageMaker makes information about all containers available in the `/opt/ml/input/config/resourceconfig.json` file.

To enable inter-container communication, this JSON file contains information for all containers. SageMaker makes this file available for both FILE and PIPE mode algorithms. The file provides the following information:

- `current_host`—The name of the current container on the container network. For example, `algo-1`. Host values can change at any time. Don't write code with specific values for this variable.
- `hosts`—The list of names of all containers on the container network, sorted lexicographically. For example, `["algo-1", "algo-2", "algo-3"]` for a three-node cluster. Containers can use these names to address other containers on the container network. Host values can change at any time. Don't write code with specific values for these variables.
- `network_interface_name`—The name of the network interface that is exposed to your container. For example, containers running the Message Passing Interface (MPI) can use this information to set the network interface name.
- Do not use the information in `/etc/hostname` or `/etc/hosts` because it might be inaccurate.
- Hostname information may not be immediately available to the algorithm container. We recommend adding a retry policy on hostname resolution operations as nodes become available in the cluster.

The following is an example file on node 1 in a three-node cluster:

```json
{
  "current_host": "algo-1",
  "hosts": ["algo-1","algo-2","algo-3"],
  "network_interface_name": "eth1"
}
```

### How Amazon SageMaker Signals Algorithm Success and Failure

A training algorithm indicates whether it succeeded or failed using the exit code of its process.

A successful training execution should exit with an exit code of 0 and an unsuccessful training execution should exit with a non-zero exit code. These will be converted to `Completed` and `Failed` in the `TrainingJobStatus` returned by `DescribeTrainingJob`. This exit code convention is standard and...
is easily implemented in all languages. For example, in Python, you can use `sys.exit(1)` to signal a failure exit, and simply running to the end of the main routine will cause Python to exit with code 0.

In the case of failure, the algorithm can write a description of the failure to the failure file. See next section for details.

**How Amazon SageMaker Processes Training Output**

As your algorithm runs in a container, it generates output including the status of the training job and model and output artifacts. Your algorithm should write this information to the following files, which are located in the container's `/output` directory. Amazon SageMaker processes the information contained in this directory as follows:

- `/opt/ml/output/failure`—If training fails, after all algorithm output (for example, logging) completes, your algorithm should write the failure description to this file. In a `DescribeTrainingJob` response, SageMaker returns the first 1024 characters from this file as `FailureReason`.

- `/opt/ml/model`—Your algorithm should write all final model artifacts to this directory. SageMaker copies this data as a single object in compressed tar format to the S3 location that you specified in the `CreateTrainingJob` request. If multiple containers in a single training job write to this directory they should ensure no file/directory names clash. SageMaker aggregates the result in a tar file and uploads to s3.

**Use Your Own Inference Code**

You can use Amazon SageMaker to interact with Docker containers and run your own inference code in one of two ways:

- To use your own inference code with a persistent endpoint to get one prediction at a time, use SageMaker hosting services.
- To use your own inference code to get predictions for an entire dataset, use SageMaker batch transform.

**Topics**

- Use Your Own Inference Code with Hosting Services (p. 1387)
- Use Your Own Inference Code with Batch Transform (p. 1391)

**Use Your Own Inference Code with Hosting Services**

This section explains how Amazon SageMaker interacts with a Docker container that runs your own inference code for hosting services. Use this information to write inference code and create a Docker image.

**Topics**

- How SageMaker Runs Your Inference Image (p. 1388)
- How SageMaker Loads Your Model Artifacts (p. 1389)
- How Containers Serve Requests (p. 1389)
- How Your Container Should Respond to Inference Requests (p. 1389)
- How Your Container Should Respond to Health Check (Ping) Requests (p. 1389)
- Use a Private Docker Registry for Real-Time Inference Containers (p. 1390)
How SageMaker Runs Your Inference Image

To configure a container to run as an executable, use an \texttt{ENTRYPOINT} instruction in a Dockerfile. Note the following:

- For model inference, SageMaker runs the container as:

  
  \begin{verbatim}
  docker run image serve
  \end{verbatim}

  SageMaker overrides default \texttt{CMD} statements in a container by specifying the \texttt{serve} argument after the image name. The \texttt{serve} argument overrides arguments that you provide with the \texttt{CMD} command in the Dockerfile.

- We recommend that you use the \texttt{exec} form of the \texttt{ENTRYPOINT} instruction:

  \begin{verbatim}
  ENTRYPOINT ["executable", "param1", "param2"]
  \end{verbatim}

  For example:

  \begin{verbatim}
  ENTRYPOINT ["python", "k_means_inference.py"]
  \end{verbatim}

  The \texttt{exec} form of the \texttt{ENTRYPOINT} instruction starts the executable directly, not as a child of \texttt{/bin/sh}. This enables it to receive signals like \texttt{SIGTERM} and \texttt{SIGKILL} from the SageMaker APIs, which is a requirement.

For example, when you use the \texttt{CreateEndpoint} API to create an endpoint, SageMaker provisions the number of ML compute instances required by the endpoint configuration, which you specify in the request. SageMaker runs the Docker container on those instances.

If you reduce the number of instances backing the endpoint (by calling the \texttt{UpdateEndpointWeightsAndCapacities} APIs), SageMaker runs a command to stop the Docker container on the instances being terminated. The command sends the \texttt{SIGTERM} signal, then it sends the \texttt{SIGKILL} signal thirty seconds later.

If you update the endpoint (by calling the \texttt{UpdateEndpoint} API), SageMaker launches another set of ML compute instances and runs the Docker containers that contain your inference code on them. Then it runs a command to stop the previous Docker containers. To stop a Docker container, command sends the \texttt{SIGTERM} signal, then it sends the \texttt{SIGKILL} signal 30 seconds later.

- SageMaker uses the container definition that you provided in your \texttt{CreateModel} request to set environment variables and the DNS hostname for the container as follows:

  - It sets environment variables using the \texttt{ContainerDefinition.Environment} string-to-string map.
  - It sets the DNS hostname using the \texttt{ContainerDefinition.ContainerHostname}.
• If you plan to use GPU devices for model inferences (by specifying GPU-based ML compute instances in your CreateEndpointConfig request), make sure that your containers are nvidia-docker compatible. Don't bundle NVIDIA drivers with the image. For more information about nvidia-docker, see NVIDIA/nvidia-docker.

• You can't use the tini initializer as your entry point in SageMaker containers because it gets confused by the train and serve arguments.

How SageMaker Loads Your Model Artifacts

In your CreateModel request, the container definition includes the ModelDataUrl parameter, which identifies the S3 location where model artifacts are stored. SageMaker uses this information to determine from where to copy the model artifacts. It copies the artifacts to the /opt/ml/model directory for use by your inference code.

The ModelDataUrl must point to a tar.gz file. Otherwise, SageMaker won't download the file.

If you trained your model in SageMaker, the model artifacts are saved as a single compressed tar file in Amazon S3. If you trained your model outside SageMaker, you need to create this single compressed tar file and save it in a S3 location. SageMaker decompresses this tar file into /opt/ml/model directory before your container starts.

How Containers Serve Requests

Containers need to implement a web server that responds to /invocations and /ping on port 8080.

How Your Container Should Respond to Inference Requests

To obtain inferences, the client application sends a POST request to the SageMaker endpoint. For more information, see the InvokeEndpoint API. SageMaker passes the request to the container, and returns the inference result from the container to the client. Note the following:

• SageMaker strips all POST headers except those supported by InvokeEndpoint. SageMaker might add additional headers. Inference containers must be able to safely ignore these additional headers.
• To receive inference requests, the container must have a web server listening on port 8080 and must accept POST requests to the /invocations endpoint.
• A customer's model containers must accept socket connection requests within 250 ms.
• A customer's model containers must respond to requests within 60 seconds. The model itself can have a maximum processing time of 60 seconds before responding to the /invocations endpoint. If your model is going to take 50-60 seconds of processing time, the SDK socket timeout should be set to be 70 seconds.

How Your Container Should Respond to Health Check (Ping) Requests

The CreateEndpoint and UpdateEndpoint API calls result in SageMaker starting new inference containers. Soon after container startup, SageMaker starts sending periodic GET requests to the /ping endpoint.

The simplest requirement on the container is to respond with an HTTP 200 status code and an empty body. This indicates to SageMaker that the container is ready to accept inference requests at the / invocations endpoint.
If the container does not begin to pass health checks, by consistently responding with 200s, during the 4 minutes after startup, `CreateEndPoint` will fail, leaving the endpoint in a failed state, and the update requested by `UpdateEndpoint` will not be completed.

While the minimum bar is for the container to return a static 200, a container developer can use this functionality to perform deeper checks. The request timeout on `/ping` attempts is 2 seconds.

**Use a Private Docker Registry for Real-Time Inference Containers**

Amazon SageMaker hosting enables you to use images stored in Amazon ECR to build your containers for real-time inference by default. Optionally, you can build containers for real-time inference from images in a private Docker registry. The private registry must be accessible from an Amazon VPC in your account. Models that you create based on the images stored in your private Docker registry must be configured to connect to the same VPC where you create the private Docker registry. For information about connecting your model to a VPC, see Give SageMaker Hosted Endpoints Access to Resources in Your Amazon VPC (p. 1697).

Your Docker registry must be secured with a TLS certificate from a known public certificate authority (CA).

**Note**

Your private Docker registry must allow inbound traffic from the security group(s) you specify in the VPC configuration for your model, so that SageMaker hosting is able to pull model images from your registry.

**Topics**

- Store Images in a Private Docker Registry (p. 1390)
- Use an Image from a Private Docker Registry for Real-time Inference (p. 1390)

**Store Images in a Private Docker Registry**

To use a private Docker registry to store your images for SageMaker real-time inference, create a private registry that is accessible from your Amazon VPC. For information about creating a Docker registry, see Deploy a registry server in the Docker documentation. The Docker registry must comply with the following:

- The registry must be a Docker Registry HTTP API V2 registry.
- The Docker registry must be accessible from the same VPC that you specify in the `VpcConfig` parameter that you specify when you create your model.
- The Docker registry must not require any authentication.

**Use an Image from a Private Docker Registry for Real-time Inference**

When you create a model and deploy it to SageMaker hosting, you can specify that it use an image from your private Docker registry to build the inference container.

**To use an image stored in your private Docker registry for your inference container**

1. Specify `Vpc` as the value of the `RepositoryAccess` field of the `ImageConfig` setting for the primary container when you call the `create_model` function.

```python
model_name = 'vpc-model'
execution_role_arn = 'arn:aws:iam::123456789012:role/SageMakerExecutionRole'
primary_container = {
    'ContainerHostname': 'ModelContainer',
    'Image': 'myteam.myorg.com/docker-local/my-inference-image:latest',
```
Use Your Own Inference Code

2. Specify one or more security groups and subnets for the VPC configuration for your model. Your private Docker registry must allow inbound traffic from the security group(s) that you specify. The subnets that you specify must be in the same VPC as your private Docker registry.

```python
vpc_config = {
    'SecurityGroupIds': ['sg-0123456789abcdef0'],
    'Subnets': ['subnet-0123456789abcdef0','subnet-0123456789abcdef1']
}
```

3. Create the model by calling `create_model`, using the values you specified in the previous steps for the `PrimaryContainer` and `VpcConfig` parameters.

```python
try:
    resp = sm.create_model(
        ModelName=model_name,
        PrimaryContainer=primary_container,
        ExecutionRoleArn=execution_role_arn,
        VpcConfig=vpc_config,
    )
except Exception as e:
    print(f'error calling CreateModel operation: {e}')
else:
    print(resp)
```

4. Finally, call `create_endpoint_config` and `create_endpoint` to create the hosting endpoint, using the model that you created in the previous step.

```python
endpoint_config_name = 'my-endpoint-config'
sm.create_endpoint_config(
    EndpointConfigName=endpoint_config_name,
    ProductionVariants=[
        {
            'VariantName': 'MyVariant',
            'ModelName': model_name,
            'InitialInstanceCount': 1,
            'InstanceType': 'ml.t2.medium'
        },
    ],
)
endpoint_name = 'my-endpoint'
sm.create_endpoint(
    EndpointName=endpoint_name,
    EndpointConfigName=endpoint_config_name,
)
sm.describe_endpoint(EndpointName=endpoint_name)
```

Use Your Own Inference Code with Batch Transform

This section explains how Amazon SageMaker interacts with a Docker container that runs your own inference code for batch transform. Use this information to write inference code and create a Docker image.

Topics
How SageMaker Runs Your Inference Image

To configure a container to run as an executable, use an `ENTRYPOINT` instruction in a Dockerfile. Note the following:

- For batch transforms, SageMaker runs the container as:
  ```
  docker run image serve
  ```

  SageMaker overrides default `CMD` statements in a container by specifying the `serve` argument after the image name. The `serve` argument overrides arguments that you provide with the `CMD` command in the Dockerfile.

- We recommend that you use the `exec` form of the `ENTRYPOINT` instruction:
  ```
  ENTRYPOINT ["executable", "param1", "param2"]
  ```

  For example:
  ```
  ENTRYPOINT ["python", "k_means_inference.py"]
  ```

- SageMaker sets environment variables specified in `CreateModel` and `CreateTransformJob` on your container. Additionally, the following environment variables are populated:
  - `SAGEMAKER_BATCH` is always set to `true` when the container runs in Batch Transform.
  - `SAGEMAKER_MAX_PAYLOAD_IN_MB` is set to the largest size payload that is sent to the container via HTTP.
  - `SAGEMAKER_BATCH_STRATEGY` is set to `SINGLE_RECORD` when the container is sent a single record per call to invocations and `MULTI_RECORD` when the container gets as many records as will fit in the payload.
  - `SAGEMAKER_MAX_CONCURRENT_TRANSFORMS` is set to the maximum number of `/invocations` requests that can be opened simultaneously.

  **Note**
  The last three environment variables come from the API call made by the user. If the user doesn't set values for them, they aren't passed. In that case, either the default values or the values requested by the algorithm (in response to the `/execution-parameters`) are used.

- If you plan to use GPU devices for model inferences (by specifying GPU-based ML compute instances in your `CreateTransformJob` request), make sure that your containers are nvidia-docker compatible. Don't bundle NVIDIA drivers with the image. For more information about nvidia-docker, see NVIDIA/nvidia-docker.

- You can't use the `init` initializer as your entry point in SageMaker containers because it gets confused by the train and serve arguments.
How SageMaker Loads Your Model Artifacts

In a CreateModel request, container definitions include the ModelDataUrl parameter, which identifies the location in Amazon S3 where model artifacts are stored. When you use SageMaker to run inferences, it uses this information to determine from where to copy the model artifacts. It copies the artifacts to the /opt/ml/model directory in the Docker container for use by your inference code.

The ModelDataUrl parameter must point to a tar.gz file. Otherwise, SageMaker can't download the file. If you train a model in SageMaker, it saves the artifacts as a single compressed tar file in Amazon S3. If you train a model in another framework, you need to store the model artifacts in Amazon S3 as a compressed tar file. SageMaker decompresses this tar file and saves it in the /opt/ml/model directory in the container before the batch transform job starts.

How Containers Serve Requests

Containers must implement a web server that responds to invocations and ping requests on port 8080. For batch transforms, you have the option to set algorithms to implement execution-parameters requests to provide a dynamic runtime configuration to SageMaker. SageMaker uses the following endpoints:

- ping—Used to periodically check the health of the container. SageMaker waits for an HTTP 200 status code and an empty body for a successful ping request before sending an invocations request. You might use a ping request to load a model into memory to generate inference when invocations requests are sent.
- (Optional) execution-parameters—Allows the algorithm to provide the optimal tuning parameters for a job during runtime. Based on the memory and CPUs available for a container, the algorithm chooses the appropriate MaxConcurrentTransforms, BatchStrategy, and MaxPayloadInMB values for the job.

Before calling the invocations request, SageMaker attempts to invoke the execution-parameters request. When you create a batch transform job, you can provide values for the MaxConcurrentTransforms, BatchStrategy, and MaxPayloadInMB parameters. SageMaker determines the values for these parameters using this order of precedence:

1. The parameter values that you provide when you create the CreateTransformJob request.
2. The values that the model container returns when SageMaker invokes the execution-parameters endpoint
3. The default parameter values, listed in the following table.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Default Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>MaxConcurrentTransforms</td>
<td>1</td>
</tr>
<tr>
<td>BatchStrategy</td>
<td>MULTI_RECORD</td>
</tr>
<tr>
<td>MaxPayloadInMB</td>
<td>6</td>
</tr>
</tbody>
</table>

The response for a GET execution-parameters request is a JSON object with keys for MaxConcurrentTransforms, BatchStrategy, and MaxPayloadInMB parameters. This is an example of a valid response:

```json
{
}
```
How Your Container Should Respond to Inference Requests

To obtain inferences, Amazon SageMaker sends a POST request to the inference container. The POST request body contains data from Amazon S3. Amazon SageMaker passes the request to the container, and returns the inference result from the container, saving the data from the response to Amazon S3.

To receive inference requests, the container must have a web server listening on port 8080 and must accept POST requests to the /invocations endpoint. Your model containers must respond to requests within 600 seconds.

How Your Container Should Respond to Health Check (Ping) Requests

The simplest requirement on the container is to respond with an HTTP 200 status code and an empty body. This indicates to SageMaker that the container is ready to accept inference requests at the /invocations endpoint.

While the minimum bar is for the container to return a static 200, a container developer can use this functionality to perform deeper checks. The request timeout on /ping attempts is 2 seconds.

Example Notebooks: Use Your Own Algorithm or Model

The following Jupyter notebooks show how to use your own algorithms or pretrained models from an Amazon SageMaker notebook instance. For links to the GitHub repositories with the prebuilt Dockerfiles for the TensorFlow, MXNet, Chainer, and PyTorch frameworks and instructions on using the AWS SDK for Python (Boto3) estimators to run your own training algorithms on SageMaker Learner and your own models on SageMaker hosting, see Prebuilt SageMaker Docker Images for TensorFlow, MXNet, Chainer, and PyTorch (p. 1358)

Setup

1. Create a SageMaker notebook instance. For instructions on how to create and access Jupyter notebook instances, see Use Amazon SageMaker Notebook Instances (p. 124).
2. Open the notebook instance you created.
3. Choose the SageMaker Examples tab for a list of all SageMaker example notebooks.
4. Open the sample notebooks from the Advanced Functionality section in your notebook instance or from GitHub using the provided links. To open a notebook, choose its Use tab, then choose Create copy.

Host Models Trained in Scikit-learn

To learn how to host models trained in Scikit-learn for making predictions in SageMaker by injecting them into first-party k-means and XGBoost containers, see the following sample notebooks.

- kmeansbring_your_own_model
- xgboostbring_your_own_model
Package TensorFlow and Scikit-learn Models for Use in SageMaker

To learn how to package algorithms that you have developed in TensorFlow and scikit-learn frameworks for training and deployment in the SageMaker environment, see the following notebooks. They show you how to build, register, and deploy your own Docker containers using Dockerfiles.

- tensorflow_bring_your_own
- scikit_bring_your_own

Train and Deploy a Neural Network on SageMaker

To learn how to train a neural network locally using MXNet or TensorFlow, and then create an endpoint from the trained model and deploy it on SageMaker, see the following notebooks. The MXNet model is trained to recognize handwritten numbers from the MNIST dataset. The TensorFlow model is trained to classify irises.

- mxnet_mnist_byom
- tensorflow_iris_byom

Training Using Pipe Mode

To learn how to use a Dockerfile to build a container that calls the train.py script and uses pipe mode to custom train an algorithm, see the following notebook. In pipe mode, the input data is transferred to the algorithm while it is training. This can decrease training time compared to using file mode.

- pipe_bring_your_own

Bring Your Own R Model

To learn how to use an R container to train and host a model with the R kernel installed in a notebook, see the following notebook. To take advantage of the AWS SDK for Python (Boto3), we use Python within the notebook. You can achieve the same results in R by invoking command line arguments.

- r_bring_your_own

Extend a Prebuilt PyTorch Container Image

To learn how to extend a prebuilt SageMaker PyTorch container image when you have additional functional requirements for your algorithm or model that the prebuilt Docker image doesn't support, see the following notebook.

- pytorch_extending_our_containers
Train and Debug Training Jobs on a Custom Container

To learn how to train and debug training jobs using SageMaker Debugger, see the following notebook. A training script provided through this example uses the TensorFlow Keras ResNet 50 model and the CIFAR10 dataset. A Docker custom container is built with the training script and pushed to Amazon ECR. While the training job is running, Debugger collects tensor outputs and identifies debugging problems. With `smdebug` client library tools, you can set a `smdebug` trial object that calls the training job and debugging information, check the training and Debugger rule status, and retrieve tensors saved in an Amazon S3 bucket to analyze training issues.

- `build_your_own_container_with_debugger`
SageMaker Workflows

You can manage your Amazon SageMaker training and inference workflows using Amazon SageMaker Studio and the Amazon SageMaker Python SDK. With the available tools, you can simplify your SageMaker process and integrate it into your existing project.

The following workflow technologies are supported.

- **Amazon SageMaker Model Building Pipelines (p. 1397)**: SageMaker's tool for building and managing end-to-end ML pipelines.
- **Airflow Workflows**: SageMaker APIs to export configurations for creating and managing Airflow workflows.
- **Kubernetes Orchestration (p. 1470)**: SageMaker custom operators for your Kubernetes cluster, as well as custom components for Kubeflow Pipelines.
- **AWS Step Functions**: Create multi-step machine learning workflows in Python that orchestrate SageMaker infrastructure without having to provision your resources separately.

For more information on managing SageMaker training and inference, see Amazon SageMaker Python SDK Workflows.

**Topics**

- Amazon SageMaker Model Building Pipelines (p. 1397)
- Automate MLOps with SageMaker Projects (p. 1439)
- Amazon SageMaker ML Lineage Tracking (p. 1463)
- Kubernetes Orchestration (p. 1470)

Amazon SageMaker Model Building Pipelines

Amazon SageMaker Model Building Pipelines is a tool for building machine learning pipelines that take advantage of direct SageMaker integration. Because of this integration, you can create a pipeline and set up SageMaker Projects for orchestration using a tool that handles much of the step creation and management for you. SageMaker Pipelines provides the following advantages over other AWS workflow offerings:

**SageMaker Integration**

SageMaker Pipelines is integrated directly with SageMaker, so you don't need to interact with any other AWS services. You also don't need to manage any resources because SageMaker Pipelines is a fully managed service, which means that it creates and manages resources for you.

**SageMaker Python SDK Integration**

Because SageMaker Pipelines is integrated with the SageMaker Python SDK, you can create your pipelines programmatically using a high-level Python interface that you may already be familiar with. This enables you and members of your team to build your workflow without having to manage your jobs using the AWS console. To view the SageMaker Python SDK documentation, see Pipelines.
SageMaker Studio Integration

SageMaker Studio offers an environment to manage the end-to-end SageMaker Pipelines experience. Using Studio, you can bypass the AWS console for your entire workflow management. For more information on managing SageMaker Pipelines from SageMaker Studio, see View, Track, and Execute SageMaker Pipelines in SageMaker Studio (p. 1426).

Data Lineage Tracking

With SageMaker Pipelines you can track the history of your data within the pipeline execution. It lets you analyze where the data came from, where it was used as an input, and the outputs that were generated from it. You can view the models created from an individual dataset and the datasets that went into creating an individual model. For more information, see Amazon SageMaker ML Lineage Tracking (p. 1463).

Step Reuse

With SageMaker Pipelines, you can designate steps for caching. When a step is cached, it is indexed for reuse later if the same step is executed again. As a result, you can reuse the output from previous step executions of the same step in the same pipeline without having to run the step again. For more information on step caching, see Caching Pipeline Steps (p. 1407).

Topics

- SageMaker Pipelines Overview (p. 1398)
- Create and Manage SageMaker Pipelines (p. 1410)

SageMaker Pipelines Overview

An Amazon SageMaker Model Building Pipelines pipeline is a series of interconnected steps that is defined by a JSON pipeline definition. This pipeline definition encodes a pipeline using a directed acyclic graph (DAG). This DAG gives information on the requirements for and relationships between each step of your pipeline. The structure of a pipeline's DAG is determined by the data dependencies between steps. These data dependencies are created when the properties of a step's output are passed as the input to another step. The following image is an example of a pipeline DAG:
The following topics describe fundamental SageMaker Pipelines concepts. For a tutorial describing the implementation of these concepts, see Create and Manage SageMaker Pipelines (p. 1410).

**Topics**
- Pipeline Structure (p. 1400)
- Access Management (p. 1400)
- Pipeline Parameters (p. 1401)
- Pipeline Steps (p. 1402)
- Property Files and JsonGet (p. 1406)
- Caching Pipeline Steps (p. 1407)
- SageMaker Pipelines Quotas (p. 1408)
Pipeline Structure

A pipeline instance is composed of a name, parameters, and steps. Pipeline names must be unique within an (account, region) pair. All parameters used in step definitions must be defined in the pipeline. Steps passed into the pipeline don’t need to be listed in the order of execution because the steps themselves define the relationships between them using data dependencies. The SageMaker Pipelines service resolves the relationships between steps in the data dependency DAG to create a series of steps that the execution completes.

```python
from sagemaker.workflow.pipeline import Pipeline

pipeline_name = f"AbalonePipeline"
pipeline = Pipeline(
   name=pipeline_name,
   parameters=[
    processing_instance_type,
    processing_instance_count,
    training_instance_type,
    model_approval_status,
    input_data,
    batch_data,
   ],
   steps=[step_process, step_train, step_eval, step_cond],
)
```

Access Management

The following sections describe the AWS Identity and Access Management (IAM) requirements for Amazon SageMaker Model Building Pipelines. For an example of how you can implement these permissions, see Prerequisites (p. 1412).

Topics

- Pipeline Role Permissions (p. 1400)
- Pipeline Step Permissions (p. 1401)
- Service Control Policies with Pipelines (p. 1401)

Pipeline Role Permissions

Your pipeline requires an IAM pipeline execution role that is passed to SageMaker Pipelines when you create a pipeline. The role for the SageMaker instance that is creating the pipeline must have the `iam:PassRole` permission for the pipeline execution role in order to pass it. For more information on IAM roles, see IAM Roles.

Your pipeline execution role requires the following permissions:

- To pass any role to a SageMaker job within a pipeline, the `iam:PassRole` permission for the role that is being passed.
- Create and Describe permissions for each of the job types in the pipeline.
- Amazon S3 permissions to use the `JsonGet` function. You control access to your Amazon S3 resources using resource-based policies and identity-based policies. A resource-based policy is applied to your Amazon S3 bucket and grants SageMaker Pipelines access to the bucket. An identity-based policy gives your pipeline the ability to make Amazon S3 calls from your account. For more information on
resource-based policies and identity-based policies, see Identity-based policies and resource-based policies.

```
{
   "Action": [
      "s3:GetObject",
      "s3:HeadObject"
   ],
   "Resource": "arn:aws:s3::<your-bucket-arn>/*",
   "Effect": "Allow"
}
```

**Pipeline Step Permissions**

SageMaker Pipelines include steps that run SageMaker jobs. In order for the pipeline steps to run these jobs, they require an IAM role in your account that provides access for the needed resource. This role is passed to the SageMaker service principal by your pipeline. For more information on IAM roles, see IAM Roles.

By default, each step takes on the pipeline execution role. You can optionally pass a different role to any of the steps in your pipeline. This ensures that the code in each step does not have the ability to impact resources used in other steps unless there is a direct relationship between the two steps specified in the pipeline definition. You pass these roles when defining the processor or estimator for your step. For examples of how to include these roles in these definitions, see the SageMaker Python SDK documentation.

**Service Control Policies with Pipelines**

Service control policies (SCPs) are a type of organization policy that you can use to manage permissions in your organization. SCPs offer central control over the maximum available permissions for all accounts in your organization. By using SageMaker Pipelines within your organization, you can ensure that data scientists manage your pipeline executions without having to interact with the AWS console.

If you're using a VPC with your SCP that restricts access to Amazon S3, you need to take steps to allow your pipeline to access other Amazon S3 resources.

To allow SageMaker Pipelines to access Amazon S3 outside of your VPC with the JsonGet function, update your organization's SCP to ensure that the role using SageMaker Pipelines can access Amazon S3. To do this, create an exception for roles that are being used by the SageMaker Pipelines executor via the pipeline execution role using a principal tag and condition key.

**To allow SageMaker Pipelines to access Amazon S3 outside of your VPC**

1. Create a unique tag for your pipeline execution role following the steps in Tagging IAM users and roles.
2. Grant an exception in your SCP using the Aws:PrincipalTag IAM condition key for the tag you created. For more information, see Creating, updating, and deleting service control policies.

**Pipeline Parameters**

You can introduce variables into your pipeline definition using parameters. Parameters that you define can be referenced throughout your pipeline definition. Parameters have a default value, which you can override by specifying parameter values when starting a pipeline execution. The default value must be an instance matching the parameter type. All parameters used in step definitions must be defined in your pipeline definition. Amazon SageMaker Model Building Pipelines supports the following parameter types:
• **ParameterString** – Representing an `str` Python type.
• **ParameterInteger** – Representing an `int` Python type.
• **ParameterFloat** – Representing a `float` Python type.

Parameters take the following format:

```python
<parameter> = <parameter_type>(
    name=<parameter_name>,
    default_value=<default_value>
)
```

The following shows an example parameter implementation:

```python
from sagemaker.workflow.parameters import (ParameterInteger, ParameterString, ParameterFloat)

processing_instance_count = ParameterInteger(
    name="ProcessingInstanceCount",
    default_value=1
)
```

You then pass the parameter when creating your pipeline as follows:

```python
pipeline = Pipeline(
    name=pipeline_name,
    parameters=[
        processing_instance_count
    ],
    steps=[step_process],
)
```

You can also pass a parameter value that differs from the default value to a pipeline execution as follows:

```python
execution = pipeline.start(
    parameters=dict(
        ProcessingInstanceType="ml.c5.xlarge",
        ModelApprovalStatus="Approved",
    )
)
```

### Pipeline Steps

SageMaker Pipelines are composed of steps. These steps define the actions that the pipeline takes, and the relationships between steps using properties.

Amazon SageMaker Model Building Pipelines support the following step types:

• Processing
• Training
• Condition
• BatchTransform
• RegisterModel
• CreateModel
The following describes the requirements of each step and provides an example implementation of the step. These are not functional implementations because they don't provide the resource and inputs needed. For a tutorial that implements these steps, see Create and Manage SageMaker Pipelines (p. 1410).

**Processing**

You use a processing step to create a processing job for data processing. For more information on processing jobs, see Process Data and Evaluate Models.

A processing step requires a processor, a Python script that defines the processing code, outputs for processing, and job arguments. The following example shows how to create a ProcessingStep definition. For more information on processing step requirements, see the `sagemaker.workflow.steps.ProcessingStep` documentation.

```python
from sagemaker.processing import ProcessingInput, ProcessingOutput
from sagemaker.workflow.steps import ProcessingStep

step_process = ProcessingStep(
    name="AbaloneProcess",
    processor=sklearn_processor,
    inputs=[
        ProcessingInput(source=input_data, destination="/opt/ml/processing/input"),
    ],
    outputs=[
        ProcessingOutput(output_name="train", source="/opt/ml/processing/train"),
        ProcessingOutput(output_name="validation", source="/opt/ml/processing/validation"),
        ProcessingOutput(output_name="test", source="/opt/ml/processing/test")
    ],
    code="abalone/preprocessing.py",
)
```

**Training**

You use a training step to create a training job to train a model. For more information on training jobs, see Train a Model with Amazon SageMaker.

A training step requires an estimator, and training and validation data inputs. The following example shows how to create a TrainingStep definition. For more information on Training step requirements, see the `sagemaker.workflow.steps.TrainingStep` documentation.

```python
from sagemaker.inputs import TrainingInput
from sagemaker.workflow.steps import TrainingStep

step_train = TrainingStep(
    name="TrainAbaloneModel",
    estimator=xgb_train,
    inputs={
        "train": TrainingInput(
            s3_data=step_process.properties.ProcessingOutputConfig.Outputs["train"].S3Output.S3Uri,
            content_type="text/csv"
        ),
        "validation": TrainingInput(
            s3_data=step_process.properties.ProcessingOutputConfig.Outputs["validation"].S3Output.S3Uri,
            content_type="text/csv"
        )
    },
)
```
**Condition**

You use a condition step to evaluate the condition of step properties to assess which action should be taken next in the pipeline.

A condition step requires a list of conditions, and a list of steps to execute if the condition evaluates to true and a list of steps to execute if the condition evaluates to false. The following example shows how to create a Condition step definition. For more information on Condition step requirements, see the `sagemaker.workflow.condition_step.ConditionStep` documentation.

**Note**

SageMaker Pipelines doesn't support the use of nested condition steps. You can't pass a condition step as the input for another condition step.

```python
from sagemaker.workflow.conditions import ConditionLessThanOrEqualTo
from sagemaker.workflow.condition_step import (ConditionStep,
                                            JsonGet,
                                            )

step_cond = ConditionStep(  
    name="AbaloneMSECond",
    conditions=[cond_lte],
    if_steps=[step_register, step_create_model, step_transform],
    else_steps=[],
)
```

**Transform**

You use a transform step for batch transformation to run inference on an entire dataset. For more information on batch transformation, see Run Batch Transforms with Inference Pipelines.

A transform step requires a transformer, and the data to run batch transformation on. The following example shows how to create a Transform step definition. For more information on Transform step requirements, see the `sagemaker.workflow.steps.TransformStep`.

```python
from sagemaker.inputs import TransformInput
from sagemaker.workflow.steps import TransformStep

step_transform = TransformStep(  
    name="AbaloneTransform",
    transformer=transformer,
    inputs=TransformInput(data=batch_data)
)
```

**RegisterModel**

You use a RegisterModel step to register a model to a model group. For more information on registering models, see Register and Deploy Models with Model Registry (p. 1136).

A RegisterModel step requires an estimator, model data output from training, and a model package group name to associate the model package with. The following example shows how to create a RegisterModel definition. For more information on RegisterModel step requirements, see the `sagemaker.workflow.step_collections.RegisterModel`.

```python
from sagemaker.workflow.step_collections import RegisterModel
```
step_register = RegisterModel(
    name="AbaloneRegisterModel",
    estimator=xgb_train,
    model_data=step_train.properties.ModelArtifacts.S3ModelArtifacts,
    content_types=["text/csv"],
    response_types=["text/csv"],
    inference_instances=["ml.t2.medium", "ml.m5.xlarge"],
    transform_instances=["ml.m5.xlarge"],
    model_package_group_name=model_package_group_name,
    approval_status=model_approval_status,
    model_metrics=model_metrics
)

CreateModel

You use a CreateModel step to create a SageMaker Model. For more information on SageMaker Models, see Train a Model with Amazon SageMaker.

A CreateModel step requires model artifacts, and information on the SageMaker instance type that you need to use to create the model. The following example shows how to create a CreateModel step definition. For more information on CreateModel step requirements, see the sagemaker.workflow.steps.CreateModelStep documentation.

from sagemaker.workflow.steps import CreateModelStep

step_create_model = CreateModelStep(
    name="AbaloneCreateModel",
    model=model,
    inputs=inputs,
)

For more information on the available steps and their input requirements, see the Pipelines documentation.

Step Properties

The properties attribute is used to add data dependencies between steps in the pipeline. These data dependencies are then used by SageMaker Pipelines to construct the DAG from the pipeline definition. These properties can be referenced as placeholder values and are resolved at runtime.

The properties attribute of a SageMaker Pipelines step matches the object returned by a Describe call for the corresponding SageMaker job type. For each job type, the Describe call returns the following response object:

- ProcessingStep – DescribeProcessingJob
- TrainingStep – DescribeTrainingJob
- TransformStep – DescribeTransformJob

Data Dependency Between Steps

You define the structure of your DAG by specifying the data relationships between steps. To create data dependencies between steps, pass the properties of one step as the input to another step in the pipeline. A data dependency uses JsonPath notation in the following format. This format traverses the JSON property file, which means you can append as many <property> instances as needed to reach the desired nested property in the file. For more information on JsonPath notation, see the JsonPath repo.

<step_name>.properties.<property>.<property>
The following is an example of a data dependency using the ProcessingOutputConfig property of a processing step:

```python
step_processing.properties.ProcessingOutputConfig.Outputs["train_data"].S3Output.S3Uri
```

The data dependency can be passed as the input to a step as follows:

```python
step_train = TrainingStep(
    name="CensusTrain",
    estimator=sklearn_train,
    inputs=TrainingInput(
        s3_data=step_process.properties.ProcessingOutputConfig.Outputs["train_data"].S3Output.S3Uri
    ),
)
```

**Using Custom Images in Pipelines**

You can use any of the available SageMaker Deep Learning Container images when you create a step in your pipeline.

You can also create a step using SageMaker S3 applications. A SageMaker S3 application is a tar.gz bundle with one or more Python scripts that can run within that bundle. For more information on application package bundling, see Deploying directly from model artifacts.

You can also use your own container with pipeline steps. Because you can't create an image from within SageMaker Studio, you must create your image using another method before using it with Amazon SageMaker Model Building Pipelines.

To use your own container when creating the steps for your pipeline, include the image URI in the estimator definition. For more information on using your own container with SageMaker, see Using Docker Containers with SageMaker.

**Property Files and JsonGet**

You use property files to store information from the output of a processing step. This is particularly useful when analyzing the results of a processing step to decide how a conditional step should be executed. The JsonGet function processes a property file and enables you to use JsonPath notation to query the property JSON file. For more information on JsonPath notation, see the JsonPath repo.

To store a property file for later use, you must first create a PropertyFile instance with the following format. The path parameter is the name of the JSON file that the property file is saved to. output_name must match the output_name of the ProcessingOutput that you define in your processing step. This enables the property file to capture the ProcessingOutput in the step.

```python
from sagemaker.workflow.properties import PropertyFile

<property_file_instance> = PropertyFile(
    name="<property_file_name>",
    output_name="<processingoutput_output_name>",
    path="<path_to_json_file>",
)
```

When you create your ProcessingStep instance, add the property_files parameter to list all of the parameter files that the Amazon SageMaker Model Building Pipelines service must index. This saves the property file for later use.

```python
property_files=[<property_file_instance>]
```
To use your property file in a condition step, add the `property_file` to the condition that you pass to your condition step as follows. This enables you to query the JSON file for your desired property using the `json_path` parameter.

```python
cond_lte = ConditionLessThanOrEqualTo(
    left=JsonGet(
        step=step_eval,
        property_file=<property_file_instance>,
        json_path="mse"
    ),
    right=6.0
)
```

## Caching Pipeline Steps

Using step signature caching, Amazon SageMaker Model Building Pipelines attempts to find a previous execution of a step with the same arguments. If a previous execution is found, a cache hit is created. SageMaker Pipelines propagates the values from the cache hit during execution, rather than recomputing the step.

Step caching only considers successful executions, so no failed executions are ever reused. When multiple successful executions exist within the timeout period, Pipelines uses the result for the most recent successful execution. If no successful executions match in the timeout period, Pipelines won't reuse any steps. If the executor finds a cache hit for a previous step execution that is still in progress, both steps continue executing and update the cache, if they're successful.

You must opt-in to step caching, otherwise it is off by default. When you enable step caching, you must also define a timeout. This timeout defines how old a previous execution can be to be considered for reuse.

Step caching is only scoped for individual pipelines, so you can't reuse a step from another pipeline. Even if there is a step signature match in the other pipeline, the step is not reused.

Step caching is available for the following step types:
- Training
- Processing
- Transform

## Enabling Step Caching

To enable step caching, you must add a `CacheConfig` property to the step definition.

`CacheConfig` properties use the following format:

```json
{
    "CacheConfig": {
        "Enabled": false,
        "ExpireAfter": "<time>"
    }
}
```

The `Enabled` field may be true or false. `ExpireAfter` is a string that defines the timeout period. Any ISO 8601 duration string is a valid `ExpireAfter` value. The `ExpireAfter` duration can contain a year, month, week, day, hour, and minute value. Each value is consists of a number followed by a letter indicating the duration unit it is for. For example:

- "30d" = Thirty days
- “5y” = Five years
- “T16m” = 16 minutes
- “30dT5h” = 30 days and five hours.

SageMaker Pipelines Quotas

The following tables list the quotas for SageMaker Pipelines resources and requirements for pipeline creation.

Pipeline Quotas

<table>
<thead>
<tr>
<th>Pipeline Property</th>
<th>Quota</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum number of pipelines</td>
<td>5000</td>
</tr>
</tbody>
</table>

Executions Quotas

<table>
<thead>
<tr>
<th>Execution Property</th>
<th>Quota</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum execution time</td>
<td>28 days</td>
</tr>
<tr>
<td>Maximum number of concurrent pipeline executions per account</td>
<td>200</td>
</tr>
<tr>
<td>Maximum number of concurrent pipeline executions per pipeline</td>
<td>200</td>
</tr>
</tbody>
</table>

Step Quotas

<table>
<thead>
<tr>
<th>Step Property</th>
<th>Quota</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum number of steps per pipeline</td>
<td>50</td>
</tr>
<tr>
<td>Maximum step name length</td>
<td>64 characters</td>
</tr>
<tr>
<td>Step name regular expression pattern</td>
<td>([A-Za-z0-9-_])*</td>
</tr>
</tbody>
</table>

Parameters Quotas

<table>
<thead>
<tr>
<th>Parameters Property</th>
<th>Quota</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum number of parameters per pipeline</td>
<td>200</td>
</tr>
<tr>
<td>Maximum parameter name length</td>
<td>64 characters</td>
</tr>
<tr>
<td>Maximum parameter name regular expression pattern</td>
<td>([A-Za-z0-9-_])*</td>
</tr>
<tr>
<td>Maximum parameter description length</td>
<td>4096 characters</td>
</tr>
<tr>
<td>Maximum number of parameter enum values</td>
<td>16 distinct values</td>
</tr>
<tr>
<td>Parameters Property</td>
<td>Quota</td>
</tr>
<tr>
<td>----------------------------------------------</td>
<td>-----------------------------</td>
</tr>
<tr>
<td>Maximum parameter enum value length</td>
<td>2048 characters</td>
</tr>
<tr>
<td>Maximum string default value limit</td>
<td>2048 characters</td>
</tr>
</tbody>
</table>

**Condition Step Quotas**

<table>
<thead>
<tr>
<th>Condition Step Property</th>
<th>Quota</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum number of Conditions per ConditionStep</td>
<td>200</td>
</tr>
<tr>
<td>Maximum number of Steps in If-List</td>
<td>20</td>
</tr>
<tr>
<td>Maximum number of Steps in Else-List</td>
<td>20</td>
</tr>
<tr>
<td>Maximum number of Conditions in an Or-list</td>
<td>200</td>
</tr>
</tbody>
</table>

**Property Files Quotas**

<table>
<thead>
<tr>
<th>Property Files Property</th>
<th>Quota</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum number of PropertyFiles in a pipeline</td>
<td>10</td>
</tr>
<tr>
<td>Maximum number of JsonGet functions in a pipeline</td>
<td>200</td>
</tr>
<tr>
<td>Maximum size of the property file</td>
<td>2MB</td>
</tr>
</tbody>
</table>

**Metadata Quotas**

<table>
<thead>
<tr>
<th>Metadata Property</th>
<th>Quota</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum metadata size</td>
<td>20 characters</td>
</tr>
<tr>
<td>Maximum metadata key size</td>
<td>128 characters</td>
</tr>
<tr>
<td>Metadata key regular expression pattern</td>
<td>([A-Za-z0-9-_])*</td>
</tr>
<tr>
<td>Maximum metadata value size</td>
<td>1024 characters</td>
</tr>
<tr>
<td>Metadata value regular expression pattern</td>
<td>[\p{M}\p{L}\p{S}\p{S}\p{N}\p{P}\s]</td>
</tr>
</tbody>
</table>

**Troubleshooting Amazon SageMaker Model Building Pipelines**

When using Amazon SageMaker Model Building Pipelines, you might run into issues for various reasons. This topic provides information about common errors and how to resolve them.

**Pipeline Definition Issues**

Your pipeline definition might not be formatted correctly. This can result in your execution failing or your job being inaccurate. These errors can be caught when the pipeline is created or when an execution
occurs. If your definition doesn't validate, SageMaker Pipelines returns an error message identifying the character where the JSON file is malformed. To fix this problem, review the steps created using the SageMaker Python SDK for accuracy.

You can only include steps in a pipeline definition once. Because of this, steps cannot exist as part of a Condition step and a pipeline in the same pipeline.

Examining Pipeline Logs

You can view the status of your steps using the following command:

```
execution.list_steps()
```

Each step includes the following information:

- The ARN of the entity launched by the pipeline – Such as SageMaker job ARN, model ARN, or model package ARN.
- The failure reason – Includes a brief explanation of the step failure.
- Condition step evaluation – If the step is a condition step, it includes whether the condition is evaluated to true or false.
- The CacheHit – If the execution is reusing a previous job execution, it lists the source execution.

You can also view the error messages and logs in the SageMaker Studio interface. For information about how to see the logs in Studio, see View a Pipeline Execution (p. 1430).

Missing Permissions

Correct permissions are required for the role that creates the pipeline execution, and the steps that create each of the jobs in your pipeline execution. Without these permissions, you may not be able to submit your pipeline execution or run your SageMaker jobs as expected. To ensure that your permissions are properly set up, see Access Management (p. 1400).

Job Execution Errors

You may run into issues when executing your steps because of issues in the scripts that define the functionality of your SageMaker jobs. Each job has a set of CloudWatch logs. To view these logs from SageMaker Studio, see View a Pipeline Execution (p. 1430). For information about using CloudWatch logs with SageMaker, see Log Amazon SageMaker Events with Amazon CloudWatch (p. 1712).

Property File Errors

You may have issues when incorrectly implementing property files with your pipeline. To ensure that your implementation of property files works as expected, see Property Files and JsonGet (p. 1406).

Create and Manage SageMaker Pipelines

You can use Amazon SageMaker Model Building Pipelines to create end-to-end workflows that manage and deploy SageMaker jobs. SageMaker Pipelines comes with SageMaker Python SDK integration, so you can build each step of your pipeline using a Python-based interface.

After your pipeline is deployed, you can view the directed acyclic graph (DAG) for your pipeline and manage your executions using Amazon SageMaker Studio. Using SageMaker Studio, you can get information about your current and historical pipelines, compare executions, see the DAG for your executions, get metadata information, and more. To learn how to view pipelines from SageMaker Studio, see View, Track, and Execute SageMaker Pipelines in SageMaker Studio (p. 1426).
Define a Pipeline

To orchestrate your workflows with Amazon SageMaker Model Building Pipelines, you need to generate a directed acyclic graph (DAG) in the form of a JSON pipeline definition. The following image is a representation of the pipeline DAG that you create in this tutorial:

You can generate your JSON pipeline definition using the SageMaker Python SDK. The following tutorial shows how to generate a pipeline definition for a pipeline that solves a regression problem to determine...
the age of an abalone based on its physical measurements. For a Jupyter notebook that includes the content in this tutorial that you can run, see Orchestrating Jobs with Amazon SageMaker Model Building Pipelines.

Topics
- Prerequisites (p. 1412)
- Create a Pipeline (p. 1413)

Prerequisites
To run the following tutorial you must do the following:

- Set up your notebook instance as outlined in Create a notebook instance. This gives your role permissions to read and write to Amazon S3, and create training, batch transform, and processing jobs in SageMaker.
- Grant your notebook permissions to get and pass its own role as shown in Modifying a role permissions policy. Add the following JSON snippet to attach this policy to your role. Replace `<your-role-arn>` with the ARN used to create your notebook instance.

```
{
   "Version": "2012-10-17",
   "Statement": [
   {
      "Effect": "Allow",
      "Action": [
      "iam:GetRole",
      "iam:PassRole"
      ],
      "Resource": "<your-role-arn>"
   }
   ]
}
```

- Trust the SageMaker service principal by following the steps in Modifying a role trust policy. Add the following statement fragment to the trust relationship of your role:

```
{
   "Sid": ",",
   "Effect": "Allow",
   "Principal": {
      "Service": "sagemaker.amazonaws.com"
   },
   "Action": "sts:AssumeRole"
}
```

Set Up Your Environment
Create a new SageMaker session using the following code block. This returns the role ARN for the session. This role ARN should be the execution role ARN that you set up as a prerequisite.

```python
import boto3
import sagemaker
import sagemaker.session

region = boto3.Session().region_name
sagemaker_session = sagemaker.session.Session()
role = sagemaker.session.get_execution_role()
```
default_bucket = sagemaker_session.default_bucket()
model_package_group_name = f"AbaloneModelPackageGroupName"

Create a Pipeline

Run the following steps from your SageMaker notebook instance to create a pipeline including steps for preprocessing, training, evaluation, conditional evaluation, and model registration.

**Step 1: Download the Dataset**

This notebook uses the UCI Machine Learning Abalone Dataset. The dataset contains the following features:

- **length** – The longest shell measurement of the abalone.
- **diameter** – The diameter of the abalone perpendicular to its length.
- **height** – The height of the abalone with meat in the shell.
- **whole_weight** – The weight of the whole abalone.
- **shucked_weight** – The weight of the meat removed from the abalone.
- **viscera_weight** – The weight of the abalone viscera after bleeding.
- **shell_weight** – The weight of the abalone shell after meat removal and drying.
- **sex** – The sex of the abalone. One of 'M', 'F', or 'I', where 'I' is an infant abalone.
- **rings** – The number of rings in the abalone shell.

The number of rings in the abalone shell is a good approximation for its age using the formula \( \text{age} = \text{rings} + 1.5 \). However, obtaining this number is a time-consuming task. You must cut the shell through the cone, stain the section, and count the number of rings through a microscope. However, the other physical measurements are easier to determine. This notebook uses the dataset to build a predictive model of the variable rings using the other physical measurements.

**To download the dataset**

1. Download the dataset into your account's default Amazon S3 bucket.

   ```
   !mkdir -p data
   local_path = "data/abalone-dataset.csv"

   s3 = boto3.resource("s3")
   s3.Bucket(f"sagemaker-servicecatalog-seedcode-{region}").download_file(
       "dataset/abalone-dataset.csv",
       local_path
   )

   base_uri = f"s3://{default_bucket}/abalone"
   input_data_uri = sagemaker.s3.S3Uploader.upload(
       local_path=local_path,
       desired_s3_uri=base_uri,
   )
   print(input_data_uri)
   ```

2. Download a second dataset for batch transformation after your model is created.

   ```
   local_path = "data/abalone-dataset-batch"

   s3 = boto3.resource("s3")
   s3.Bucket(f"sagemaker-servicecatalog-seedcode-{region}").download_file(
       "dataset/abalone-dataset-batch",
       local_path
   )
   ```
Step 2: Define Pipeline Parameters

This code block defines the following parameters for your pipeline:

- `processing_instance_type` – The ml.* instance type of the processing jobs.
- `processing_instance_count` – The instance count of the processing job.
- `training_instance_type` – The ml.* instance type of the training jobs.
- `input_data` – The Amazon S3 location of the input data.
- `batch_data` – The Amazon S3 location of the input data for batch transformation.
- `model_approval_status` – The approval status to register the trained model with for CI/CD. For more information, see Automate MLOps with SageMaker Projects (p. 1439).

```python
from sagemaker.workflow.parameters import (ParameterInteger,
                                           ParameterString,
)

processing_instance_count = ParameterInteger(
    name="ProcessingInstanceCount",
    default_value=1
)

processing_instance_type = ParameterString(
    name="ProcessingInstanceType",
    default_value="ml.m5.xlarge"
)

training_instance_type = ParameterString(
    name="TrainingInstanceType",
    default_value="ml.m5.xlarge"
)

model_approval_status = ParameterString(
    name="ModelApprovalStatus",
    default_value="PendingManualApproval"
)

input_data = ParameterString(
    name="InputData",
    default_value=input_data_uri,
)

batch_data = ParameterString(
    name="BatchData",
    default_value=batch_data_uri,
)
```

Step 3: Define a Processing Step for Feature Engineering

This section shows how to create a processing step to prepare the data from the dataset for training.

To create a processing step

1. Create a directory for the processing script.
2. Create a file in the /abalone directory named preprocessing.py with the following content. This preprocessing script is passed in to the processing step for execution on the input data. The training step then uses the preprocessed training features and labels to train a model, and the evaluation step uses the trained model and preprocessed test features and labels to evaluate the model. The script uses scikit-learn to do the following:

- Fill in missing sex categorical data and encode it so it's suitable for training.
- Scale and normalize all numerical fields except for rings and sex.
- Split the data into training, test, and validation datasets.

```python
# Because this is a headerless CSV file, specify the column names here.
feature_columns_names = [
    "sex",
    "length",
    "diameter",
    "height",
    "whole_weight",
    "shucked_weight",
    "viscera_weight",
    "shell_weight",
]
label_column = "rings"

feature_columns_dtype = {
    "sex": str,
    "length": np.float64,
    "diameter": np.float64,
    "height": np.float64,
    "whole_weight": np.float64,
    "shucked_weight": np.float64,
    "viscera_weight": np.float64,
    "shell_weight": np.float64
}
label_column_dtype = {"rings": np.float64}

def merge_two_dicts(x, y):
    z = x.copy()
    z.update(y)
    return z

if __name__ == "__main__":
    base_dir = "/opt/ml/processing"
```
df = pd.read_csv(f"{base_dir}/input/abalone-dataset.csv", header=None, names=feature_columns_names + [label_column], dtype=merge_two_dicts(feature_columns_dtype, label_column_dtype))

numeric_features = list(feature_columns_names)
numeric_features.remove("sex")

numeric_transformer = Pipeline(
    steps=[
        ("imputer", SimpleImputer(strategy="median")),
        ("scaler", StandardScaler())
    ]
)

categorical_features = ["sex"]
categorical_transformer = Pipeline(
    steps=[
        ("imputer", SimpleImputer(strategy="constant", fill_value="missing")),
        ("onehot", OneHotEncoder(handle_unknown="ignore"))
    ]
)

preprocess = ColumnTransformer(
    transformers=[
        ("num", numeric_transformer, numeric_features),
        ("cat", categorical_transformer, categorical_features)
    ]
)

y = df.pop("rings")
X_pre = preprocess.fit_transform(df)
y_pre = y.to_numpy().reshape(len(y), 1)

X = np.concatenate((y_pre, X_pre), axis=1)

np.random.shuffle(X)

train, validation, test = np.split(X, [int(.7*len(X)), int(.85*len(X))])

pd.DataFrame(train).to_csv(f"{base_dir}/train/train.csv", header=False, index=False)
pd.DataFrame(validation).to_csv(f"{base_dir}/validation/validation.csv", header=False, index=False)
pd.DataFrame(test).to_csv(f"{base_dir}/test/test.csv", header=False, index=False)

3. Create an instance of an SKLearnProcessor to pass in to the processing step.

from sagemaker.sklearn.processing import SKLearnProcessor

framework_version = "0.23-1"
sklearn_processor = SKLearnProcessor(
    framework_version=framework_version,
    instance_type=processing_instance_type,
    instance_count=processing_instance_count,
    base_job_name="sklearn-abalone-process",
    role=role,
)

4. Create a processing step. This step takes in the SKLearnProcessor, the input and output channels, and the preprocessing.py script that you created. This is very similar to a processor
instance's `run` method in the SageMaker Python SDK. The `input_data` parameter passed into `ProcessingStep` is the input data of the step itself. This input data is used by the processor instance when it runs.

Note the "train," "validation," and "test" named channels specified in the output configuration for the processing job. Step Properties such as these can be used in subsequent steps and resolve to their runtime values at execution.

```python
from sagemaker.processing import ProcessingInput, ProcessingOutput
from sagemaker.workflow.steps import ProcessingStep

step_process = ProcessingStep(
    name="AbaloneProcess",
    processor=sklearn_processor,
    inputs=[
        ProcessingInput(source=input_data, destination="/opt/ml/processing/input"),
    ],
    outputs=[
        ProcessingOutput(output_name="train", source="/opt/ml/processing/train"),
        ProcessingOutput(output_name="validation", source="/opt/ml/processing/validation"),
        ProcessingOutput(output_name="test", source="/opt/ml/processing/test")
    ],
    code="abalone/preprocessing.py",
)
```

**Step 4: Define a Training step**

This section shows how to use the SageMaker XGBoost Algorithm to train a logistic regression model on the training data output from the processing steps.

**To define a training step**

1. Specify the model path where you want to save the models from training.

   ```python
   model_path = f"s3://{default_bucket}/AbaloneTrain"
   ```

2. Configure an estimator for the XGBoost algorithm and the input dataset. The `training_instance_type` is passed into the estimator. A typical training script loads data from the input channels, configures training with hyperparameters, trains a model, and saves a model to `model_dir` so that it can be hosted later. SageMaker uploads the model to Amazon S3 in the form of a `model.tar.gz` at the end of the training job.

   ```python
   from sagemaker.estimator import Estimator

   image_uri = sagemaker.image_uris.retrieve(
       framework="xgboost",
       region=region,
       version="1.0-1",
       py_version="py3",
       instance_type=training_instance_type,
   )
   xgb_train = Estimator(
       image_uri=image_uri,
       instance_type=training_instance_type,
       instance_count=1,
       output_path=model_path,
       role=role,
   )
   ```
3. Create a TrainingStep using the estimator instance and properties of the ProcessingStep. In particular, pass in the S3Uri of the "train" and "validation" output channel to the TrainingStep.

```python
from sagemaker.inputs import TrainingInput
from sagemaker.workflow.steps import TrainingStep

step_train = TrainingStep(
    name="AbaloneTrain",
    estimator=xgb_train,
    inputs={
        "train": TrainingInput(
            s3_data=step_process.properties.ProcessingOutputConfig.Outputs["train"].S3Output.S3Uri,
            content_type="text/csv"
        ),
        "validation": TrainingInput(
            s3_data=step_process.properties.ProcessingOutputConfig.Outputs["validation"].S3Output.S3Uri,
            content_type="text/csv"
        )
    },
)
```

**Step 5: Define a Processing Step for Model Evaluation**

This section shows how to create a processing step to evaluate the accuracy of the model. The result of this model evaluation is used in the condition step to determine which execute path to take.

**To define a processing step for model evaluation**

1. Create a file in the /abalone directory named evaluation.py. This script is used in a processing step to perform model evaluation. It takes a trained model and the test dataset as input, then produces a JSON file containing classification evaluation metrics. These metrics include a precision, recall, and F1 score for each label, and accuracy and ROC curve for the model.

```python
%%writefile abalone/evaluation.py
import json
import pathlib
import pickle
import tarfile
import joblib
import numpy as np
import pandas as pd
import xgboost
```
from sklearn.metrics import mean_squared_error

if __name__ == "__main__":
    model_path = f"/opt/ml/processing/model/model.tar.gz"
    with tarfile.open(model_path) as tar:
        tar.extractall(path="."

    model = pickle.load(open("xgboost-model", "rb"))

    test_path = "/opt/ml/processing/test/test.csv"
    df = pd.read_csv(test_path, header=None)
    y_test = df.iloc[:, 0].to_numpy()
    df.drop(df.columns[0], axis=1, inplace=True)
    X_test = xgboost.DMatrix(df.values)

    predictions = model.predict(X_test)
    mse = mean_squared_error(y_test, predictions)
    std = np.std(y_test - predictions)

    report_dict = {
        "regression_metrics": {
            "mse": {
                "value": mse,
                "standard_deviation": std
            }
        }
    }

    output_dir = "/opt/ml/processing/evaluation"
    pathlib.Path(output_dir).mkdir(parents=True, exist_ok=True)

    evaluation_path = f"{output_dir}/evaluation.json"
    with open(evaluation_path, "w") as f:
        f.write(json.dumps(report_dict))

2. Create an instance of a ScriptProcessor that is used to create a ProcessingStep.

```python
from sagemaker.processing import ScriptProcessor

script_eval = ScriptProcessor(
    image_uri=image_uri,
    command=["python3"],
    instance_type=processing_instance_type,
    instance_count=1,
    base_job_name="script-abalone-eval",
    role=role,
)
```

3. Create a ProcessingStep using the processor instance, the input and output channels, and the evaluation.py script. In particular, pass in the S3ModelArtifacts property from the step_train training step, as well as the S3Uri of the "test" output channel of the step_process processing step. This is very similar to a processor instance's run method in the SageMaker Python SDK.

```python
from sagemaker.workflow.properties import PropertyFile

evaluation_report = PropertyFile(
    name="EvaluationReport",
    output_name="evaluation",
)```
Step 6: Define a CreateModelStep for Batch Transformation

This section shows how to create a SageMaker model from the output of the training step. This model is used for batch transformation on a new dataset. This step is passed into the condition step and only executes if the condition step evaluates to `true`.

To define a CreateModelStep for batch transformation


   ```python
   from sagemaker.model import Model
   
   model = Model(
       image_uri=image_uri,
       model_data=step_train.properties.ModelArtifacts.S3ModelArtifacts,
       sagemaker_session=sagemaker_session,
       role=role,
   )
   ```

2. Define the model input for your SageMaker model.

   ```python
   from sagemaker.inputs import CreateModelInput
   
   inputs = CreateModelInput(
       instance_type="ml.m5.large",
       accelerator_type="ml.eia1.medium",
   )
   ```

3. Create your CreateModelStep using the CreateModelInput and SageMaker model instance you defined.

   ```python
   from sagemaker.workflow.steps import CreateModelStep
   ```
Step 7: Define a TransformStep to Perform Batch Transformation

This section shows how to create a TransformStep to perform batch transformation on a dataset after the model is trained. This step is passed into the condition step and only executes if the condition step evaluates to true.

To define a TransformStep to perform batch transformation

1. Create a transformer instance with the appropriate compute instance type, instance count, and desired output Amazon S3 bucket URI. Pass in the ModelName property from the step_create_model CreateModel step.

   from sagemaker.transformer import Transformer
   transformer = Transformer(
       model_name=step_create_model.properties.ModelName,
       instance_type="ml.m5.xlarge",
       instance_count=1,
       output_path="s3://{default_bucket}/AbaloneTransform"
   )

2. Create a TransformStep using the transformer instance you defined and the batch_data pipeline parameter.

   from sagemaker.inputs import TransformInput
   from sagemaker.workflow.steps import TransformStep

   step_transform = TransformStep(
       name="AbaloneTransform",
       transformer=transformer,
       inputs=TransformInput(data=batch_data)
   )

Step 8: Define a RegisterModel Step to Create a Model Package

This section shows how to construct an instance of RegisterModel. The result of executing RegisterModel in a pipeline is a model package. A model package is a reusable model artifacts abstraction that packages all ingredients necessary for inference. It consists of an inference specification that defines the inference image to use along with an optional model weights location. A model package group is a collection of model packages. You can use a ModelPackageGroup for SageMaker Pipelines to add a new version and model package to the group for every pipeline execution. For more information about model registry, see Register and Deploy Models with Model Registry (p. 1136).

This step is passed into the condition step and only executes if the condition step evaluates to true.

To define a RegisterModel step to create a model package

- Construct a RegisterModel step using the estimator instance you used for the training step. Pass in the S3ModelArtifacts property from the step_train training step and specify a ModelPackageGroup. SageMaker Pipelines creates this ModelPackageGroup for you.
Step 9: Define a Condition Step to Verify Model Accuracy

A ConditionStep allows SageMaker Pipelines to support conditional execution in your pipeline DAG based on the condition of step properties. In this case, you only want to register a model package if the accuracy of that model, as determined by the model evaluation step, exceeds the required value. If the accuracy exceeds the required value, the pipeline also creates a SageMaker Model and runs batch transformation on a dataset. This section shows how to define the Condition step.

To define a condition step to verify model accuracy

1. Define a `ConditionLessThanOrEqualTo` condition using the accuracy value found in the output of the model evaluation processing step, `step_eval`. Get this output using the property file you indexed in the processing step and the respective JSONPath of the mean squared error value, "mse".

```
from sagemaker.workflow.conditions import ConditionLessThanOrEqualTo
from sagemaker.workflow.condition_step import (ConditionStep, JsonGet,
)
cond_lte = ConditionLessThanOrEqualTo(
    left=JsonGet(
        step=step_eval,
        property_file=evaluation_report,
        json_path="regression_metrics.mse.value"
    ),
    right=6.0
)
```

2. Construct a ConditionStep. Pass the `ConditionEquals` condition in, then set the model package registration and batch transformation steps as the next steps if the condition passes.
Step 10: Create a pipeline

Now that you've created all of the steps, combine them into a pipeline.

To create a pipeline

1. Define the following for your pipeline: name, parameters, and steps. Names must be unique within an (account, region) pair.

   Note
   A step can only appear once in either the pipeline's step list or the if/else step lists of the condition step. It cannot appear in both.

   ```python
   from sagemaker.workflow.pipeline import Pipeline

   pipeline_name = f"AbalonePipeline"
   pipeline = Pipeline(
       name=pipeline_name,
       parameters=[
           processing_instance_type,
           processing_instance_count,
           training_instance_type,
           model_approval_status,
           input_data,
           batch_data,
       ],
       steps=[step_process, step_train, step_eval, step_cond],
   )
   ```

2. (Optional) Examine the JSON pipeline definition to ensure that it's well-formed.

   ```python
   import json
   json.loads(pipeline.definition())
   ```

This pipeline definition is ready to submit to SageMaker. In the next tutorial, you submit this pipeline to SageMaker and start an execution.

Next step: Run a pipeline (p. 1423)

Run a pipeline

After you've created a pipeline definition using the SageMaker Python SDK, you can submit it to SageMaker to start your execution. The following tutorial shows how to submit a pipeline, start an execution, examine the results of that execution, and delete your pipeline.

Topics
- Prerequisites (p. 1424)
- Step 1: Start the Pipeline (p. 1424)
• Step 2: Examine a Pipeline Execution (p. 1424)
• Step 3: Override Default Parameters for a Pipeline Execution (p. 1426)
• Step 4: Stop and Delete a Pipeline Execution (p. 1426)

Prerequisites

This tutorial requires the following:

• A SageMaker notebook instance.
• A SageMaker Pipelines pipeline definition. This tutorial assumes you're using the pipeline definition created by completing the Define a Pipeline (p. 1411) tutorial.

Step 1: Start the Pipeline

First, you need to start the pipeline.

To start the pipeline

1. Examine the JSON pipeline definition to ensure that it's well-formed.

   ```python
   import json
   json.loads(pipeline.definition())
   ```

2. Submit the pipeline definition to the SageMaker Pipelines service to create a pipeline if it doesn't exist, or update the pipeline if it does. The role passed in is used by SageMaker Pipelines to create all of the jobs defined in the steps.

   ```python
   pipeline.upsert(role_arn=role)
   ```

3. Start a pipeline execution.

   ```python
   execution = pipeline.start()
   ```

Step 2: Examine a Pipeline Execution

Next, you need to examine the pipeline execution.

To examine a pipeline execution

1. Describe the pipeline execution status to ensure that it has been created and started successfully.

   ```python
   execution.describe()
   ```

2. Wait for the execution to finish.

   ```python
   execution.wait()
   ```

3. List the execution steps and their status.

   ```python
   execution.list_steps()
   ```

Your output should look like the following:
4. After your pipeline execution is complete, download the resulting evaluation.json file from Amazon S3 to examine the report.

```python
evaluation_json = sagemaker.s3.S3Downloader.read_file("/evaluation.json").format(step_eval.arguments["ProcessingOutputConfig"])["Outputs"][0]["S3Output"]
```
json.loads(evaluation_json)
Step 3: Override Default Parameters for a Pipeline Execution

You can run additional executions of the pipeline by specifying different pipeline parameters to override the defaults.

To override default parameters

1. Create the pipeline execution. This starts another pipeline execution on a compute-optimized instance type and sets the model approval status to "Approved". This means that the model package version generated by the `RegisterModel` step is automatically ready for deployment through CI/CD pipelines, such as with SageMaker Projects. For more information, see Automate MLOps with SageMaker Projects (p. 1439).

   ```python
   execution = pipeline.start(
       parameters=dict(
           ProcessingInstanceType="ml.c5.xlarge",
           ModelApprovalStatus="Approved",
       )
   )
   ```

2. Wait for the execution to finish.

   ```python
   execution.wait()
   ```

3. List the execution steps and their status.

   ```python
   execution.list_steps()
   ```

4. After your pipeline execution is complete, download the resulting `evaluation.json` file from Amazon S3 to examine the report.

   ```python
   evaluation_json = sagemaker.s3.S3Downloader.read_file("{}/evaluation.json".format(
       step_eval.arguments["ProcessingOutputConfig"]["Outputs"][0]["S3Output"]["S3Uri"]
   ))
   json.loads(evaluation_json)
   ```

Step 4: Stop and Delete a Pipeline Execution

When you're finished with your pipeline, you can stop any ongoing executions and delete the pipeline.

To stop and delete a pipeline execution

1. Stop the pipeline execution.

   ```python
   execution.stop()
   ```

2. Delete the pipeline.

   ```python
   pipeline.delete()
   ```

View, Track, and Execute SageMaker Pipelines in SageMaker Studio

To view, track, and execute Amazon SageMaker Pipelines in Amazon SageMaker Studio, you must sign in to Studio. For more information, see Onboard to Amazon SageMaker Studio (p. 34).
View a Pipeline

This procedure shows you how to directly find a pipeline and view its details page. Pipelines that are part of a project can also be found listed in the project's details page. For information on finding a pipeline that is part of a project, see Automate MLOps with SageMaker Projects (p. 1439).

To view a list of pipelines

1. In the left sidebar of Studio, choose the Components and registries icon (צ"א).

2. In the drop-down menu, select Pipelines.
3. Drag the right border of the **Components and registries** pane to the right to view all the columns. Use search to narrow the list of pipelines. Search is a two-step process. First, you enter characters that match the column name you want to search. Second, you enter characters to match the item in that column. You can have multiple search filters. An an example, the following screenshot limits the displayed pipelines to those with a name that starts with "aba" and created by Me. For more information on searching in SageMaker Studio, see Search Experiments Using Amazon SageMaker Studio (p. 859).

![Components and registries pane](image)

4. Double-click a pipeline to view details about the pipeline. The pipeline details tab opens and displays a list of pipeline executions. You can start an execution or choose one of the other tabs for more information about the pipeline. Use the **Settings** icon (🗂️) on the top right of the screen to choose which columns to display.

![Pipeline details page](image)

5. From the pipeline details page, choose one of the following tabs to view details about the pipeline:
   - **Executions** – Details about the executions. You can start an execution from this tab or the **Graph** tab.
   - **Graph** – The DAG for the pipeline.
• **Parameters** - Includes the model approval status.

```
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ProcessingInstanceType</td>
<td>String</td>
<td>ml.m5.xlarge</td>
</tr>
<tr>
<td>ProcessingInstanceCount</td>
<td>Integer</td>
<td>1</td>
</tr>
<tr>
<td>TrainingInstanceType</td>
<td>String</td>
<td>ml.m5.xlarge</td>
</tr>
<tr>
<td>ModelApprovalStatus</td>
<td>String</td>
<td>PendingManualApproval</td>
</tr>
<tr>
<td>InputData</td>
<td>String</td>
<td><a href="https://s3-us-west-2.amazonaws">https://s3-us-west-2.amazonaws</a>...</td>
</tr>
</tbody>
</table>
```

• **Settings** – The metadata associated with the pipeline. You can download the pipeline definition file, and edit the pipeline name and description from this tab.
View a Pipeline Execution

This procedure shows you how to view a pipeline execution. For information on how to view a list of pipeline executions, and how to use SageMaker search to narrow the executions in the list, see View a Pipeline (p. 1427).

To view details of a pipeline execution

1. In the execution list, double-click an execution to view details about the execution. The execution details tab opens and displays a graph of the steps in the pipeline. You can choose one of the other tabs to view information about the pipeline execution, which is similar to that shown when viewing the pipeline details.

2. Use search to find a step in the graph. Type characters that match a step name. You can drag the graph around or use the resizing icons on the lower-left side of the graph. The inset on the lower-right side of the graph displays where you are in the graph.
3. Choose one of the steps in the graph to see details about the step. The following tabs are displayed:

- **Output** – The metrics, files, and evaluation outcome of the step, dependent on the step type.
• **Logs** – The Amazon CloudWatch logs produced by the step.

• **Info** – The parameters and metadata associated with the step.
Execute a Pipeline

This procedure shows you how to execute a pipeline. For information on how to view a list of pipeline executions, see View a Pipeline (p. 1427).

To start a pipeline execution

1. From the Executions or Graph tab in the execution list, choose Start an execution.
2. Enter or update the following required information:
   - **Name** – Must be unique to your account in the AWS Region.
   - **ProcessingInstanceType** – The instance type for processing.
   - **ProcessingInstanceCount** – The number of instances to use for processing.
   - **TrainingInstanceType** – The instance type for training.
   - **ModelApprovalStatus** – For your convenience.
   - **InputData** – The S3 URI of the input data.

3. Choose **Submit**.

4. To see details of the execution or to stop the execution, choose **View details** on the status banner.

5. To stop the execution, choose **Stop** on the status banner.
The view the list of registered models, see Automate MLOps with SageMaker Projects (p. 1439).

**Track the Lineage of a SageMaker ML Pipeline**

In this tutorial, you use Amazon SageMaker Studio to track the lineage of an Amazon SageMaker ML Pipeline.

The pipeline was created by the Orchestrating Jobs with Amazon SageMaker Model Building Pipelines notebook in the Amazon SageMaker example GitHub repository. For detailed information on how the pipeline was created, see Define a Pipeline (p. 1411).

Lineage tracking in Studio is centered around a directed acyclic graph (DAG). The DAG represents the steps in a pipeline. From the DAG you can track the lineage from any step to any other step. The following diagram displays the steps in the pipeline. These steps appear as a DAG in Studio.

---

**Prerequisites**

- Access to Amazon SageMaker Studio. For more information, see Onboard to Amazon SageMaker Studio (p. 34).
- Familiarity with the SageMaker Studio user interface. For more information, see Amazon SageMaker Studio UI Overview (p. 79).
- (Recommended) A completed run of the example notebook.

**To track the lineage of a pipeline**

1. Sign in to SageMaker Studio.
2. In the left sidebar of Studio, choose the SageMaker Components and registries icon ( ).
3. In the drop-down menu, select Pipelines.
4. Use the **Search** box to filter the pipelines list. To view all available columns, drag the right border of the pane to the right. For more information, see [Search Experiments Using Amazon SageMaker Studio](p. 859).

The following screenshot shows the list filtered by a name that starts with “aba” and that was created on 12/5/20.
5. Double-click the AbalonePipeline pipeline to view the execution list and other details about the pipeline. The following screenshot shows the TABLE PROPERTIES pane open where you can choose which properties to view.

6. Choose the Settings tab and then choose Download pipeline definition file. You can view the file to see how the pipeline graph was defined.

7. On the Execution tab, double-click the first row in the execution list to view its execution graph and other details about the execution. Note that the graph matches the diagram displayed at the beginning of the tutorial.

    You can drag the graph around (select an area not on the graph itself) or use the resizing icons on the lower-left side of the graph. The inset on the lower-right side of the graph displays your location in the graph.
8. On the Graph tab, choose the AbaloneProcess step to view details about the step.

9. Find the Amazon S3 paths to the training, validation, and test datasets in the Output tab, under Files.

   **Note**
   To get the full paths, right-click the path and then choose Copy cell contents.
10. Choose the AbaloneTrain step.

11. Find the Amazon S3 path to the model artifact in the Output tab, under Files:

s3://sagemaker-eu-west-1-acct-id/AbaloneTrain/pipelines-6locnsqz4bfu-AbaloneTrain-NtfEpI0Ahu/output/model.tar.gz

12. Choose the AbaloneRegisterModel step.

13. Find the ARN of the model package in the Output tab, under Files:

arn:aws:sagemaker:eu-west-1:acct-id:model-package/abalonemodelpackagegroupname/2

Automate MLOps with SageMaker Projects

Create end-to-end ML solutions with CI/CD by using SageMaker projects.

Use SageMaker projects to create an ML ops solution to orchestrate and manage:

- Data preparation and feature engineering
- Training models
- Evaluating models
- Deploying models
What is a SageMaker Project?

By using a SageMaker project, teams of data scientists and developers can work in machine learning business problems. You can create a SageMaker project with a SageMaker provided MLOps template that automates the model building and deployment pipelines using continuous integrations and continuous delivery (CI/CD). A SageMaker provided template provisions the initial setup required for a complete end-to-end MLOps system including model building, training, and deployment, depending on the template you choose. You can also provide your own custom template to provision the resources you need for your MLOps system.

A SageMaker project is an AWS Service Catalog provisioned product that enables you to easily create an end-to-end ML solution. For information about AWS Service Catalog, see What is AWS Service Catalog.

Each SageMaker project has a unique name and ID that are passed to all SageMaker and AWS resources created in the project. By using the name and ID, you can view all entities associated with your project. These include:

- Pipeline executions
- Registered models
- Deployed models (endpoints)
- Datasets
- AWS Service Catalog Products
- AWS CodePipeline pipelines
- AWS CodeCommit repositories

A typical SageMaker project with a CICD template might include the following.

- One or more CodeCommit repositories with sample code for building and deploying ML solutions. These are working examples that you can clone locally to explore the code provided by SageMaker and modify for your needs. You own this code and you can use the repositories as version control for your work.
- A SageMaker pipeline that defines steps for data preparation, training, model evaluation, and model deployment.
- A CodePipeline that runs your SageMaker pipeline every time a new version of the code is checked in. For information about CodePipeline, see What is AWS AWS CodePipeline.
- A model group that contains model versions. Each time the SageMaker pipeline runs, and the resulting model version is accepted in the conditional validation step, a new model version is deployed to a SageMaker endpoint.
When Should You Use a SageMaker Project?

While notebooks are helpful for model building and experimentation, when you have a team of data scientists and/or ML engineers working on an ML problem, you need a more scalable way to maintain code consistency and have stricter version control. Having the code only in notebook files makes it harder to collaborate and risks losing code or model artifacts if the notebook is accidentally deleted or changed. By using SageMaker projects, you can manage the versions for your Git repositories so you can collaborate across teams more efficiently, ensure code consistency, and enable CI/CD.

In addition to managing code, SageMaker projects enable MLOps for model building, model deployment, and end-to-end ML workflows. You can run training jobs or SageMaker pipelines to build models in SageMaker Studio. However, if you want to create a CI/CD system that generates models based on triggers, such as when someone checks in a code change, then consider creating a SageMaker project and using a SageMaker provided template. For a list of the project templates that SageMaker provides, see Use SageMaker Provided Project Templates (p. 1448).

Do I Need to Create a Project to Use SageMaker Pipelines?

No. SageMaker pipelines are standalone entities just like training jobs, processing jobs, and other SageMaker jobs within SageMaker. You can create, update, and run pipelines without directly within a notebook by using the SageMaker Python SDK without using a SageMaker project.

Projects provide an additional layer to help you organize your code and adopt operational best practices that you need for a production-quality system.

Why Should You Use MLOps?

As you move from running individual artificial intelligence and machine learning (AI/ML) projects to using AI/ML to transform your business at scale, the discipline of ML Operations (MLOps) can help. MLOps accounts for the unique aspects of AI/ML projects in project management, CI/CD, and quality assurance, helping customers improve delivery time, reduce defects, and make data scientists more productive. MLOps refers to a methodology that is built on applying DevOps practices to machine learning workloads. You can review the Introduction to DevOps on AWS white paper for a discussion of DevOps principles. Practicing CI/CD on AWS and Infrastructure as Code go deeper into implementation using AWS services.

Like DevOps, MLOps relies on a collaborative and streamlined approach to the machine learning development lifecycle where the intersection of people, process and technology are required to optimize the end-to-end activities required to develop, build, and operate machine learning workloads.

MLOps focuses on the intersection of data science and data engineering in combination with existing DevOps practices to streamline model delivery across the machine learning development lifecycle. MLOps is the discipline of integrating ML workloads into release management, CI/CD, and operations. MLOps requires the integration of software development, operations, data engineering, and data science.

Challenges with MLOps

- **Project management** - ML projects involve data scientists, a relatively new role, and one not often integrated into cross-functional teams. These new team members often speak a very different technical language than product owners and software engineers, compounding the usual problem of translating business requirements into technical requirements.

- **Communication and collaboration** - Building visibility on ML projects and enabling collaboration across different stakeholders such as data engineers, data scientists, ML engineers, and DevOps.

- **Everything is code**
Why Should You Use MLOps?

- Use of production data in development activities, longer experimentation lifecycles, dependencies on data pipelines, retraining deployment pipelines, and unique metrics in evaluating the performance of a model.
- Models often have a lifecycle independent of the applications and systems integrating with those models.
- The entire end-to-end system is reproducible through versioned code and artifacts. DevOps projects use Infrastructure-as-Code(IaC) and Configuration-as-Code(CaC) to build environments, and Pipelines-as-code to ensure consistent CI/CD patterns. The pipelines have to integrate with Big Data and ML training workflows. That often means that our pipeline is a combination of a traditional CI/CD tool and another workflow engine. There are important policy concerns for many ML projects, so our pipeline may also need to enforce those policies. Biased input data produces biased results, an increasing concern for business stakeholders.

- **CI/CD**
  - In MLOps, the source data is a first-class input, along with source code. That's why MLOps calls for versioning the source data and triggering pipeline runs when the source or inference data changes.
  - Pipelines must also version the ML models along with their inputs and other outputs, in order to provide for traceability.
  - Automated testing must include proper validation of the ML model, during build phases and when the model is in production.
  - Build phases may include model training and retraining, a time-consuming and resource-intensive process. Pipelines must be granular enough to only perform a full training cycle when the source data or ML code changes, not when related components change.
  - Because machine learning code is typically a small part of an overall solution, a deployment pipeline may also incorporate the additional steps required to package your model for consumption as an API by other applications and systems.

- **Monitoring and logging**
  - In the feature engineering and model training phases, we need to capture model training metrics as well as model experiments. Tuning an ML model requires manipulating the form of the input data as well as algorithm hyperparameters, and we need to systematically capture those experiments. Experiment tracking helps data scientists work more effectively and gives us a reproducible snapshot of their work.
  - Deployed ML models require monitoring of the data passed to the model for inference, along with the standard endpoint stability and performance metrics. The monitoring system must also capture the quality of model output, as evaluated by an appropriate ML metric.

Benefits of MLOps

Adopting MLOps practices gives you faster time-to-market for ML projects by delivering the following benefits.

- **Productivity** - Providing self-service environments with access to curated data sets lets data engineers and data scientists move faster and waste less time with missing or invalid data.
- **Repeatability** - Automating all the steps in the MLDC helps you ensure a repeatable process, including how the model is trained, evaluated, versioned, and deployed.
- **Reliability** - Incorporating CI/CD practices allows for the ability to not only deploy quickly but with increased quality and consistency.
- **Auditability** - Versioning all inputs and outputs, from data science experiments to source data to trained model, means that we can demonstrate exactly how the model was built, and where it was deployed.
- **Data and model quality** - MLOps lets us enforce policies that guard against model bias and track changes to data statistical properties and model quality over time.
SageMaker Studio Permissions Required to Use Projects

To enable users to view SageMaker provided project templates and create projects with those templates, enable **Projects** permissions for users when you onboard or update SageMaker Studio. There are 2 permissions to enable.

1. Enable **Projects** permissions for the SageMaker Studio administrator. This enables the SageMaker Studio administrator to view the SageMaker provided templates in the AWS Service Catalog console. The administrator can see what other SageMaker Studio users create if you grant them permission to use SageMaker projects. The administrator can also view the AWS CloudFormation template that the SageMaker provided project templates define in the Service Catalog Console. For information about using the Service Catalog console, see [*What Is AWS Service Catalog*](https://docs.aws.amazon.com/servicecatalog/latest/adminguide/what-is-servicecatalog.html) in the *AWS Service Catalog User Guide*.

2. Enable projects for SageMaker studio users who are configured to use the domain execution role. This grants permission for SageMaker Studio users the ability to use the SageMaker provided project templates to create a project from within SageMaker Studio.

To enable users who use any role other than the domain execution role to view and use SageMaker provided project templates, you need to enable **Projects** permissions on the individual user profiles.

The following procedures show how to enable **Projects** permissions when you are onboarding to SageMaker Studio. For more information about onboarding to SageMaker Studio, see [*Onboard to Amazon SageMaker Studio*](https://docs.aws.amazon.com/sagemaker/latest/dg/whats-new.html) (p. 34).

**To enable Projects permissions for the administrator and domain execution role users**

1. Open the *SageMaker console*.
2. Choose *Amazon SageMaker Studio* at the top left of the page.
3. If you choose **Quick start**, under **Projects**, make sure **Enable SageMaker project templates for this account and Studio users** is enabled.
4. If you choose **Standard setup**, under **Projects**, make sure both **Enable SageMaker project templates for this account** and **Enable SageMaker project templates for Studio users** are enabled.

**Create an MLOps Project**

Create a SageMaker MLOps project by using SageMaker studio.

**Prerequisites**

To create a project in SageMaker Studio, you need:
To create a project in SageMaker Studio

1. Sign in to Studio. For more information, see Onboard to Amazon SageMaker Studio (p. 34).

2. Choose Components and registries, and then choose Projects in the drop-down list.

3. Choose Create project.
The **Create project** tab appears.

4. For **SageMaker project templates**, choose **Organization templates** to choose a custom template that your organization created, or **SageMaker templates** to choose a SageMaker-provided template. Then choose a template from the list of available templates, and choose **Choose project template**. For more information about project templates, see MLOps Project Templates (p. 1447).
5. For **Project details** enter a name and description for your project. Optionally, add tags, which are key value pairs that you can use to track your projects. When you are finished, choose **Create project**
MLOps Project Templates

A SageMaker project template automates setting up and implementing MLOps for your projects. A SageMaker project template is an AWS Service Catalog product that SageMaker makes available to SageMaker Studio users. These Service Catalog products are visible in your Service Catalog console after enable permissions when you onboard or update SageMaker Studio. For information about enabling permissions to use SageMaker project templates, see SageMaker Studio Permissions Required to Use Projects (p. 1443). Use SageMaker project templates to create a project that is an end-to-end MLOps solution.

If you are an administrator, you can create your own custom project templates. SageMaker Studio users in your organization can use custom project templates that you create to create projects.

Topics
- Use SageMaker Provided Project Templates (p. 1448)
- Create Custom Project Templates (p. 1448)
Use SageMaker Provided Project Templates

SageMaker provides project templates that create the infrastructure you need to create an MLOps solution. Currently, SageMaker offers the following project templates.

- **MLOps template for model building and training** - This template enables you to build and train machine learning models and register models to the model registry. Use this template when you want a MLOps solution for building and training models. This template provides the following resources:
  - An AWS CodeCommit repository that contains sample code that creates SageMaker pipeline in python code and shows how to create and update a SageMaker pipeline. This repository also has a Python Jupyter notebook that you can open and run in SageMaker Studio.
  - An CodePipeline that has source and build steps. The source step points to the CodeCommit repository and the build step gets the code from that repository, creates and/or updates the SageMaker pipeline, starts a pipeline execution, and waits for the pipeline execution to complete.
  - An Amazon S3 bucket to store artifacts, including CodePipeline and CodeBuild artifacts, and any artifacts generated from the SageMaker pipeline runs.

In a collaborative environment with multiple Studio users working on a same project, we recommend creating this project. After data scientists experiment in SageMaker Studio and check heir code in to CodeCommit, model building and training happens in the common infrastructure so that there is a central, authoritative location to keep track of the ML Models and artifacts that are ready to go to production.

- **MLOps template for model deployment** - This template deploys machine learning models from the Amazon SageMaker model registry to SageMaker hosted endpoints for real-time inference. Use this template when you have trained models that you want to deploy for inference. This template provides the following resources:
  - An AWS CodeCommit repository that contains sample code that deploys models to endpoints in staging and production environments.
  - An CodePipeline that has source, build, deploy to staging, and deploy to production steps. The source step points to the CodeCommit repository, the build step gets the code from that repository, generates AWS CloudFormation stacks to deploy. The deploy to staging and deploy to production steps deploy the AWS CloudFormation stacks to their respective environments. There is a manual approval step between the staging and production build steps, so that a MLOps engineer must approve the model before it is deployed to production.

  There is also a programmatic approval step with placeholder tests in the example code in the CodeCommit repository. You can add additional tests to replace the placeholders tests.

  - An Amazon S3 bucket to store artifacts, including CodePipeline and CodeBuild artifacts, and any artifacts generated from the SageMaker pipeline runs.

This template recognizes changes in the model registry. When a new model version is registered and approved, it automatically triggers a deployment.

- **MLOps template for model building, training, and deployment** - This template enables you to easily build, train, and deploy machine learning models. Use this template when you want a complete MLOps solution from data preparation to model deployment.

  This template is a combination of the previous 2 templates, and contains all of the resources provided in those templates.

Create Custom Project Templates

If the SageMaker provided templates do not meet your needs (for example, you want to have more complex orchestration in the CodePipeline with multiple stages, custom approval steps, etc.) create your own templates.
We recommend starting by using SageMaker provided templates to understand how to organize your code and resources and build on top of it. To do this, after you enable administrator access to the SageMaker templates, login to the https://console.aws.amazon.com/servicecatalog/, choose Portfolios, then choose Imported. For information about AWS Service Catalog, see Overview of AWS Service Catalog in the AWS Service Catalog User Guide.

Create your own project templates to customize your MLOps project. SageMaker project templates are AWS Service Catalog provisioned products to provision the resources for your MLOps project.

To create a custom project template, complete the following steps.

1. Create a Portfolio. For information, see Step 3: Create an AWS Service Catalog Portfolio.
2. Create a Product. A Product is a AWS CloudFormation template. You can create multiple versions of the product. For information, see Step 4: Create an AWS Service Catalog Product.

For the Product to work with SageMaker projects, add the following parameters to your Product template.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SageMakerProjectName:</td>
<td>String</td>
<td>Name of the project</td>
</tr>
<tr>
<td>SageMakerProjectId:</td>
<td>String</td>
<td>Service generated Id of the project</td>
</tr>
</tbody>
</table>

3. Add a launch constraint. A launch constraint designates an IAM role that Service Catalog assumes when a user launches a product. For information, see Step 6: Add a Launch Constraint to Assign an IAM Role.
4. Provision the product on the https://console.aws.amazon.com/servicecatalog/ to test the template. If you are satisfied with your template, continue to the next step to make the template available in SageMaker Studio.
5. Grant access to the Service Catalog Portfolio that you created in step 1 to your SageMaker Studio execution role. Use either the SageMaker Studio domain execution role or a user role that has SageMaker Studio access. For information about adding a role to the Portfolio, see Step 7: Grant End Users Access to the Portfolio.
6. To make your project template available in your Organization templates list in SageMaker Studio, create a tag with the following key and value to the Service Catalog Product you created in step 2.
   - **key** - sagemaker:studio-visibility
   - **value** - true

After you complete these steps, SageMaker Studio users in your organization can create a project with the template you created by following the steps in Create an MLOps Project (p. 1443) and choosing Organization templates when you choose a template.

**View Project Resources**

After you create a project, view the resources associated with the project in SageMaker Studio.

**To create a project in SageMaker Studio**

1. Sign in to Studio. For more information, see Onboard to Amazon SageMaker Studio (p. 34).
2. Choose Components and registries, and then choose Projects
3. Double-click the name of the project for which you want to view details.

A tab with the project details appears.
On the product details tab, you can view the following entities associated with the project.

- **Repositories** - code repositories associated with this project. If you use a SageMaker provided template when you create your project, it creates AWS CodeCommit pipelines. For more information about CodeCommit, see [What is AWS CodeCommit](#).
- **Pipelines** - SageMaker ML pipelines that define steps to prepare data, train, and deploy models. For information about SageMaker ML pipelines, see [Create and Manage SageMaker Pipelines](#) (p. 1410).
- **Experiments** - one or more SageMaker AutoPilot experiments associated with the project. For information about
- **Model groups** - groups of model versions that were created by pipeline executions in the project. For information about model groups, see [Create a Model Group](#) (p. 1137).
- **Endpoints** - SageMaker endpoints that host deployed models for real-time inference. When a model version is approved, it is deployed to an endpoint.
- **Settings** - Settings for the project. This includes the name and description of the project.

### SageMaker MLOps Project Walkthrough

This walkthrough demonstrates how to use MLOps projects to create a CI/CD system to build, train, and deploy models.

**Prerequisites**

To complete this walkthrough, you need:

- An AWS SSO or IAM account to sign in to Studio. For information, see [Onboard to Amazon SageMaker Studio](#) (p. 34).
- Permission to use SageMaker provided project templates. For information, see [SageMaker Studio Permissions Required to Use Projects](#) (p. 1443).
• Basic familiarity with the Studio user interface. For information, see Amazon SageMaker Studio UI Overview (p. 79).

Topics
• Step 1: Create the Project (p. 1452)
• Step 2: Clone the Code Repository (p. 1455)
• Step 3: Make a Change in the Code (p. 1457)
• Step 4: Approve the Model (p. 1460)
• (Optional) Step 5: Deploy the Model Version to Production (p. 1462)
• Step 6: Cleanup Resources (p. 1463)

Step 1: Create the Project

In this step, you create a SageMaker MLOps project by using a SageMaker provided project template to build, train, and deploy models.

To create the SageMaker MLOps project

1. Sign in to Studio. For more information, see Onboard to Amazon SageMaker Studio (p. 34).
2. Choose Components and registries, and then choose Projects in the drop-down list.
3. Choose **Create project**.

The **Create project** tab appears.
4. **For SageMaker project templates**, choose **Organization templates**, then choose **MLOps template for model building, training, and deployment**.

5. **For Project details** enter a name and description for your project. **Create project**
When the project appears in the Projects list with a Status of Created, move on to the next step.

**Step 2: Clone the Code Repository**

After you create the project, two CodeCommit repositories are created in the project. One of the repositories contains code to build and train a model, and one contains code to deploy the model. In this step, you clone the repository to the local SageMaker that contains the code to build and train the model to the local SageMaker Studio environment so that you can work with the code.

**To clone the code repository**

1. Choose Components and registries, and then choose Projects in the drop-down list.
2. Find the name of the project you created in the previous step and double-click on it to open the project tab for your project.

3. In the project tab, choose Repositories, and in the Local path column for the repository that ends with modelbuild, choose clone repo...
4. In the dialog box that appears, accept the defaults and choose **Clone repository**.

When clone of the repository is complete, the local path appears in the **Local path** column. Click on the path to open the local folder that contains the repository code in SageMaker Studio.

**Step 3: Make a Change in the Code**

Now make a change to the pipeline code that builds the model and check in the change to trigger a new pipeline run. The pipeline run registers a new model version.
To make a code change

1. In SageMaker Studio, choose the file browser icon ( ), and navigate to the pipelines/abalone folder. Double-click pipeline.py to open the code file.

2. In the pipeline.py file, find the line that sets the training instance type.

```python
training_instance_type = ParameterString(
    name="TrainingInstanceType", default_value="ml.m5.xlarge"
)
```

Change `ml.m5.xlarge` to `ml.m5.large`, then type Ctrl+S to save the change.

3. Choose the Git icon ( ). Stage, commit, and push the change in pipeline.py. For information about using Git in SageMaker Studio, see Clone a Git Repository in SageMaker Studio (p. 120).
Current Repository
sagemaker-myproject-p-kjwrqqlr05qn-modelbuild

Current Branch
master

Changes

<table>
<thead>
<tr>
<th>Changes</th>
<th>History</th>
</tr>
</thead>
<tbody>
<tr>
<td>Staged</td>
<td>(1)</td>
</tr>
<tr>
<td>pipeline.py pipelines/abalone</td>
<td>M</td>
</tr>
<tr>
<td>Changed</td>
<td>(0)</td>
</tr>
<tr>
<td>Untracked</td>
<td>(4)</td>
</tr>
<tr>
<td>.sagemaker-code-config</td>
<td>U</td>
</tr>
<tr>
<td>Untitled-checkpoint.ipynb pipelines/.ipynb_checkpoints</td>
<td>U</td>
</tr>
<tr>
<td>pipeline-checkpoint.py pipelines/abalone/.ipynb_checkpoints</td>
<td>U</td>
</tr>
</tbody>
</table>

Changed instance type

Description

Commit

Git: refreshing... Python
After pushing your code change, the MLOps system triggers a run of the pipeline that creates a new model version. In the next step, you approve the new model version to deploy it to production.

**Step 4: Approve the Model**

Now you approve the new model version that was created in the previous step to trigger a deployment of the model version to a SageMaker endpoint.

**To approve the model version**

1. Choose *Components and registries*, and then choose *Projects* in the drop-down list.

2. Find the name of the project you created in the first step and double-click on it to open the project tab for your project.

3. In the project tab, choose *Model groups*, then double-click the name of the model group that appears.
The model group tab appears.

4. In the model group tab, double-click Version 2. The Version 2 tab opens. Choose Update status

5. In the model Update model version status dialog box, in the Status drop-down, choose Approve, then choose Update status.
Approving the model version causes the MLOps system to deploy the model to staging. To view the endpoint, choose the **Endpoints** tab on the project tab.

**(Optional) Step 5: Deploy the Model Version to Production**

Now you can deploy the model version to the production environment.

**Note**
To complete this step, you need to be an administrator in your SageMaker Studio domain. If you are not an administrator, skip this step.

**To deploy the model version to the production environment**

1. Log in to the CodePipeline console at [https://console.aws.amazon.com/codepipeline/](https://console.aws.amazon.com/codepipeline/)
2. Choose **Pipelines**, then choose the pipeline with the name `sagemaker-projectname-projectid-modeldeploy`, where `projectname` is the name of your project, and `projectid` is the ID of your project.
3. In the **DeployStaging** stage, choose **Review**.
4. In the Review dialog box, choose Approve.

Approving the DeployStaging stage causes the MLOps system to deploy the model to production. To view the endpoint, choose the Endpoints tab on the project tab in SageMaker Studio.

Step 6: Cleanup Resources

To stop incurring charges, clean up the resources that were created in this walkthrough. To do this, complete the following steps.

Note
To delete the AWS CloudFormation stack and the Amazon S3 bucket, you need to be an administrator in SageMaker Studio. If you are not an administrator, ask your administrator to complete those steps.

1. From the SageMaker Studio menu, choose File, choose New, and then choose Notebook.
2. In the Select Kernel dialog box, choose Python 3 (Data Science), then choose Select.
3. In the notebook, enter the following code in a cell, and then run the cell. Replace MyProject with the name of your project.

```python
import boto3
sm_client=boto3.client("sagemaker")
sm_client.delete_project(ProjectName="MyProject")
```

This deletes the Service Catalog provisioned product that the project created. This includes the CodeCommit, CodePipeline, and CodeBuild resources that were created for the project.

4. Delete the AWS CloudFormation stacks that the project created. There are 2 stacks, one for staging and one for production. The names of the stacks are sagemaker-projectname-project-id-deploy-staging and sagemaker-projectname-project-id-deploy-prod, where projectname is the name of your project, and project-id is the ID of your project.

For information about how to delete a AWS CloudFormation stack, see Deleting a stack on the AWS CloudFormation console in the AWS CloudFormation User Guide.

5. Delete the Amazon S3 bucket that the project created. The name of the bucket is sagemaker-project-project-id, where project-id is the ID of your project.

Amazon SageMaker ML Lineage Tracking

Amazon SageMaker ML Lineage Tracking creates and stores information about the steps of a machine learning (ML) workflow from data preparation to model deployment. With the tracking information you can reproduce the workflow steps, track model and dataset lineage, and establish model governance and audit standards.

With SageMaker Lineage Tracking data scientists and model builders can do the following:

- Keep a running history of model discovery experiments.
- Establish model governance by tracking model lineage artifacts for auditing and compliance verification.
- Clone and rerun workflows to experiment with what-if scenarios while developing models.
- Share a workflow that colleagues can reproduce and enhance (for example, while collaborating on solving a business problem).
• Clone and rerun workflows with additional debugging or logging routines, or new input variations for troubleshooting issues in production models.

Topics
• Tracking Entities (p. 1464)
• Amazon SageMaker Created Tracking Entities (p. 1465)
• Manually Create Tracking Entities (p. 1467)

Tracking Entities

Tracking entities maintain a representation of all the elements of your end-to-end machine learning workflow. You can use this representation to establish model governance, reproduce your workflow, and maintain a record of your work history.

Amazon SageMaker automatically creates tracking entities for trial components and their associated trials and experiments when you create SageMaker jobs such as processing jobs, training jobs, and batch transform jobs. For more information, see Manage Machine Learning with Amazon SageMaker Experiments (p. 845).

SageMaker also automatically creates tracking entities for the other steps in a workflow that enable you to track the workflow from end to end. For more information, see Amazon SageMaker Created Tracking Entities (p. 1465).

You can create additional entities to supplement those created by SageMaker. For more information, see Manually Create Tracking Entities (p. 1467).

SageMaker reuses any existing entities rather than create new ones. For example, there can be only one artifact with a unique SourceUri.

The following tracking entities are defined:

Experiment entities

• Trial component – A stage of a machine learning trial. Includes processing jobs, training jobs, and batch transform jobs.
• Trial – A combination of trial components that generally produces a model.
• Experiment – A grouping of trials generally focused on solving a specific use case.

Lineage entities

• Context – Provides a logical grouping of other tracking or experiment entities. Conceptually, experiments and trials are contexts. Some examples are an endpoint and a model package.
• Action – Represents an action or activity. Generally, an action involves at least one input artifact or output artifact. Some examples are a workflow step and a model deployment.
• Artifact – Represents a URI addressable object or data. An artifact is generally either an input or an output to a trial component or Action. Some examples include a dataset (Amazon S3 bucket URI), an image (Amazon ECR registry path), or an action (ARN).
• Association – Links other tracking or experiment entities. For example, an association between the location of training data and a training job.

An association has an optional AssociationType property. The following values are available along with the suggested use for each type. SageMaker places no restrictions on their use:

• ContributedTo – The source contributed to the destination or had a part in enabling the destination. For example, the training data contributed to the training job.
SageMaker Created Entities

- **AssociatedWith** - The source is connected to the destination. For example, an approval workflow is associated with a model deployment.
- **DerivedFrom** - The destination is a modification of the source. For example, a digest output of a channel input for a processing job is derived from the original inputs.
- **Produced** - The source generated the destination. For example, a training job produced a model artifact.

Common properties

- **Type property**
  The action, artifact, and context entities have a *type* property, **ActionType**, **ArtifactType**, and **ContextType**, respectively. This property is a custom string which can associate meaningful information with the entity and be used as a filter in the List APIs.

- **Source property**
  The action, artifact, and context entities have a *Source* property. This property provides the underlying URI that the entity represents. Some examples are:
  - An **UpdateEndpoint** action where the source is the **EndpointArn**.
  - An image artifact for a processing job where the source is the **ImageUri**.
  - An **Endpoint** context where the source is the **EndpointArn**.

- **Metadata property**
  The action and artifact entities have an optional **Metadata** property which can provide the following information:
  - **ProjectId** – For example, the ID of the SageMaker MLOps project a model belongs to.
  - **GeneratedBy** – For example, the SageMaker pipeline execution that registered a model package version.
  - **Repository** – For example, the repository that contains an algorithm.
  - **CommitId** – For example, the commit ID of an algorithm version.

Amazon SageMaker Created Tracking Entities

Amazon SageMaker automatically creates tracking entities for SageMaker jobs, models, model packages, and endpoints if the data is available. There is no limit to the number of lineage entities automatically created by SageMaker.

For information on how you can manually create tracking entities, see [Manually Create Tracking Entities](p. 1467).

Topics

- [Tracking Entities for SageMaker Jobs](p. 1465)
- [Tracking Entities for Model Packages](p. 1466)
- [Tracking Entities for Endpoints](p. 1466)

Tracking Entities for SageMaker Jobs

A trial component is created for and associated with each SageMaker job. Artifacts are created to track the job metadata. Associations are created between each artifact and the job.

Artifacts are created for the following job properties and associated with the Amazon Resource Name (ARN) of the SageMaker job. The artifact **SourceUri** is listed in parentheses.
Training Job

- The image that contains the training algorithm (TrainingImage)
- The data source of each input channel (S3Uri)
- The location for the model (S3OutputPath)
- The location for the managed spot checkpoint data (S3Uri)

Processing Job

- The container to be run by the processing job (ImageUri)
- The data location for each processing input and processing output (S3Uri)

Transform Job

- The input data source to be transformed (S3Uri)
- The results of the transform (S3OutputPath)

Note
Amazon Simple Storage Service (Amazon S3) artifacts are tracked based on the S3 URI values provided to the create API, for example CreateTrainingJob, and not on the S3 key and hash/etag values from each file.

Tracking Entities for Model Packages

The following entities are created:

Model Packages

- A context for each model package group
- An artifact for each model package
- An association between each model package artifact and the context for each model package group the package belongs to
- An action for the creation of a model package version
- An association between the model package artifact and the creation action
- An association between the model package artifact and each model package group context the package belongs to
- Inference containers
  - An artifact for the image used in each container defined in the model package
  - An artifact for the model used in each container
  - An association between each artifact and the model package artifact
- Algorithms
  - An artifact for each algorithm defined in the model package
  - An artifact for the model created by each algorithm
  - An association between each artifact and the model package artifact

Tracking Entities for Endpoints

The following entities are created:
Endpoints

- A context for each endpoint
- An action for the model deployment that created each endpoint
- An artifact for each model deployed to the endpoint
- An artifact for the image used in the model
- An artifact for the model package for the model
- An artifact for each image deployed to the endpoint
- An association between each artifact and the model deployment action

Manually Create Tracking Entities

You can manually create tracking entities for any property. For information on the tracking entities that Amazon SageMaker automatically creates, see Amazon SageMaker Created Tracking Entities (p. 1465).

You can add tags to all entities except associations. Tags are arbitrary key-value pairs that provide custom information. You can filter or sort a list or search query by tags. For more information, see Tagging AWS resources in the AWS General Reference.

For a sample notebook that demonstrates how to create lineage entities, see the Amazon SageMaker Lineage notebook in the Amazon SageMaker example GitHub repository.

Topics

- Manually Create Entities (p. 1467)
- Manually Track a Workflow (p. 1469)
- Limits (p. 1470)

Manually Create Entities

The following procedure shows you how to create and associate artifacts between a SageMaker training job and endpoint. You perform the following steps:

- Create input artifacts for the source code, training data, and testing data locations.
- Create an output artifact for the generated model location.
- Create a trial component as a training job.
- Associate the input artifact and output artifacts with the training job.
- Train the model and create an endpoint.
- Create a context for the endpoint.
- Associate the training job and the endpoint context.

To create tracking entities and associations

1. Import the tracking entities.

```python
import sys
!{sys.executable} -m pip install -q sagemaker
from sagemaker import get_execution_role
from sagemaker.session import Session
from sagemaker.lineage import context, artifact, association, action
```
import boto3
boto_session = boto3.Session(region_name=region)
sagemaker_client = boto_session.client("sagemaker")

2. Create the input and output artifacts.

code_location_arn = artifact.Artifact.create(
    artifact_name='source-code-location',
    source_uri='s3://...',
    artifact_type='code-location'
).artifact_arn

# Similar constructs for train_data_location_arn and test_data_location_arn

model_location_arn = artifact.Artifact.create(
    artifact_name='model-location',
    source_uri='s3://...',
    artifact_type='model-location'
).artifact_arn

3. Train the model and get the trial_component_arn that represents the training job.
4. Associate the input artifacts and output artifacts with the training job (trial component).

input_artifacts = [code_location_arn, train_data_location_arn, test_data_location_arn]
for artifact_arn in input_artifacts:
    try:
        association.Association.create(
            source_arn=artifact_arn,
            destination_arn=trial_component_arn,
            association_type='ContributedTo'
        )
    except:
        logging.info('association between {} and {} already exists', artifact_arn,
            trial_component_arn)

output_artifacts = [model_location_arn]
for artifact_arn in output_artifacts:
    try:
        association.Association.create(
            source_arn=trial_component_arn,
            destination_arn=artifact_arn,
            association_type='Produced'
        )
    except:
        logging.info('association between {} and {} already exists', artifact_arn,
            trial_component_arn)

5. Create the inference endpoint.

predictor = mnist_estimator.deploy(initial_instance_count=1,
    instance_type='ml.m4.xlarge')

6. Create the endpoint context.

from sagemaker.lineage import context
endpoint = sagemaker_client.describe_endpoint(EndpointName=predictor.endpoint_name)
endpoint_arn = endpoint['EndpointArn']

endpoint_context_arn = context.Context.create(
    context_name=predictor.endpoint_name,
    context_type='Endpoint',
    source_uri=endpoint_arn)
7. Associate the training job (trial component) and endpoint context.

```python
association.Association.create(
    source_arn=trial_component_arn,
    destination_arn=endpoint_context_arn
)
```

**Manually Track a Workflow**

You can manually track the workflow created in the previous section.

Given the endpoint Amazon Resource Name (ARN) from the previous example, the following procedure shows you how to track the workflow back to the datasets used to train the model that was deployed to the endpoint. You perform the following steps:

- Given the endpoint ARN, get the endpoint context.
- Get the trial component from the association between the trial component and the endpoint context.
- Get the training data location artifact from the association between the trial component and the endpoint context.
- Get the training data location from the training data location artifact.

**To track a workflow from endpoint to training data source**

1. Import the tracking entities.

```python
import sys
!{sys.executable} -m pip install -q sagemaker
from sagemaker import get_execution_role
from sagemaker.session import Session
from sagemaker.lineage import context, artifact, association, action
import boto3
boto_session = boto3.Session(region_name=region)
sagemaker_client = boto3.Session.client("sagemaker")
```

2. Get the endpoint context from the endpoint ARN.

```python
endpoint_context_arn = sagemaker_client.list_contexts(
    SourceUri=endpoint_arn)['ContextSummaries'][0]['ContextArn']
```

3. Get the trial component from the association between the trial component and the endpoint context.

```python
trial_component_arn = sagemaker_client.list_associations(
    DestinationArn=endpoint_context_arn)['AssociationSummaries'][0]['SourceArn']
```

4. Get the training data location artifact from the association between the trial component and the endpoint context.

```python
train_data_location_artifact_arn = sagemaker_client.list_associations(
    DestinationArn=trial_source_arn)['AssociationSummaries'][0]['SourceArn']
```

5. Get the training data location from the training data location artifact.
train_data_location = sagemaker_client.describe_artifact(
    ArtifactArn=train_data_location_artifact_arn)["Source"]['SourceUri']
print(train_data_location)

Response:
s3://sagemaker-sample-data-us-east-2/mxnet/mnist/train

Limits

An association can be created between any entities, experiment and lineage, except the following:

- An association can't be created between two experiment entities. Experiment entities consist of experiments, trials, and trial components.
- An association can't be created with another association.

An error occurs if you try to create an entity that already exists.

Maximum number of manually created lineage entities

- Actions: 3000
- Artifacts: 6000
- Associations: 6000
- Contexts: 500

There is no limit to the number of lineage entities automatically created by SageMaker.

Kubernetes Orchestration

You can orchestrate your SageMaker training and inference jobs with SageMaker Operators for Kubernetes and SageMaker Components for Kubeflow Pipelines. SageMaker Operators for Kubernetes make it easier for developers and data scientists using Kubernetes to train, tune, and deploy machine learning (ML) models in SageMaker. SageMaker Components for Kubeflow Pipelines allow you to move your data processing and training jobs from the Kubernetes cluster to SageMaker’s machine learning-optimized managed service.

Contents

- SageMaker Operators for Kubernetes (p. 1470)
- SageMaker Components for Kubeflow Pipelines (p. 1501)

SageMaker Operators for Kubernetes

SageMaker Operators for Kubernetes make it easier for developers and data scientists using Kubernetes to train, tune, and deploy machine learning (ML) models in SageMaker. You can install these SageMaker Operators on your Kubernetes cluster in Amazon Elastic Kubernetes Service (Amazon EKS) to create SageMaker jobs natively using the Kubernetes API and command-line Kubernetes tools such as kubectl. This guide shows you how to set up the operators. The guide also explains how to use the operators to run model training, hyperparameter tuning, and inference (real-time and batch).
There is no additional charge to use these operators. You do incur charges for any SageMaker resources that you use through these operators. The procedures and guidelines here assume you are familiar with Kubernetes and its basic commands.

Contents

- What is an operator? (p. 1471)
- IAM role-based setup and operator deployment (p. 1472)
- Delete operators (p. 1481)
- Troubleshooting (p. 1483)
- Images and SMlogs in each Region (p. 1483)
- Using SageMaker Jobs (p. 1484)

What is an operator?

Kubernetes is built on top of what is called the controller pattern. This pattern allows applications and tools to listen to a central state manager (ETCD) and act when something happens. Examples of such applications include cloud-controller-manager and controller-manager. The controller pattern allows you to create decoupled experiences and not have to worry about how other components are integrated. To add new capabilities to Kubernetes, developers can extend the Kubernetes API by creating a custom resource that contains their application-specific or domain-specific logic and components. Operators in Kubernetes allow users to natively invoke these custom resources and automate associated workflows.

Prerequisites

This guide assumes that you've completed the following prerequisites:

- Installed the following tools on the client machine used to access your Kubernetes cluster:
  - kubectl Version 1.13 or later. Use a kubectl version that is within one minor version of your Amazon EKS cluster control plane. For example, a 1.13 kubectl client works with Kubernetes 1.13 and 1.14 clusters. OpenID Connect (OIDC) is not supported in versions earlier than 1.13.
  - eksctl Version 0.7.0 or later
  - AWS CLI Version 1.16.232 or later
  - (optional) Helm Version 3.0 or later
  - aws-iam-authenticator

- Have IAM permissions to create roles and attach policies to roles.
- Created a Kubernetes cluster on which to run the operators. It should either be Kubernetes version 1.13 or 1.14. For automated cluster creation using eksctl, see Getting Started with eksctl. It takes 20 to 30 minutes to provision a cluster.

Permissions overview

The SageMaker Operators for Kubernetes allow you to manage jobs in SageMaker from your Kubernetes cluster. The operators access SageMaker resources on your behalf. The IAM role that the operator assumes to interact with AWS resources differs from the credentials you use to access the Kubernetes cluster. The role also differs from the role that SageMaker assumes when running your machine learning jobs. The following image explains this design and flow.
IAM role-based setup and operator deployment

The following sections describe the steps to set up and deploy the operator.
Cluster-scoped deployment

Before you can deploy your operator using an IAM role, associate an OpenID Connect (OIDC) provider with your role to authenticate with the IAM service.

Create an OpenID Connect Provider for Your Cluster

The following instructions show how to create and associate an OIDC provider with your Amazon EKS cluster.

1. Set the local `CLUSTER_NAME` and `AWS_REGION` environment variables as follows:

```bash
# Set the Region and cluster
export CLUSTER_NAME="<your cluster name>"
export AWS_REGION="<your region>"
```

2. Use the following command to associate the OIDC provider with your cluster. For more information, see Enabling IAM Roles for Service Accounts on your Cluster.

```bash
eksctl utils associate-iam-oidc-provider --cluster ${CLUSTER_NAME} --region ${AWS_REGION} --approve
```

Your output should look like the following:

```bash
[.] eksctl version 0.10.1
[.] using region us-east-1
[.] IAM OpenID Connect provider is associated with cluster "my-cluster" in "us-east-1"
```

Now that the cluster has an OIDC identity provider, you can create a role and give a Kubernetes ServiceAccount permission to assume the role.

Get the OIDC ID

To set up the ServiceAccount, obtain the OpenID Connect issuer URL using the following command:

```bash
aws eks describe-cluster --name ${CLUSTER_NAME} --region ${AWS_REGION} --query cluster.identity.oidc.issuer --output text
```

The command returns a URL like the following:

```
https://oidc.eks.${AWS_REGION}.amazonaws.com/id/D48675832CA65BD10A532F5970IDCID
```

In this URL, the value `D48675832CA65BD10A532F5970IDCID` is the OIDC ID. The OIDC ID for your cluster is different. You need this OIDC ID value to create a role.

If your output is `None`, it means that your client version is old. To work around this, run the following command:

```bash
aws eks describe-cluster --region ${AWS_REGION} --query cluster --name ${CLUSTER_NAME} --output text | grep OIDC
```

The OIDC URL is returned as follows:

```
OIDC https://oidc.eks.us-east-1.amazonaws.com/id/D48675832CA65BD10A532F5970IDCID
```
Create an IAM Role

1. Create a file named `trust.json` and insert the following trust relationship code block into it. Be sure to replace all `<OIDC ID>`, `<AWS account number>`, and `<EKS Cluster region>` placeholders with values corresponding to your cluster.

   ```json
   {  
     "Version": "2012-10-17",
     "Statement": [  
       {  
         "Effect": "Allow",
         "Principal": {  
           "Federated": "arn:aws:iam::<AWS account number>:oidc-provider/oidc.eks.<EKS Cluster region>.amazonaws.com/id/<OIDC ID>"
         },  
         "Action": "sts:AssumeRoleWithWebIdentity",
         "Condition": {  
           "StringEquals": {  
             "oidc.eks.<EKS Cluster region>.amazonaws.com/id/<OIDC ID>:aud": "sts.amazonaws.com",
           }
         }
       }
     ]
   }
   ```

2. Run the following command to create a role with the trust relationship defined in `trust.json`. This role enables the Amazon EKS cluster to get and refresh credentials from IAM.

   ```bash
   aws iam create-role --region ${AWS_REGION} --role-name <role name> --assume-role-policy-document file://trust.json --output=text
   ```

   Your output should look like the following:

   ```
   ASSUMEROLEPOLICYDOCUMENT        2012-10-17
   STATEMENT       sts:AssumeRoleWithWebIdentity   Allow
   PRINCIPAL       arn:aws:iam::123456789012:oidc-provider/oidc.eks.us-east-1.amazonaws.com/id/
   ```

   Take note of ROLE ARN; you pass this value to your operator.

Attach the AmazonSageMakerFullAccess Policy to the Role

To give the role access to SageMaker, attach the AmazonSageMakerFullAccess policy. If you want to limit permissions to the operator, you can create your own custom policy and attach it.

To attach AmazonSageMakerFullAccess, run the following command:

```bash
aws iam attach-role-policy --role-name <role name> --policy-arn arn:aws:iam::aws:policy/AmazonSageMakerFullAccess
```

The Kubernetes ServiceAccount `sagemaker-k8s-operator-default` should have AmazonSageMakerFullAccess permissions. Confirm this when you install the operator.
Deploy the Operator

When deploying your operator, you can use either a YAML file or Helm charts.

Deploy the Operator Using YAML

This is the simplest way to deploy your operators. The process is as follows:

1. Download the installer script using the following command:

   ```bash
   ```

2. Edit the `installer.yaml` file to replace `eks.amazonaws.com/role-arn`. Replace the ARN here with the Amazon Resource Name (ARN) for the OIDC-based role you've created.

3. Use the following command to deploy the cluster:

   ```bash
   kubectl apply -f installer.yaml
   ```

Deploy the Operator Using Helm Charts

Use the provided Helm Chart to install the operator.

1. Clone the Helm installer directory using the following command:

   ```bash
   git clone https://github.com/aws/amazon-sagemaker-operator-for-k8s.git
   ```

2. Navigate to the `amazon-sagemaker-operator-for-k8s/hack/charts/installer` folder. Edit the `rolebased/values.yaml` file, which includes high-level parameters for the chart. Replace the role ARN here with the Amazon Resource Name (ARN) for the OIDC-based role you've created.

3. Install the Helm Chart using the following command:

   ```bash
   kubectl create namespace sagemaker-k8s-operator-system
   helm install --namespace sagemaker-k8s-operator-system sagemaker-operator rolebased/
   ```

   If you decide to install the operator into a namespace other than the one specified, you need to adjust the namespace defined in the IAM role `trust.json` file to match.

4. After a moment, the chart is installed with a randomly generated name. Verify that the installation succeeded by running the following command:

   ```bash
   helm ls
   ```

   Your output should look like the following:

<table>
<thead>
<tr>
<th>NAME</th>
<th>NAMESPACE</th>
<th>REVISION</th>
<th>UPDATED</th>
</tr>
</thead>
<tbody>
<tr>
<td>sagemaker-operator</td>
<td>sagemaker-k8s-operator-system</td>
<td>1</td>
<td>2019-11-20</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>STATUS</th>
<th>CHART</th>
<th>APP VERSION</th>
</tr>
</thead>
<tbody>
<tr>
<td>deployed</td>
<td>sagemaker-k8s-operator-0.1.0</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>sagemaker-operator</td>
</tr>
</tbody>
</table>

   23:14:59.6777082 +0000 UTC

   1475

Verify the operator deployment

1. You should be able to see the SageMaker Custom Resource Definitions (CRDs) for each operator deployed to your cluster by running the following command:
kubectl get crd | grep sagemaker

<table>
<thead>
<tr>
<th>Name</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>batchtransformjobs.sagemaker.aws.amazon.com</td>
<td>2019-11-20T17:12:34Z</td>
</tr>
<tr>
<td>endpointconfigs.sagemaker.aws.amazon.com</td>
<td>2019-11-20T17:12:34Z</td>
</tr>
<tr>
<td>hostingdeployments.sagemaker.aws.amazon.com</td>
<td>2019-11-20T17:12:34Z</td>
</tr>
<tr>
<td>hyperparametertuningjobs.sagemaker.aws.amazon.com</td>
<td>2019-11-20T17:12:34Z</td>
</tr>
<tr>
<td>models.sagemaker.aws.amazon.com</td>
<td>2019-11-20T17:12:34Z</td>
</tr>
<tr>
<td>trainingjobs.sagemaker.aws.amazon.com</td>
<td>2019-11-20T17:12:34Z</td>
</tr>
</tbody>
</table>

2. Ensure that the operator pod is running successfully. Use the following command to list all pods:

```bash
kubectl -n sagemaker-k8s-operator-system get pods
```

You should see a pod named `sagemaker-k8s-operator-controller-manager-*****` in the namespace `sagemaker-k8s-operator-system` as follows:

<table>
<thead>
<tr>
<th>NAME</th>
<th>AGE</th>
<th>READY</th>
<th>STATUS</th>
<th>RESTARTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>sagemaker-k8s-operator-controller-manager-12345678-r8abc</td>
<td>23s</td>
<td>2/2</td>
<td>Running</td>
<td>0</td>
</tr>
</tbody>
</table>

### Namespace-scoped deployment

You have the option to install your operator within the scope of an individual Kubernetes namespace. In this mode, the controller only monitors and reconciles resources with SageMaker if the resources are created within that namespace. This allows for finer-grained control over which controller is managing which resources. This is useful for deploying to multiple AWS accounts or controlling which users have access to particular jobs.

This guide outlines how to install an operator into a particular, predefined namespace. To deploy a controller into a second namespace, follow the guide from beginning to end and change out the namespace in each step.

#### Create an OpenID Connect Provider for Your Amazon EKS cluster

The following instructions show how to create and associate an OIDC provider with your Amazon EKS cluster.

1. Set the local `CLUSTER_NAME` and `AWS_REGION` environment variables as follows:

```bash
# Set the region and cluster
export CLUSTER_NAME="<your cluster name>"
export AWS_REGION="<your region>"
```

2. Use the following command to associate the OIDC provider with your cluster. For more information, see [Enabling IAM Roles for Service Accounts on your Cluster](#).

```bash
ekubectl -n sagemaker-k8s-operator-system get pods
```

Your output should look like the following:

```
2/2 Running 0
```

Your output should look like the following:

```
[...] eksctl version 0.10.1
```
Now that the cluster has an OIDC identity provider, create a role and give a Kubernetes ServiceAccount permission to assume the role.

Get your OIDC ID

To set up the ServiceAccount, first obtain the OpenID Connect issuer URL using the following command:

```
aws eks describe-cluster --name ${CLUSTER_NAME} --region ${AWS_REGION} \
--query cluster.identity.oidc.issuer --output text
```

The command returns a URL like the following:

```
https://oidc.eks.${AWS_REGION}.amazonaws.com/id/D48675832CA65BD10A532F597OIDCID
```

In this URL, the value D48675832CA65BD10A532F597OIDCID is the OIDC ID. The OIDC ID for your cluster will be different. You need this OIDC ID value to create a role.

If your output is `None`, it means that your client version is old. To work around this, run the following command:

```
aws eks describe-cluster --region ${AWS_REGION} --query cluster --name ${CLUSTER_NAME} --output text | grep OIDC
```

The OIDC URL is returned as follows:

```
OIDC https://oidc.eks.us-east-1.amazonaws.com/id/D48675832CA65BD10A532F597OIDCID
```

Create your IAM Role

1. Create a file named `trust.json` and insert the following trust relationship code block into it. Be sure to replace all `<OIDC ID>`, `<AWS account number>`, `<EKS Cluster region>`, and `<Namespace>` placeholders with values corresponding to your cluster. For the purposes of this guide, `my-namespace` is used for the `<Namespace>` value.

```json
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Federated": "arn:aws:iam::<AWS account number>:oidc-provider/oidc.eks.<EKS Cluster region>.amazonaws.com/id/<OIDC ID>"
      },
      "Action": "sts:AssumeRoleWithWebIdentity",
      "Condition": {
        "StringEquals": {
          "oidc.eks.<EKS Cluster region>.amazonaws.com/id/<OIDC ID>:aud": "sts.amazonaws.com",
          "oidc.eks.<EKS Cluster region>.amazonaws.com/id/<OIDC ID>:sub": "system:serviceaccount:<Namespace>:sagemaker-k8s-operator-default"
        }
      }
    }
  ]
}
```

1477
2. Run the following command to create a role with the trust relationship defined in trust.json. This role enables the Amazon EKS cluster to get and refresh credentials from IAM.

```bash
aws iam create-role --region ${AWS_REGION} --role-name <role name> --assume-role-policy-document file://trust.json --output=text
```

Your output should look like the following:

```
ASSUMEROLEPOLICYDOCUMENT 2012-10-17
STATEMENT  sts:AssumeRoleWithWebIdentity Allow
STRINGEQUALS sts.amazonaws.com system:serviceaccount:my-namespace:sagemaker-k8s-operator-default
PRINCIPAL   arn:aws:iam::123456789012:oidc-provider/oidc.eks.us-east-1.amazonaws.com/id/
```

Take note of ROLE ARN. You pass this value to your operator.

**Attach the AmazonSageMakerFullAccess Policy to your Role**

To give the role access to SageMaker, attach the AmazonSageMakerFullAccess policy. If you want to limit permissions to the operator, you can create your own custom policy and attach it.

To attach AmazonSageMakerFullAccess, run the following command:

```bash
aws iam attach-role-policy --role-name <role name> --policy-arn arn:aws:iam::aws:policy/AmazonSageMakerFullAccess
```

The Kubernetes ServiceAccount sagemaker-k8s-operator-default should have AmazonSageMakerFullAccess permissions. Confirm this when you install the operator.

**Deploy the Operator to Your Namespace**

When deploying your operator, you can use either a YAML file or Helm charts.

**Deploy the Operator to Your Namespace Using YAML**

There are two parts to deploying an operator within the scope of a namespace. The first is the set of CRDs that are installed at a cluster level. These resource definitions only need to be installed once per Kubernetes cluster. The second part is the operator permissions and deployment itself.

If you have not already installed the CRDs into the cluster, apply the CRD installer YAML using the following command:

```bash
kubectl apply -f https://raw.githubusercontent.com/aws/amazon-sagemaker-operator-for-k8s/master/release/rolebased/namespaced/crd.yaml
```

To install the operator onto the cluster:

1. Download the operator installer YAML using the following command:

   ```bash
   ```

2. Update the installer YAML to place the resources into your specified namespace using the following command:
3. Edit the `operator.yaml` file to place resources into your eks.amazonaws.com/role-arn. Replace the ARN here with the Amazon Resource Name (ARN) for the OIDC-based role you've created.

4. Use the following command to deploy the cluster:

   ```
   kubectl apply -f operator.yaml
   ```

### Deploy the Operator to Your Namespace Using Helm Charts

There are two parts needed to deploy an operator within the scope of a namespace. The first is the set of CRDs that are installed at a cluster level. These resource definitions only need to be installed once per Kubernetes cluster. The second part is the operator permissions and deployment itself. When using helm charts you have to first create the namespace using `kubectl`.

1. Clone the Helm installer directory using the following command:

   ```
   git clone https://github.com/aws/amazon-sagemaker-operator-for-k8s.git
   ```

2. Navigate to the `amazon-sagemaker-operator-for-k8s/hack/charts/installer/namespaced` folder. Edit the `rolebased/values.yaml` file, which includes high-level parameters for the chart. Replace the role ARN here with the Amazon Resource Name (ARN) for the OIDC-based role you've created.

3. Install the Helm Chart using the following command:

   ```
   helm install crds crd_chart/
   ```

4. Create the required namespace and install the operator using the following command:

   ```
   kubectl create namespace <namespace>
   helm install --n <namespace> op operator_chart/
   ```

5. After a moment, the chart is installed with the name `sagemaker-operator`. Verify that the installation succeeded by running the following command:

   ```
   helm ls
   ```

   Your output should look like the following:

<table>
<thead>
<tr>
<th>NAME</th>
<th>NAMESPACE</th>
<th>REVISION</th>
<th>UPDATED</th>
</tr>
</thead>
<tbody>
<tr>
<td>sagemaker-operator</td>
<td>my-namespace</td>
<td>1</td>
<td>2019-11-20</td>
</tr>
<tr>
<td>sagemaker-k8s-operator</td>
<td>my-namespace</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2019-11-20</td>
<td>sagemaker-k8s-operator-0.1.0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Verify the operator deployment to your namespace

1. You should be able to see the SageMaker Custom Resource Definitions (CRDs) for each operator deployed to your cluster by running the following command:

   ```
   kubectl get crd | grep sagemaker
   ```
Your output should look like the following:

<table>
<thead>
<tr>
<th>Resource</th>
<th>Created Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>batchtransformjobs.sagemaker.aws.amazon.com</td>
<td>2019-11-20T17:34Z</td>
</tr>
<tr>
<td>endpointconfigs.sagemaker.aws.amazon.com</td>
<td>2019-11-20T17:34Z</td>
</tr>
<tr>
<td>hostingdeployments.sagemaker.aws.amazon.com</td>
<td>2019-11-20T17:34Z</td>
</tr>
<tr>
<td>hyperparametertuningjobs.sagemaker.aws.amazon.com</td>
<td>2019-11-20T17:34Z</td>
</tr>
<tr>
<td>models.sagemaker.aws.amazon.com</td>
<td>2019-11-20T17:34Z</td>
</tr>
<tr>
<td>trainingjobs.sagemaker.aws.amazon.com</td>
<td>2019-11-20T17:34Z</td>
</tr>
</tbody>
</table>

2. Ensure that the operator pod is running successfully. Use the following command to list all pods:

```
kubectl -n my-namespace get pods
```

You should see a pod named `sagemaker-k8s-operator-controller-manager-{*}` in the namespace `my-namespace` as follows:

<table>
<thead>
<tr>
<th>NAME</th>
<th>READY</th>
<th>STATUS</th>
<th>RESTARTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>sagemaker-k8s-operator-controller-manager-12345678-r8abc</td>
<td>2/2</td>
<td>Running</td>
<td>0</td>
</tr>
<tr>
<td>23s</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Install the SageMaker logs `kubectl` plugin

As part of the SageMaker Operators for Kubernetes, you can use the `smlogs` plugin for `kubectl`. This enables SageMaker CloudWatch logs to be streamed with `kubectl`. `kubectl` must be installed onto your `PATH`. The following commands place the binary in the `sagemaker-k8s-bin` directory in your home directory, and add that directory to your `PATH`.

```bash
export os="linux"
wget https://amazon-sagemaker-operator-for-k8s-us-east-1.s3.amazonaws.com/kubectl-smlogs-plugin/v1/${os}.amd64.tar.gz
tar xvzf ${os}.amd64.tar.gz
# Move binaries to a directory in your homedir.
mkdir ~/sagemaker-k8s-bin
cp ./kubectl-smlogs-${os}.amd64/kubectl-smlogs ~/sagemaker-k8s-bin/.

# This line adds the binaries to your PATH in your .bashrc.
echo "export PATH=$PATH:~/sagemaker-k8s-bin" >> ~/.bashrc

# Source your .bashrc to update environment variables:
source ~/.bashrc
```

Use the following command to verify that the `kubectl` plugin is installed correctly:

```
kubectl smlogs
```

If the `kubectl` plugin is installed correctly, your output should look like the following:

**View SageMaker logs via Kubernetes**

**Usage:**

```
smlogs [command]
```
Amazon SageMaker Developer Guide
SageMaker Operators for Kubernetes

Aliases:
smlogs, SMLogs, Smlogs

Available Commands:
- **BatchTransformJob** View BatchTransformJob logs via Kubernetes
- **TrainingJob** View TrainingJob logs via Kubernetes
- **help** Help about any command

Flags:
- `--help` help for smlogs

Use "smlogs [command] --help" for more information about a command.

## Delete operators

### Delete cluster-based operators

Operators installed using YAML

To uninstall the operator from your cluster, make sure that all SageMaker resources have been deleted from the cluster. Failure to do so causes the operator delete operation to hang. Once you have deleted all SageMaker jobs, use `kubectl` to delete the operator from the cluster. Run the following commands to stop all jobs and delete the operator from the cluster:

```bash
# Delete all SageMaker jobs from Kubernetes
kubectl delete --all --all-namespaces hyperparametertuningjob.sagemaker.aws.amazon.com
kubectl delete --all --all-namespaces trainingjobs.sagemaker.aws.amazon.com
kubectl delete --all --all-namespaces batchtransformjob.sagemaker.aws.amazon.com
kubectl delete --all --all-namespaces hostingdeployment.sagemaker.aws.amazon.com

# Delete the operator and its resources
kubectl delete -f /installer.yaml
```

You should see output like the following:

```bash
# kubectl delete --all --all-namespaces trainingjobs.sagemaker.aws.amazon.com
trainingjobs.sagemaker.aws.amazon.com "xgboost-mnist-from-for-s3" deleted

# kubectl delete --all --all-namespaces hyperparametertuningjob.sagemaker.aws.amazon.com
hyperparametertuningjob.sagemaker.aws.amazon.com "xgboost-mnist-hpo" deleted

# kubectl delete --all --all-namespaces batchtransformjob.sagemaker.aws.amazon.com
batchtransformjob.sagemaker.aws.amazon.com "xgboost-mnist" deleted

# kubectl delete --all --all-namespaces hostingdeployment.sagemaker.aws.amazon.com
hostingdeployment.sagemaker.aws.amazon.com "host-xgboost" deleted

# kubectl delete -f raw-yaml/installer.yaml
namespace "sagemaker-k8s-operator-system" deleted
customresourcedefinition.apiextensions.k8s.io "batchtransformjobs.sagemaker.aws.amazon.com" deleted
customresourcedefinition.apiextensions.k8s.io "endpointconfigs.sagemaker.aws.amazon.com" deleted
customresourcedefinition.apiextensions.k8s.io "hostingdeployments.sagemaker.aws.amazon.com" deleted
"hyperparametertuningjobs.sagemaker.aws.amazon.com" deleted
customresourcedefinition.apiextensions.k8s.io "models.sagemaker.aws.amazon.com" deleted
customresourcedefinition.apiextensions.k8s.io "trainingjobs.sagemaker.aws.amazon.com" deleted
role.rbac.authorization.k8s.io "sagemaker-k8s-operator-leader-election-role" deleted
```
clusterrole.rbac.authorization.k8s.io "sagemaker-k8s-operator-manager-role" deleted
clusterrole.rbac.authorization.k8s.io "sagemaker-k8s-operator-proxy-role" deleted
rolebinding.rbac.authorization.k8s.io "sagemaker-k8s-operator-leader-election-rolebinding" deleted
clusterrolebinding.rbac.authorization.k8s.io "sagemaker-k8s-operator-manager-rolebinding" deleted
clusterrolebinding.rbac.authorization.k8s.io "sagemaker-k8s-operator-proxy-rolebinding" deleted
service "sagemaker-k8s-operator-controller-manager-metrics-service" deleted
deployment.apps "sagemaker-k8s-operator-controller-manager" deleted
secrets "sagemaker-k8s-operator-abcde" deleted

Operators installed using Helm Charts

To delete the operator CRDs, first delete all the running jobs. Then delete the Helm Chart that was used to deploy the operators using the following commands:

```
# get the helm charts
$ helm ls

# delete the charts
$ helm delete <chart name>
```

Delete namespace-based operators

Operators installed with YAML

To uninstall the operator from your cluster, make sure that all SageMaker resources have been deleted from the cluster. Failure to do so causes the operator delete operation to hang. Once you have deleted all SageMaker jobs, use `kubectl` to first delete the operator from the namespace and then the CRDs from the cluster. Run the following commands to stop all jobs and delete the operator from the cluster:

```
# Delete all SageMaker jobs from Kubernetes
kubectl delete --all --all-namespaces hyperparametertuningjob.sagemaker.aws.amazon.com
kubectl delete --all --all-namespaces trainingjobs.sagemaker.aws.amazon.com
kubectl delete --all --all-namespaces batchtransformjob.sagemaker.aws.amazon.com
kubectl delete --all --all-namespaces hostingdeployment.sagemaker.aws.amazon.com

# Delete the operator using the same yaml file that was used to install the operator
kubectl delete -f operator.yaml

# Now delete the CRDs using the CRD installer yaml

# Now you can delete the namespace if you want
kubectl delete namespace <namespace>
```

Operators installed with Helm Charts

To delete the operator CRDs, first delete all the running jobs. Then delete the Helm Chart that was used to deploy the operators using the following commands:

```
# Delete the operator
$ helm delete -n <namespace> op

# delete the crds
$ helm delete crds
```
Troubleshooting

Debugging a Failed Job

- Check the job status by running the following:

  ```
kubectl get <CRD Type> <job name>
  ```

- If the job was created in SageMaker, you can use the following command to see the STATUS and the SageMaker Job Name:

  ```
kubectl get <crd type> <job name>
  ```

- You can use `smlogs` to find the cause of the issue using the following command:

  ```
kubectl smlogs <crd type> <job name>
  ```

- You can also use the describe command to get more details about the job using the following command. The output has an additional field that has more information about the status of the job.

  ```
kubectl describe <crd type> <job name>
  ```

- If the job was not created in SageMaker, then use the logs of the operator’s pod to find the cause of the issue as follows:

  ```
  $ kubectl get pods -A | grep sagemaker
  # Output:
  sagemaker-k8s-operator-system   sagemaker-k8s-operator-controller-manager-5cd7df4d74-wh22z   2/2     Running   0          3h33m
  $ kubectl logs -p <pod name> -c manager -n sagemaker-k8s-operator-system
  ```

Deleting an Operator CRD

If deleting a job is not working, check if the operator is running. If the operator is not running, then you have to delete the finalizer using the following steps:

1. In a new terminal, open the job in an editor using `kubectl edit` as follows:

   ```
   $ kubectl edit <crd type> <job name>
   ```

2. Edit the job to delete the finalizer by removing the following two lines from the file. Save the file and the job is be deleted.

   ```
   finalizers:
   - sagemaker-operator-finalizer
   ```

Images and SMlogs in each Region

The following table lists the available operator images and SMLogs in each region.
Using SageMaker Jobs

To run a job using the SageMaker Operators for Kubernetes, you can either apply a YAML file or use the supplied Helm Charts.

All operator sample jobs in the following tutorials use sample data taken from a public MNIST dataset. In order to run these samples, download the dataset into your Amazon S3 bucket. You can find the dataset in Download the MNIST Dataset.

Contents
- TrainingJob operator (p. 1484)
- HyperParameterTuningJobs operator (p. 1488)
- BatchTransformJobs operator (p. 1493)
- Real-time inference (p. 606)

TrainingJob operator

Training job operators reconcile your specified training job spec to SageMaker by launching it for you in SageMaker. You can learn more about SageMaker training jobs in the SageMaker CreateTrainingJob API documentation.

Create a TrainingJob Using a Simple YAML File

1. Download the sample YAML file for training using the following command:

   ```bash
   wget https://raw.githubusercontent.com/aws/amazon-sagemaker-operator-for-k8s/master/samples/xgboost-mnist-trainingjob.yaml
   ```

2. Edit the `xgboost-mnist-trainingjob.yaml` file to replace the `roleArn` parameter with your `<sagemaker-execution-role>`, and `outputPath` with your Amazon S3 bucket that the SageMaker execution role has write access to. The `roleArn` must have permissions so that SageMaker can access Amazon S3, Amazon CloudWatch, and other services on your behalf. For more information on creating an SageMaker ExecutionRole, see SageMaker Roles. Apply the YAML file using the following command:

   ```bash
   kubectl apply -f xgboost-mnist-trainingjob.yaml
   ```
Create a TrainingJob Using a Helm Chart

You can use Helm Charts to run TrainingJobs.

1. Clone the GitHub repo to get the source using the following command:

```sh
git clone https://github.com/aws/amazon-sagemaker-operator-for-k8s.git
```

2. Navigate to the `amazon-sagemaker-operator-for-k8s/hack/charts/training-jobs/` folder and edit the `values.yaml` file to replace values like `rolearn` and `outputpath` with values that correspond to your account. The RoleARN must have permissions so that SageMaker can access Amazon S3, Amazon CloudWatch, and other services on your behalf. For more information on creating an SageMaker ExecutionRole, see SageMaker Roles.

Create the Training Job

With the roles and Amazon S3 buckets replaced with appropriate values in `values.yaml`, you can create a training job using the following command:

```sh
helm install . --generate-name
```

Your output should look like the following:

```
NAME: chart-12345678
LAST DEPLOYED: Wed Nov 20 23:35:49 2019
NAMESPACE: default
STATUS: deployed
REVISION: 1
TEST SUITE: None
NOTES: 
Thanks for installing the sagemaker-k8s-trainingjob.
```

Verify Your Training Helm Chart

To verify that the Helm Chart was created successfully, run:

```sh
helm ls
```

Your output should look like the following:

```
NAME                    NAMESPACE       REVISION        UPDATED                   CHART                           APP VERSION
chart-12345678          default         1               2019-11-20 23:35:49.9136092 +0000 UTC
rolebased-12345678      default         1               2019-11-20 23:14:59.6777082 +0000 UTC
```

helm install creates a TrainingJob Kubernetes resource. The operator launches the actual training job in SageMaker and updates the TrainingJob Kubernetes resource to reflect the status of the job in SageMaker. You incur charges for SageMaker resources used during the duration of your job. You do not incur any charges once your job completes or stops.

**Note:** SageMaker does not allow you to update a running training job. You cannot edit any parameter and re-apply the file/config. Either change the metadata name or delete the existing job and create a new one. Similar to existing training job operators like TFJob in Kubeflow, update is not supported.
List Training Jobs

Use the following command to list all jobs created using the Kubernetes operator:

```
kubectl get TrainingJob
```

The output listing all jobs should look like the following:

```
kubectl get trainingjobs
```

<table>
<thead>
<tr>
<th>NAME</th>
<th>STATUS</th>
<th>SECONDARY-STATUS</th>
<th>CREATION-TIME</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAGEMAKER-JOB-NAME</td>
<td>InProgress</td>
<td>Starting</td>
<td>2019-11-20T23:35:42Z</td>
</tr>
<tr>
<td>xgboost-mnist-from-for-s3</td>
<td></td>
<td></td>
<td>xgboost-mnist-from-for-s3-examplef1eab94e0ed4671d5a8f</td>
</tr>
</tbody>
</table>

A training job continues to be listed after the job has completed or failed. You can remove a TrainingJob job from the list by following the Delete a Training Job steps. Jobs that have completed or stopped do not incur any charges for SageMaker resources.

Training Job Status Values

The STATUS field can be one of the following values:

- Completed
- InProgress
- Failed
- Stopped
- Stopping

These statuses come directly from the SageMaker official API documentation.

In addition to the official SageMaker status, it is possible for STATUS to be SynchronizingK8sJobWithSageMaker. This means that the operator has not yet processed the job.

Secondary Status Values

The secondary statuses come directly from the SageMaker official API documentation. They contain more granular information about the status of the job.

Describe a Training Job

You can get more details about the training job by using the describe kubectl verb. This is typically used for debugging a problem or checking the parameters of a training job. To get information about your training job, use the following command:

```
kubectl describe trainingjob xgboost-mnist-from-for-s3
```

The output for your training job should look like the following:

```
Name:         xgboost-mnist-from-for-s3
Namespace:    default
Labels:       <none>
Annotations:  <none>
API Version:  sagemaker.aws.amazon.com/v1
Kind:         TrainingJob
Metadata:     
Creation Timestamp:  2019-11-20T23:35:42Z
Finalizers:
```
<table>
<thead>
<tr>
<th><strong>sagemaker-operator-finalizer</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Generation: 2</td>
</tr>
<tr>
<td>Resource Version: 23119</td>
</tr>
<tr>
<td>Self Link: /apis/sagemaker.aws.amazon.com/v1/namespaces/default/trainingjobs/xgboost-mnist-from-for-s3</td>
</tr>
<tr>
<td>UID: 6d7uiui-0bef-11ea-b94e-0ed467example</td>
</tr>
</tbody>
</table>

**Spec:**

- **Algorithm Specification:**
  - Training Image: 8256416981234.dkr.ecr.us-east-2.amazonaws.com/xgboost:1
  - Training Input Mode: File
  - Hyper Parameters:
    - Name: `eta`
      - Value: 0.2
    - Name: `gamma`
      - Value: 4
    - Name: `max_depth`
      - Value: 5
    - Name: `min_child_weight`
      - Value: 6
    - Name: `num_class`
      - Value: 10
    - Name: `num_round`
      - Value: 10
    - Name: `objective`
      - Value: `multi:softmax`
    - Name: `silent`
      - Value: 0

- **Input Data Config:**
  - Channel Name: `train`
    - Compression Type: None
    - Content Type: text/csv
    - Data Source:
      - S3 Data Source:
        - S3 Data Distribution Type: FullyReplicated
        - S3 Data Type: S3Prefix
        - S3 Uri: https://s3-us-east-2.amazonaws.com/my-bucket/sagemaker/xgboost-mnist/train/
  - Channel Name: `validation`
    - Compression Type: None
    - Content Type: text/csv
    - Data Source:
      - S3 Data Source:
        - S3 Data Distribution Type: FullyReplicated
        - S3 Data Type: S3Prefix
        - S3 Uri: https://s3-us-east-2.amazonaws.com/my-bucket/sagemaker/xgboost-mnist/validation/

- **Output Data Config:**
  - S3 Output Path: s3://my-bucket/sagemaker/xgboost-mnist/xgboost/
  - Region: us-east-2

- **Resource Config:**
  - Instance Count: 1
  - Instance Type: ml.m4.xlarge
  - Volume Size In GB: 5
  - Role Arn: arn:aws:iam:12345678910:role/service-role/AmazonSageMaker-ExecutionRole

- **Stopping Condition:**
  - Max Runtime In Seconds: 86400

- **Status:**
  - Training Job Name: xgboost-mnist-from-for-s3-6d7fa0af0bef11eab94e0example
  - Last Check Time: 2019-11-20T23:44:29Z
  - Sage Maker Training Job Name: xgboost-mnist-from-for-s3-6d7fa0af0bef11eab94e0example
  - Secondary Status: Downloading
  - Training Job Status: InProgress
View Logs from Training Jobs

Use the following command to see the logs from the kmeans-mnist training job:

```
kubectl smlogs trainingjob kmeans-mnist-from-for-s3
```

Your output should look similar to the following. The logs from instances are ordered chronologically.

```
xgboost-mnist-from-for-s3 has SageMaker TrainingJobName "xgboost-mnist-from-for-s3-123456789" in region "us-east-2", status "InProgress" and secondary status "Starting"
xgboost-mnist-from-for-s3-6d7fa0af0bef11eab94e0ed46example/algo-1-1574293123 2019-11-20 23:45:24.7 +0000 UTC Arguments: train
xgboost-mnist-from-for-s3-6d7fa0af0bef11eab94e0ed46example/algo-1-1574293123 2019-11-20 23:45:24.7 +0000 UTC [2019-11-20:23:45:22:INFO] File size need to be processed in the node: 1122.95mb. Available memory size in the node: 8586.0mb
xgboost-mnist-from-for-s3-6d7fa0af0bef11eab94e0ed46example/algo-1-1574293123 2019-11-20 23:45:24.7 +0000 UTC [2019-11-20:23:45:22:INFO] Determined delimiter of CSV input is ','
xgboost-mnist-from-for-s3-6d7fa0af0bef11eab94e0ed46example/algo-1-1574293123 2019-11-20 23:45:24.7 +0000 UTC [23:45:22] S3DistributionType set as FullyReplicated
```

Delete Training Jobs

Use the following command to stop a training job on Amazon SageMaker:

```
kubectl delete trainingjob xgboost-mnist-from-for-s3
```

This command removes the SageMaker training job from Kubernetes. This command returns the following output:

```
trainingjob.sagemaker.aws.amazon.com "xgboost-mnist-from-for-s3" deleted
```

If the job is still in progress on SageMaker, the job stops. You do not incur any charges for SageMaker resources after your job stops or completes.

**Note:** SageMaker does not delete training jobs. Stopped jobs continue to show on the SageMaker console. The delete command takes about 2 minutes to clean up the resources from SageMaker.

HyperParameterTuningJobs operator

Hyperparameter tuning job operators reconcile your specified hyperparameter tuning job spec to SageMaker by launching it in SageMaker. You can learn more about SageMaker hyperparameter tuning jobs in the SageMaker HyperparameterTuningJob API documentation.

Create a Hyperparameter Tuning Job Using a Simple YAML File

1. Download the sample YAML file for the hyperparameter tuning job using the following command:

   ```
wget https://raw.githubusercontent.com/aws/amazon-sagemaker-operator-for-k8s/master/samples/xgboost-mnist-hpo.yaml
```

2. Edit the xgboost-mnist-hpo.yaml file to replace the roleArn parameter with your sagemaker-execution-role. For the hyperparameter tuning job to succeed, you must also change the s3InputPath and s3OutputPath to values that correspond to your account. Apply the updates YAML file using the following command:
kubectl apply -f xgboost-mnist-hpo.yaml

Create a Hyperparameter Tuning Job using a Helm Chart

You can use Helm Charts to run hyperparameter tuning jobs.

1. Clone the GitHub repo to get the source using the following command:

   ```bash
git clone https://github.com/aws/amazon-sagemaker-operator-for-k8s.git
   ```

2. Navigate to the `amazon-sagemaker-operator-for-k8s/hack/charts/hyperparameter-tuning-jobs/` folder.

3. Edit the `values.yaml` file to replace the `roleArn` parameter with your `sagemaker-execution-role`. For the hyperparameter tuning job to succeed, you must also change the `s3InputPath` and `s3OutputPath` to values that correspond to your account.

Create the Hyperparameter Tuning Job

With the roles and Amazon S3 paths replaced with appropriate values in `values.yaml`, you can create a hyperparameter tuning job using the following command:

   ```bash
   helm install . --generate-name
   ```

Your output should look similar to the following:

```
NAME: chart-1574292948
LAST DEPLOYED: Wed Nov 20 23:35:49 2019
NAMESPACE: default
STATUS: deployed
REVISION: 1
TEST SUITE: None
NOTES:
Thanks for installing the sagemaker-k8s-hyperparametertuningjob.
```

Verify Chart Installation

To verify that the Helm Chart was created successfully, run the following command:

   ```bash
   helm ls
   ```

Your output should look like the following:

```
NAME                    NAMESPACE       REVISION        UPDATED
chart-1474292948        default         1               2019-11-20 23:35:49.9136092 +0000
UTC deployed            sagemaker-k8s-hyperparametertuningjob-0.1.0
STATUS                   CHART                           APP VERSION
chart-1574292948        default         1               2019-11-20 23:35:49.9136092 +0000
UTC deployed            sagemaker-k8s-trainingjob-0.1.0
rolebased-1574291698    default         1               2019-11-20 23:14:59.6777082 +0000
UTC deployed            sagemaker-k8s-operator-0.1.0
```

helm install creates a HyperParameterTuningJob Kubernetes resource. The operator launches the actual hyperparameter optimization job in SageMaker and updates the HyperParameterTuningJob
Kubernetes resource to reflect the status of the job in SageMaker. You incur charges for SageMaker resources used during the duration of your job. You do not incur any charges once your job completes or stops.

**Note:** SageMaker does not allow you to update a running hyperparameter tuning job. You cannot edit any parameter and re-apply the file/config. You must either change the metadata name or delete the existing job and create a new one. Similar to existing training job operators like TFJob in Kubeflow, update is not supported.

**List Hyperparameter Tuning Jobs**

Use the following command to list all jobs created using the Kubernetes operator:

```
kubectl get hyperparametertuningjob
```

Your output should look like the following:

<table>
<thead>
<tr>
<th>NAME</th>
<th>STATUS</th>
<th>CREATION-TIME</th>
<th>COMPLETED</th>
<th>INPROGRESS</th>
<th>ERRORS</th>
<th>STOPPED</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEST-TRAINING-JOB</td>
<td>Completed</td>
<td>2019-10-17T01:15:52Z</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>xgboost-mnist-hpo</td>
<td>xgboost-tha92f5e3cf07b11e9bf6c06d6-009-4c7a123</td>
<td>xgboost-tha92f5e3cf07b11e9bf6c123</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A hyperparameter tuning job continues to be listed after the job has completed or failed. You can remove a hyperparametertuningjob from the list by following the steps in Delete a Hyperparameter Tuning Job. Jobs that have completed or stopped do not incur any charges for SageMaker resources.

**Hyperparameter Tuning Job Status Values**

The **STATUS** field can be one of the following values:

- Completed
- InProgress
- Failed
- Stopped
- Stopping

These statuses come directly from the SageMaker official API documentation.

In addition to the official SageMaker status, it is possible for **STATUS** to be **SynchronizingK8sJobWithSageMaker**. This means that the operator has not yet processed the job.

**Status Counters**

The output has several counters, like **COMPLETED** and **INPROGRESS**. These represent how many training jobs have completed and are in progress, respectively. For more information about how these are determined, see **TrainingJobStatusCounters** in the SageMaker API documentation.

**Best Training Job**

This column contains the name of the **TrainingJob** that best optimized the selected metric.

To see a summary of the tuned hyperparameters, run:

```
kubectl describe hyperparametertuningjob xgboost-mnist-hpo
```

To see detailed information about the **TrainingJob**, run:
Spawned Training Jobs

You can also track all 10 training jobs in Kubernetes launched by HyperparameterTuningJob by running the following command:

```
kubectl get trainingjobs
```

Describe a Hyperparameter Tuning Job

You can obtain debugging details using the describe `kubectl` verb by running the following command.

```
kubectl describe hyperparametertuningjob xgboost-mnist-hpo
```

In addition to information about the tuning job, the SageMaker Operator for Kubernetes also exposes the best training job found by the hyperparameter tuning job in the `describe` output as follows:

```
Name:         xgboost-mnist-hpo
Namespace:    default
Labels:       <none>
Annotations:  kubectl.kubernetes.io/last-applied-configuration:
   "apiVersion":"sagemaker.aws.amazon.com/v1","kind":"HyperparameterTuningJob","metadata":{},"name":"xgboost-mnist-hpo","namespace":...
API Version:  sagemaker.aws.amazon.com/v1
Kind:         HyperparameterTuningJob
Metadata:
   Creation Timestamp:  2019-10-17T01:15:52Z
   Finalizers:        sagemaker-operator-finalizer
   Generation:        2
   Resource Version:  8167
   Self Link:         /apis/sagemaker.aws.amazon.com/v1/namespaces/default/hyperparametertuningjobs/xgboost-mnist-hpo
   UID:               a92f5e3c-f07b-11e9-bf6c-06d6f303uidu
Spec:
   Hyper Parameter Tuning Job Config:
      Hyper Parameter Tuning Job Objective:
         Metric Name:  validation:error
         Type:         Minimize
   Parameter Ranges:
      Integer Parameter Ranges:
         Max Value:     20
         Min Value:     10
         Name:          num_round
         Scaling Type:  Linear
   Resource Limits:
      Max Number Of Training Jobs:     10
      Max Parallel Training Jobs:      10
      Strategy:                          Bayesian
      Training Job Early Stopping Type:  Off
      Hyper Parameter Tuning Job Name:  xgboosta92f5e3cf07b11e9bf6c06d6
      Region:                              us-east-2
      Training Job Definition:
         Algorithm Specification:
            Training Image:       12345678910.dkr.ecr.us-east-2.amazonaws.com/xgboost:1
            Training Input Mode:  File
      Input Data Config:
         Channel Name:  train
```
Content Type: text/csv
Data Source:
s3DataSource:
  s3DataDistributionType: FullyReplicated
  s3DataType: S3Prefix
  s3Uri: https://s3-us-east-2.amazonaws.com/my-bucket/sagemaker/
xgboost-mnist/train/
    Channel Name: validation
    Content Type: text/csv
    Data Source:
      s3DataSource:
        s3DataDistributionType: FullyReplicated
        s3DataType: S3Prefix
        s3Uri: https://s3-us-east-2.amazonaws.com/my-bucket/sagemaker/
xgboost-mnist/validation/
    Output Data Config:
      s3OutputPath: https://s3-us-east-2.amazonaws.com/my-bucket/sagemaker/xgboost-mnist/
xgboost
Resource Config:
  Instance Count: 1
  Instance Type: ml.m4.xlarge
  Volume Size In GB: 5
  Role Arn: arn:aws:iam::123456789012:role/service-role/AmazonSageMaker-
ExecutionRole
Static Hyper Parameters:
  Name: base_score
  Value: 0.5
  Name: booster
  Value: gbtree
  Name: csv_weights
  Value: 0
  Name: dsplit
  Value: row
  Name: grow_policy
  Value: depthwise
  Name: lambda_bias
  Value: 0.0
  Name: max_bin
  Value: 256
  Name: max_leaves
  Value: 0
  Name: normalize_type
  Value: tree
  Name: objective
  Value: reg:linear
  Name: one_drop
  Value: 0
  Name: prob_buffer_row
  Value: 1.0
  Name: process_type
  Value: default
  Name: rate_drop
  Value: 0.0
  Name: refresh_leaf
  Value: 1
  Name: sample_type
  Value: uniform
  Name: scale_pos_weight
  Value: 1.0
  Name: silent
  Value: 0
  Name: sketch_eps
  Value: 0.03
  Name: skip_drop
  Value: 0.0
  Name: tree_method
View Logs from Hyperparameter Tuning Jobs

Hyperparameter tuning jobs do not have logs, but all training jobs launched by them do have logs. These logs can be accessed as if they were a normal training job. For more information, see View Logs from Training Jobs.

Delete a Hyperparameter tuning Job

Use the following command to stop a hyperparameter job in SageMaker.

```
kubectl delete hyperparametertuningjob xgboost-mnist-hpo
```

This command removes the hyperparameter tuning job and associated training jobs from your Kubernetes cluster and stops them in SageMaker. Jobs that have stopped or completed do not incur any charges for SageMaker resources. SageMaker does not delete hyperparameter tuning jobs. Stopped jobs continue to show on the SageMaker Console.

Your output should look like the following:

```
hyperparametertuningjob.sagemaker.aws.amazon.com "xgboost-mnist-hpo" deleted
```

**Note:** The delete command takes about 2 minutes to clean up the resources from SageMaker.

**BatchTransformJobs operator**

Batch transform job operators reconcile your specified batch transform job spec to SageMaker by launching it in SageMaker. You can learn more about SageMaker batch transform job in the SageMaker CreateTransformJob API documentation.
Create a BatchTransformJob Using a Simple YAML File

1. Download the sample YAML file for the batch transform job using the following command:

   ```bash
   wget https://raw.githubusercontent.com/aws/amazon-sagemaker-operator-for-k8s/master/samples/xgboost-mnist-batchtransform.yaml
   ```

2. Edit the file `xgboost-mnist-batchtransform.yaml` to change necessary parameters to replace the `inputdataconfig` with your input data and `s3OutputPath` with your Amazon S3 buckets that the SageMaker execution role has write access to.

3. Apply the YAML file using the following command:

   ```bash
   kubectl apply -f xgboost-mnist-batchtransform.yaml
   ```

Create a BatchTransformJob Using a Helm Chart

You can use Helm Charts to run batch transform jobs.

Get the Helm installer directory

Clone the GitHub repo to get the source using the following command:

```bash
git clone https://github.com/aws/amazon-sagemaker-operator-for-k8s.git
```

Configure the Helm Chart

Navigate to the `amazon-sagemaker-operator-for-k8s/hack/charts/batch-transform-jobs/` folder.

Edit the `values.yaml` file to replace the `inputdataconfig` with your input data and `outputPath` with your S3 buckets to which the SageMaker execution role has write access.

Create a Batch Transform Job

1. Use the following command to create a batch transform job:

   ```bash
   helm install . --generate-name
   ```

   Your output should look like the following:

   ```
   NAME: chart-1574292948
   LAST DEPLOYED: Wed Nov 20 23:35:49 2019
   NAMESPACE: default
   STATUS: deployed
   REVISION: 1
   TEST SUITE: None
   NOTES:
   Thanks for installing the sagemaker-k8s-batch-transform-job.
   ```

2. To verify that the Helm Chart was created successfully, run the following command:

   ```bash
   helm ls
   ```

   Your output should look like:

   ```
   NAME                    NAMESPACE       REVISION        UPDATED
   STATUS          CHART                           APP VERSION
   chart-1474292948        default         1               2019-11-20 23:35:49.913609+0000 UTC   deployed        sagemaker-k8s-batchtransformjob-0.1.0
   ```
This command creates a BatchTransformJob Kubernetes resource. The operator launches the actual transform job in SageMaker and updates the BatchTransformJob Kubernetes resource to reflect the status of the job in SageMaker. You incur charges for SageMaker resources used during the duration of your job. You do not incur any charges once your job completes or stops.

Note: SageMaker does not allow you to update a running batch transform job. You cannot edit any parameter and re-apply the file/config. You must either change the metadata name or delete the existing job and create a new one. Similar to existing training job operators like TFJob in Kubeflow, update is not supported.

List Batch Transform Jobs

Use the following command to list all jobs created using the Kubernetes operator:

```
kubectl get batchtransformjob
```

Your output should look like the following:

<table>
<thead>
<tr>
<th>NAME</th>
<th>STATUS</th>
<th>CREATION-TIME</th>
<th>SAGEMAKER-JOB-NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>xgboost-mnist-batch-transform</td>
<td>Completed</td>
<td>2019-11-18T03:44:00Z</td>
<td>xgboost-mnist-</td>
</tr>
<tr>
<td>a88fb19809b511eaac440aa8axgboost</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A batch transform job will continue to be listed after the job has completed or failed. You can remove a hyperparametertuningjob from the list by following the Delete a Batch Transform Job steps. Jobs that have completed or stopped do not incur any charges for SageMaker resources.

Batch Transform Status Values

The STATUS field can be one of the following values:

- Completed
- InProgress
- Failed
- Stopped
- Stopping

These statuses come directly from the SageMaker official API documentation.

In addition to the official SageMaker status, it is possible for STATUS to be SynchronizingK8sJobWithSageMaker. This means that the operator has not yet processed the job.

Describe a Batch Transform Job

You can obtain debugging details using the describe kubectl verb by running the following command.

```
kubectl describe batchtransformjob xgboost-mnist-batch-transform
```

Your output should look like the following:
**View Logs from Batch Transform Jobs**

Use the following command to see the logs from the `xgboost-mnist` batch transform job:

```
kubectl smlogs batchtransformjob xgboost-mnist-batch-transform
```

**Delete a Batch Transform Job**

Use the following command to stop a batch transform job in SageMaker.

```
kubectl delete batchTransformJob xgboost-mnist-batch-transform
```

Your output should look like the following:

```
batchtransformjob.sagemaker.aws.amazon.com "xgboost-mnist" deleted
```

This command removes the batch transform job from your Kubernetes cluster, as well as stops them in SageMaker. Jobs that have stopped or completed do not incur any charges for SageMaker resources. Delete takes about 2 minutes to clean up the resources from SageMaker.

**Note:** SageMaker does not delete batch transform jobs. Stopped jobs continue to show on the SageMaker console.
Real-time inference

HostingDeployments support creating and deleting an endpoint, as well as updating an existing endpoint. The hosting deployment operator reconciles your specified hosting deployment job spec to SageMaker by creating models, endpoint-configs and endpoints in SageMaker. You can learn more about SageMaker inference in the SageMaker CreateEndpoint API documentation.

Configure a HostingDeployment Resource

Download the sample YAML file for the hosting deployment job using the following command:

```
wget https://raw.githubusercontent.com/aws/amazon-sagemaker-operator-for-k8s/master/samples/xgboost-mnist-hostingdeployment.yaml
```

The `xgboost-mnist-hostingdeployment.yaml` file has the following components that can be edited as required:

- **ProductionVariants.** A production variant is a set of instances serving a single model. SageMaker load-balances between all production variants according to set weights.
- **Models.** A model is the containers and execution role ARN necessary to serve a model. It requires at least a single container.
- **Containers.** A container specifies the dataset and serving image. If you are using your own custom algorithm instead of an algorithm provided by SageMaker, the inference code must meet SageMaker requirements. For more information, see Using Your Own Algorithms with SageMaker.

Create a HostingDeployment

To create a HostingDeployment, use `kubectl` to apply the file `hosting.yaml` with the following command:

```
kubectl apply -f hosting.yaml
```

SageMaker creates an endpoint with the specified configuration. You incur charges for SageMaker resources used during the lifetime of your endpoint. You do not incur any charges once your endpoint is deleted.

The creation process takes approximately 10 minutes.

List HostingDeployments

To verify that the HostingDeployment was created, use the following command:

```
kubectl get hostingdeployments
```

Your output should look like the following:

<table>
<thead>
<tr>
<th>NAME</th>
<th>STATUS</th>
<th>SAGEMAKER-ENDPOINT-NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>host-xgboost</td>
<td>Creating</td>
<td>host-xgboost-def0e83e0d5f1leaaa450aSMLOGS</td>
</tr>
</tbody>
</table>

HostingDeployment Status Values

The status field can be one of several values:

- **SynchronizingK8sJobWithSageMaker:** The operator is preparing to create the endpoint.
- **ReconcilingEndpoint:** The operator is creating, updating, or deleting endpoint resources. If the HostingDeployment remains in this state, use `kubectl describe` to see the reason in the Additional field.
- **OutOfService**: Endpoint is not available to take incoming requests.
- **Creating**: `CreateEndpoint` is executing.
- **Updating**: `UpdateEndpoint` or `UpdateEndpointWeightsAndCapacities` is executing.
- **SystemUpdating**: Endpoint is undergoing maintenance and cannot be updated or deleted or re-scaled until it has completed. This maintenance operation does not change any customer-specified values such as VPC config, KMS encryption, model, instance type, or instance count.
- **RollingBack**: Endpoint fails to scale up or down or change its variant weight and is in the process of rolling back to its previous configuration. Once the rollback completes, endpoint returns to an `InService` status. This transitional status only applies to an endpoint that has autoscaling enabled and is undergoing variant weight or capacity changes as part of an `UpdateEndpointWeightsAndCapacities` call or when the `UpdateEndpointWeightsAndCapacities` operation is called explicitly.
- **InService**: Endpoint is available to process incoming requests.
- **Deleting**: `DeleteEndpoint` is executing.
- **Failed**: Endpoint could not be created, updated, or re-scaled. Use `DescribeEndpoint:FailureReason` for information about the failure. `DeleteEndpoint` is the only operation that can be performed on a failed endpoint.

### Describe a Hostingdeployment

You can obtain debugging details using the `describe` `kubectl` verb by running the following command.

```bash
kubectl describe hostingdeployment
```

Your output should look like the following:

```yaml
Name:         host-xgboost
Namespace:    default
Labels:       <none>
Annotations:  kubectl.kubernetes.io/last-applied-configuration:
               {"apiVersion":"sagemaker.aws.amazon.com/v1","kind":"HostingDeployment","metadata":{"annotations":{},"name":"host-xgboost","namespace":"default"}}
API Version:  sagemaker.aws.amazon.com/v1
Kind:         HostingDeployment
Metadata:
   Creation Timestamp: 2019-11-22T19:40:00Z
   Finalizers:        sagemaker-operator-finalizer
   Generation:       1
   Resource Version: 4258134
   Self Link:        /apis/sagemaker.aws.amazon.com/v1/namespaces/default/hostingdeployments/host-xgboost
   UID:              def0e83e-0d5f-11ea-aa45-0a3507uiduid
Spec:
   Containers:
      Container Hostname: xgboost
      Image:           123456789012.dkr.ecr.us-east-2.amazonaws.com/xgboost:latest
      Model Data URL:  s3://my-bucket/inference/xgboost-mnist/model.tar.gz
   Models:
      Containers:    xgboost
      Execution Role Arn: arn:aws:iam::123456789012:role/service-role/AmazonSageMaker-
                    ExecutionRole
      Name:          xgboost-model
      Primary Container: xgboost
      Production Variants:
         Initial Instance Count: 1
         Instance Type:          ml.c5.large
```

1498
The status field provides more information using the following fields:

- **Additional**: Additional information about the status of the hosting deployment. This field is optional and only gets populated in case of error.
- **Creation Time**: When the endpoint was created in SageMaker.
- **Endpoint ARN**: The SageMaker endpoint ARN.
- **Endpoint Config Name**: The SageMaker name of the endpoint configuration.
- **Endpoint Name**: The SageMaker name of the endpoint.
- **Endpoint Status**: The status of the endpoint.
- **Endpoint URL**: The HTTPS URL that can be used to access the endpoint. For more information, see Deploy a Model on SageMaker Hosting Services.
- **FailureReason**: If a create, update, or delete command fails, the cause is shown here.
- **Last Check Time**: The last time the operator checked the status of the endpoint.
- **Last Modified Time**: The last time the endpoint was modified.
- **Model Names**: A key-value pair of HostingDeployment model names to SageMaker model names.

### Invoking the Endpoint

Once the endpoint status is **InService**, you can invoke the endpoint in two ways: using the AWS CLI, which does authentication and URL request signing, or using an HTTP client like cURL. If you use your own client, you need to do AWSv4 URL signing and authentication on your own.

To invoke the endpoint using the AWS CLI, run the following command. Make sure to replace the Region and endpoint-name with your endpoint's Region and SageMaker endpoint name. This information can be obtained from the output of `kubectl describe`.

```bash
# Invoke the endpoint with mock input data.
aws sagemaker-runtime invoke-endpoint \ 
--region us-east-2 \ 
--endpoint-name <endpoint name> \ 
--body $(seq 784 | xargs echo | sed 's/ /,/g') \
>(cat) \ 
--content-type text/csv > /dev/null
```

For example, if your Region is `us-east-2` and your endpoint config name is `host-xgboost-f56b6b280d7511ea824b129926example`, then the following command would invoke the endpoint:

```bash
aws sagemaker-runtime invoke-endpoint \ 
--region us-east-2 \ 
--endpoint-name host-xgboost-f56b6b280d7511ea824b129926example \ 
--body $(seq 784 | xargs echo | sed 's/ /,/g') \
```

```
1499
```
Here, 4.95847082138 is the prediction from the model for the mock data.

**Update HostingDeployment**

1. Once a HostingDeployment has a status of **InService**, it can be updated. It might take about 10 minutes for HostingDeployment to be in service. To verify that the status is **InService**, use the following command:

```
kubectl get hostingdeployments
```

2. The HostingDeployment can be updated before the status is **InService**. The operator waits until the SageMaker endpoint is **InService** before applying the update.

To apply an update, modify the `hosting.yaml` file. For example, change the `initialInstanceCount` field from 1 to 2 as follows:

```yaml
apiVersion: sagemaker.aws.amazon.com/v1
kind: HostingDeployment
metadata:
  name: host-xgboost
spec:
  region: us-east-2
  productionVariants:
  - variantName: all-traffic
    modelName: xgboost-model
    initialInstanceCount: 2
    instanceType: ml.c5.large
  models:
  - name: xgboost-model
    executionRoleArn: arn:aws:iam::123456789012:role/service-role/AmazonSageMaker-ExecutionRole
    primaryContainer: xgboost
    containers:
      - xgboost
      containerHostname: xgboost
      modelDataUrl: s3://my-bucket/inference/xgboost-mnist/model.tar.gz
      image: 123456789012.dkr.ecr.us-east-2.amazonaws.com/xgboost:latest
```

3. Save the file, then use `kubectl` to apply your update as follows. You should see the status change from **InService** to **ReconcilingEndpoint**, then **Updating**.

```
$ kubectl apply -f hosting.yaml
hostingdeployment.sagemaker.aws.amazon.com/host-xgboost configured
$ kubectl get hostingdeployments
NAME           STATUS     SAGEMAKER-ENDPOINT-NAME
host-xgboost   ReconcilingEndpoint host-xgboost-def0e83e0d5f11eaa450a350abcdef
$ kubectl get hostingdeployments
NAME           STATUS     SAGEMAKER-ENDPOINT-NAME
host-xgboost   Updating   host-xgboost-def0e83e0d5f11eaa450a3507abcdef
```

SageMaker deploys a new set of instances with your models, switches traffic to use the new instances, and drains the old instances. As soon as this process begins, the status becomes **Updating**. After the update is complete, your endpoint becomes **InService**. This process takes approximately 10 minutes.
Delete the HostingDeployment

1. Use `kubectl` to delete a HostingDeployment with the following command:

   ```bash
   kubectl delete hostingdeployments host-xgboost
   ```

   Your output should look like the following:

   ```
   hostingdeployment.sagemaker.aws.amazon.com "host-xgboost" deleted
   ```

2. To verify that the hosting deployment has been deleted, use the following command:

   ```bash
   kubectl get hostingdeployments
   No resources found.
   ```

Endpoints that have been deleted do not incur any charges for SageMaker resources.

### SageMaker Components for Kubeflow Pipelines

This document outlines how to use SageMaker Components for Kubeflow Pipelines (KFP). With these pipeline components, you can create and monitor training, tuning, endpoint deployment, and batch transform jobs in SageMaker. By running Kubeflow Pipeline jobs on SageMaker, you move data processing and training jobs from the Kubernetes cluster to SageMaker's machine learning-optimized managed service. This document assumes prior knowledge of Kubernetes and Kubeflow.

**Contents**

- What is Kubeflow Pipelines? (p. 1501)
- Kubeflow Pipeline components (p. 1502)
- IAM permissions (p. 1503)
- Converting Pipelines to use SageMaker (p. 1503)
- Using SageMaker Components (p. 1504)

### What is Kubeflow Pipelines?

Kubeflow Pipelines (KFP) is a platform for building and deploying portable, scalable machine learning (ML) workflows based on Docker containers. The Kubeflow Pipelines platform consists of the following:

- A user interface (UI) for managing and tracking experiments, jobs, and runs.
- An engine (Argo) for scheduling multi-step ML workflows.
- A Python SDK for defining and manipulating pipelines and components.
- Notebooks for interacting with the system using the SDK.

A pipeline is a description of an ML workflow expressed as a directed acyclic graph as shown in the following diagram. Every step in the workflow is expressed as a Kubeflow Pipeline component, which is a Python module.

If your data has been preprocessed, the standard pipeline takes a subset of the data and runs hyperparameter optimization of the model. The pipeline then trains a model with the full dataset using the optimal hyperparameters. This model is used for both batch inference and endpoint creation.

For more information on Kubeflow Pipelines, see the Kubeflow Pipelines documentation.
Kubeflow Pipeline components

A Kubeflow Pipeline component is a set of code used to execute one step in a Kubeflow pipeline. Components are represented by a Python module that is converted into a Docker image. These components make it fast and easy to write pipelines for experimentation and production environments without having to interact with the underlying Kubernetes infrastructure.

What do SageMaker Components for Kubeflow Pipelines provide?

SageMaker Components for Kubeflow Pipelines offer an alternative to launching compute-intensive jobs in SageMaker. These components integrate SageMaker with the portability and orchestration of Kubeflow Pipelines. Using the SageMaker components, each of the jobs in the pipeline workflow runs on SageMaker instead of the local Kubernetes cluster. The job parameters, status, logs, and outputs from SageMaker are still accessible from the Kubeflow Pipelines UI. The following SageMaker components have been created to integrate six key SageMaker features into your ML workflows. You can create a Kubeflow Pipeline built entirely using these components, or integrate individual components into your workflow as needed.

There is no additional charge for using SageMaker Components for Kubeflow Pipelines. You incur charges for any SageMaker resources you use through these components.

Training components

Training

The Training component allows you to submit SageMaker Training jobs directly from a Kubeflow Pipelines workflow. For more information, see SageMaker Training Kubeflow Pipelines component.

Hyperparameter Optimization

The Hyperparameter Optimization component enables you to submit hyperparameter tuning jobs to SageMaker directly from a Kubeflow Pipelines workflow. For more information, see SageMaker hyperparameter optimization Kubeflow Pipeline component.

Processing

The Processing component enables you to submit processing jobs to SageMaker directly from a Kubeflow Pipelines workflow. For more information, see SageMaker Processing Kubeflow Pipeline component.

Inference components

Hosting Deploy

The Deploy component enables you to deploy a model in SageMaker Hosting from a Kubeflow Pipelines workflow. For more information, see SageMaker Hosting Services - Create Endpoint Kubeflow Pipeline component.

Batch Transform component

The Batch Transform component enables you to run inference jobs for an entire dataset in SageMaker from a Kubeflow Pipelines workflow. For more information, see SageMaker Batch Transform Kubeflow Pipeline component.

Ground Truth components

Ground Truth The Ground Truth component enables you to submit SageMaker Ground Truth labeling jobs directly from a Kubeflow Pipelines workflow. For more information, see SageMaker Ground Truth Kubeflow Pipelines component.
Workteam

The Workteam component enables you to create SageMaker private workteam jobs directly from a Kubeflow Pipelines workflow. For more information, see SageMaker create private workteam Kubeflow Pipelines component.

IAM permissions

Deploying Kubeflow Pipelines with SageMaker components requires the following three levels of IAM permissions:

- An IAM user/role to access your AWS account (*your_credentials*). Note: You don’t need this at all if you already have access to KFP web UI and have your input data in Amazon S3, or if you already have an Amazon Elastic Kubernetes Service (Amazon EKS) cluster with KFP.

  You use this user/role from your gateway node, which can be your local machine or a remote instance, to:
  - Create an Amazon EKS cluster and install KFP
  - Create IAM roles/users
  - Create Amazon S3 buckets for your sample input data

  The IAM user/role needs the following permissions:
  - CloudWatchLogsFullAccess
  - AWSCloudFormationFullAccess
  - IAMFullAccess
  - AmazonS3FullAccess
  - AmazonEC2FullAccess
  - AmazonEKSAdminPolicy (Create this policy using the schema from Amazon EKS Identity-Based Policy Examples)

- An IAM role used by KFP pods to access SageMaker (*kfp-example-pod-role*) The KFP pods use this permission to create SageMaker jobs from KFP components. Note: If you want to limit permissions to the KFP pods, create your own custom policy and attach it.

  The role needs the following permission:
  - AmazonSageMakerFullAccess

- An IAM role used by SageMaker jobs to access resources such as Amazon S3 and Amazon ECR etc. (*kfp-example-sagemaker-execution-role*).

  Your SageMaker jobs use this role to:
  - Access SageMaker resources
  - Input Data from Amazon S3
  - Store your output model to Amazon S3

  The role needs the following permissions:
  - AmazonSageMakerFullAccess
  - AmazonS3FullAccess

These are all the IAM users/roles you need to run KFP components for SageMaker.

When you have run the components and have created the SageMaker endpoint, you also need a role with the sagemaker:InvokeEndpoint permission to query inference endpoints.

Converting Pipelines to use SageMaker

You can convert an existing pipeline to use SageMaker by porting your generic Python processing containers and training containers. If you are using SageMaker for inference, you also need to attach IAM permissions to your cluster and convert an artifact to a model.
Using SageMaker Components

In this tutorial, you run a pipeline using SageMaker Components for Kubeflow Pipelines to train a classification model using Kmeans with the MNIST dataset. This workflow uses Kubeflow pipelines as the orchestrator and SageMaker as the backend to run the steps in the workflow. For the full code for this and other pipeline examples, see the Sample SageMaker Kubeflow Pipelines. For information on the components used, see the KubeFlow Pipelines GitHub repository.

Contents
- Setup (p. 1504)
- Running the Kubeflow Pipeline (p. 1510)

Setup

To use Kubeflow Pipelines (KFP), you need an Amazon Elastic Kubernetes Service (Amazon EKS) cluster and a gateway node to interact with that cluster. The following sections show the steps needed to set up these resources.

Set up a gateway node

A gateway node is used to create an Amazon EKS cluster and access the Kubeflow Pipelines UI. Use your local machine or an Amazon EC2 instance as your gateway node. If you want to use a new Amazon EC2 instance, create one with the latest Ubuntu 18.04 DLAMI version from the AWS console using the steps in Launching and Configuring a DLAMI.

Complete the following steps to set up your gateway node. Depending on your environment, you may have certain requirements already configured.

1. If you don't have an existing Amazon EKS cluster, create a user named your_credentials using the steps in Creating an IAM User in Your AWS Account. If you have an existing Amazon EKS cluster, use the credentials of the IAM role or user that has access to it.
2. Add the following permissions to your user using the steps in Changing Permissions for an IAM User:
   - CloudWatchLogsFullAccess
   - AWSCloudFormationFullAccess
   - IAMFullAccess
   - AmazonS3FullAccess
   - AmazonEC2FullAccess
   - AmazonEKSAdminPolicy (Create this policy using the schema from Amazon EKS Identity-Based Policy Examples)
3. Install the following on your gateway node to access the Amazon EKS cluster and KFP UI.
   - AWS CLI. If you are using an IAM user, configure your Access Key ID, Secret Access Key and preferred AWS Region by running: aws configure
   - aws-iam-authenticator version 0.1.31 and above
   - eksctl version above 0.15.
   - kubectl (The version needs to match your Kubernetes version within one minor version)
4. Install boto3.
   ```bash
   pip install boto3
   ```

Set up an Amazon EKS cluster

Run the following steps from the command line of your gateway node to set up an Amazon EKS cluster:
1. If you do not have an existing Amazon EKS cluster, complete the following substeps. If you already have an Amazon EKS cluster, skip this step.

   a. Run the following from your command line to create an Amazon EKS Cluster with version 1.14 or above. Replace `<your-cluster-name>` with any name for your cluster.

```
eksctl create cluster --name <your-cluster-name> --region us-east-1 --auto-kubeconfig --timeout=50m --managed --nodes=1
```

   b. When cluster creation is complete, verify that you have access to the cluster using the following command.

```
kubectl get nodes
```

2. Verify that the current kubectl context is the cluster you want to use with the following command. The current context is marked with an asterisk (*) in the output.

```
kubectl config get-contexts
```

3. If the desired cluster is not configured as your current default, update the default with the following command.

```
aws eks update-kubeconfig --name <clustername> --region us-east-1
```

### Install Kubeflow Pipelines

Run the following steps from the command line of your gateway node to install Kubeflow Pipelines on your cluster.

1. Install Kubeflow Pipelines on your cluster by following step 1 of Deploying Kubeflow Pipelines documentation. Your KFP version must be 0.5.0 or above.

2. Verify that the Kubeflow Pipelines service and other related resources are running.

```
kubectl -n kubeflow get all | grep pipeline
```

Your output should look like the following.

<table>
<thead>
<tr>
<th>Name</th>
<th>Phase</th>
<th>Duration</th>
<th>Ready</th>
<th>Age</th>
</tr>
</thead>
<tbody>
<tr>
<td>pod/ml-pipeline-6b88c67994-kdtjv</td>
<td>Running</td>
<td>2d</td>
<td>1/1</td>
<td></td>
</tr>
<tr>
<td>pod/ml-pipeline-persistenceagent-64d74dfdbf-66stk</td>
<td>Running</td>
<td>2d</td>
<td>1/1</td>
<td></td>
</tr>
<tr>
<td>pod/ml-pipeline-scheduledworkflow-65bf46db7-5x9qj</td>
<td>Running</td>
<td>2d</td>
<td>1/1</td>
<td></td>
</tr>
<tr>
<td>pod/ml-pipeline-ui-66cc4cfdff5-cmsdb</td>
<td>Running</td>
<td>2d</td>
<td>1/1</td>
<td></td>
</tr>
<tr>
<td>pod/ml-pipeline-viewer-crd-6db65ccc4-wqlrj</td>
<td>Running</td>
<td>2d</td>
<td>1/1</td>
<td></td>
</tr>
<tr>
<td>pod/ml-pipeline-visualizationserver-9c47576f4-bqmx4</td>
<td>Running</td>
<td>2d</td>
<td>1/1</td>
<td></td>
</tr>
<tr>
<td>service/ml-pipeline</td>
<td>ClusterIP</td>
<td>10.100.170.170</td>
<td>&lt;none&gt;</td>
<td></td>
</tr>
<tr>
<td>8888/TCP, 8887/TCP</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>service/ml-pipeline-ui</td>
<td>ClusterIP</td>
<td>10.100.38.71</td>
<td>&lt;none&gt;</td>
<td></td>
</tr>
<tr>
<td>80/TCP</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>service/ml-pipeline-visualizationserver</td>
<td>ClusterIP</td>
<td>10.100.61.47</td>
<td>&lt;none&gt;</td>
<td></td>
</tr>
<tr>
<td>8888/TCP</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Access the KFP UI

The Kubeflow Pipelines UI is used for managing and tracking experiments, jobs, and runs on your cluster. You can use port forwarding to access the Kubeflow Pipelines UI from your gateway node.

Set up port forwarding to the KFP UI service

Run the following from the command line of your gateway node:

1. Verify that the KFP UI service is running using the following command:

   ```bash
   kubectl -n kubeflow get service ml-pipeline-ui
   ```

<table>
<thead>
<tr>
<th>NAME</th>
<th>TYPE</th>
<th>CLUSTER-IP</th>
<th>EXTERNAL-IP</th>
<th>PORT(S)</th>
<th>AGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>ml-pipeline-ui</td>
<td>ClusterIP</td>
<td>10.100.38.71</td>
<td>&lt;none&gt;</td>
<td>80/TCP</td>
<td>2d22h</td>
</tr>
</tbody>
</table>

2. Run the following command to set up port forwarding to the KFP UI service. This forwards the KFP UI to port 8080 on your gateway node and allows you to access the KFP UI from your browser.

   ```bash
   kubectl port-forward -n kubeflow service/ml-pipeline-ui 8080:80
   ```

   The port forward from your remote machine drops if there is no activity. Run this command again if your dashboard is unable to get logs or updates. If the commands return an error, ensure that there is no process already running on the port you are trying to use.

Access the KFP UI service

Your method of accessing the KFP UI depends on your gateway node type.

- Local machine as the gateway node

  1. Access the dashboard in your browser as follows:

     `http://localhost:8080`

  2. Choose **Pipelines** to access the pipelines UI.

- Amazon EC2 instance as the gateway node

  1. You need to set up an SSH tunnel on your Amazon EC2 instance to access the Kubeflow dashboard from your local machine’s browser.
From a new terminal session in your local machine, run the following. Replace `<public-DNS-of-gateway-node>` with the IP address of your instance found on the Amazon EC2 console. You can also use the public DNS. Replace `<path_to_key>` with the path to the pem key used to access the gateway node.

```
public_DNS_address=<public-DNS-of-gateway-node>
key=<path_to_key>
on Ubuntu:
ssh -i ${key} -L 9000:localhost:8080 ubuntu@${public_DNS_address}
or on Amazon Linux:
ssh -i ${key} -L 9000:localhost:8080 ec2-user@${public_DNS_address}
```

2. Access the dashboard in your browser.

```
http://localhost:9000
```

3. Choose Pipelines to access the KFP UI.

### Create IAM Users/Roles for KFP pods and the SageMaker service

You now have a Kubernetes cluster with Kubeflow set up. To run SageMaker Components for Kubeflow Pipelines, the Kubeflow Pipeline pods need access to SageMaker. In this section, you create IAM users/roles to be used by Kubeflow Pipeline pods and SageMaker.

#### Create a KFP execution role

Run the following from the command line of your gateway node:

1. Enable OIDC support on the Amazon EKS cluster with the following command. Replace `<cluster_name>` with the name of your cluster and `<cluster_region>` with the region your cluster is in.

   ```
   eksctl utils associate-iam-oidc-provider --cluster <cluster-name> \
   --region <cluster-region> --approve
   ```

2. Run the following to get the OIDC issuer URL. This URL is in the form `https://oidc.eks.<region>.amazonaws.com/id/<OIDC_ID>`.

   ```
   aws eks describe-cluster --region <cluster-region> --name <cluster-name> --query "cluster.identity.oidc.issuer" --output text
   ```

3. Run the following to create a file named `trust.json`. Replace `<OIDC_URL>` with your OIDC issuer URL. Don't include `https://` when in your OIDC issuer URL. Replace `<AWS_account_number>` with your AWS account number.

   ```
   OIDC_URL="<OIDC_URL>"
   AWS_ACC_NUM="<AWS-account-number>"
   # Run this to create trust.json file
   cat <<EOF > trust.json
   {
   "Version": "2012-10-17",
   "Statement": [
   {
   "Effect": "Allow",
   "Principal": {
   "Federated": "arn:aws:iam::${AWS_ACC_NUM}:oidc-provider/${OIDC_URL}"
   }
   }
   ```

1507
4. Create an IAM role named kfp-example-pod-role using trust.json using the following command. This role is used by KFP pods to create SageMaker jobs from KFP components. Note the ARN returned in the output.

```bash
aws iam create-role --role-name kfp-example-pod-role --assume-role-policy-document file://trust.json
aws iam attach-role-policy --role-name kfp-example-pod-role --policy-arn arn:aws:iam::aws:policy/AmazonSageMakerFullAccess
aws iam get-role --role-name kfp-example-pod-role --output text --query 'Role.Arn'
```

5. Edit your pipeline-runner service account with the following command.

```bash
kubectl edit -n kubeflow serviceaccount pipeline-runner
```

6. In the file, add the following Amazon EKS role annotation and replace <role_arn> with your role ARN.

```yaml
eks.amazonaws.com/role-arn: <role_arn>
```

7. Your file should look like the following when you've added the Amazon EKS role annotation. Save the file.

```yaml
apiVersion: v1
category: ServiceAccount
metadata:
  annotations:
    eks.amazonaws.com/role-arn: <role_arn>
  kubectl.kubernetes.io/last-applied-configuration: |
    |
      |
      |
      |
      |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
Create an SageMaker execution role

The `kfp-example-sagemaker-execution-role` IAM role is used by SageMaker jobs to access AWS resources. For more information, see the IAM Permissions section. You provide this role as an input parameter when running the pipeline.

Run the following to create the role. Note the ARN that is returned in your output.

```
SAGEMAKER_EXECUTION_ROLE_NAME=kfp-example-sagemaker-execution-role

aws iam create-role --role-name ${SAGEMAKER_EXECUTION_ROLE_NAME} --assume-role-policy-document "$TRUST"
aws iam attach-role-policy --role-name ${SAGEMAKER_EXECUTION_ROLE_NAME} --policy-arn arn:aws:iam::aws:policy/AmazonSageMakerFullAccess
aws iam attach-role-policy --role-name ${SAGEMAKER_EXECUTION_ROLE_NAME} --policy-arn arn:aws:iam::aws:policy/AmazonS3FullAccess
aws iam get-role --role-name ${SAGEMAKER_EXECUTION_ROLE_NAME} --output text --query 'Role.Arn'
```

Add access to additional IAM users or roles

If you use an intuitive IDE like Jupyter or want other people in your organization to use the cluster you set up, you can also give them access. The following steps run through this workflow using SageMaker notebooks. An SageMaker notebook instance is a fully managed Amazon EC2 compute instance that runs the Jupyter Notebook App. You use the notebook instance to create and manage Jupyter notebooks to create ML workflows. You can define, compile, deploy, and run your pipeline using the KFP Python SDK or CLI. If you're not using an SageMaker notebook to run Jupyter, you need to install the AWS CLI and the latest version of kubectl.

1. Follow the steps in Create an SageMaker Notebook Instance to create a SageMaker notebook instance if you do not already have one. Give the IAM role for this instance the S3FullAccess permission.
2. Amazon EKS clusters use IAM users and roles to control access to the cluster. The rules are implemented in a config map named `aws-auth`. Only the user/role that has access to the cluster will be able to edit this config map. Run the following from the command line of your gateway node to get the IAM role of the notebook instance you created. Replace `<instance-name>` with the name of your instance.

```
aws sagemaker describe-notebook-instance --notebook-instance-name <instance-name> --region <region> --output text --query 'RoleArn'
```

This command outputs the IAM role ARN in the `arn:aws:iam::<account-id>:role/<role-name>` format. Take note of this ARN.

3. Run the following to attach the policies the IAM role. Replace `<role-name>` with the `<role-name>` in your ARN.

```
aws iam attach-role-policy --role-name <role-name> --policy-arn arn:aws:iam::aws:policy/AmazonSageMakerFullAccess
aws iam attach-role-policy --role-name <role-name> --policy-arn arn:aws:iam::aws:policy/AmazonEKSWorkerNodePolicy
aws iam attach-role-policy --role-name <role-name> --policy-arn arn:aws:iam::aws:policy/AmazonS3FullAccess
```
4. `eksctl` provides commands to read and edit the `aws-auth config map`. `system:masters` is one of the default user groups. You add the user to this group. The `system:masters` group has super user permissions to the cluster. You can also create a group with more restrictive permissions or you can bind permissions directly to users. Replace `<IAM-Role-arn>` with the ARN of the IAM role. `<your_username>` can be any unique username.

```
eksctl create iamidentitymapping \
  --cluster <cluster-name> \
  --arn <IAM-Role-arn> \
  --group system:masters \
  --username <your-username> \
  --region <region>
```

5. Open the Jupyter notebook on your SageMaker instance and run the following to verify that it has access to the cluster.

```
aws eks --region <region> update-kubeconfig --name <cluster-name>
kubectl -n kubeflow get all | grep pipeline
```

**Running the Kubeflow Pipeline**

Now that setup of your gateway node and Amazon EKS cluster is complete, you can create your classification pipeline. To create your pipeline, you need to define and compile it. You then deploy it and use it to run workflows. You can define your pipeline in Python and use the KFP dashboard, KFP CLI, or Python SDK to compile, deploy, and run your workflows. The full code for the MNIST classification pipeline example is available in the Kubeflow Github repository. To use it, clone the example Python files to your gateway node.

**Prepare datasets**

To run the pipelines, you need to upload the data extraction pre-processing script to an Amazon S3 bucket. This bucket and all resources for this example must be located in the `us-east-1` Amazon Region. If you don’t have a bucket, create one using the steps in Creating a bucket.

From the `mnist-kmeans-sagemaker` folder of the Kubeflow repository you cloned on your gateway node, run the following command to upload the `kmeans_preprocessing.py` file to your Amazon S3 bucket. Change `<bucket-name>` to the name of the Amazon S3 bucket you created.

```
aws s3 cp mnist-kmeans-sagemaker/kmeans_preprocessing.py s3://<bucket-name>/mnist_kmeans_example/processing_code/kmeans_preprocessing.py
```

**Create a Kubeflow Pipeline using SageMaker Components**

The full code for the MNIST classification pipeline is available in the Kubeflow Github repository. To use it, clone the example Python files to your gateway node.

**Input Parameters**

The full MNIST classification pipeline has run-specific parameters for which you must provide values when creating a run. You must provide these parameters for each component of your pipeline. These parameters can also be updated when using other pipelines. We have provided default values for all parameters in the sample classification pipeline file.

The following are the only parameters you need to pass to run the sample pipelines. To pass these parameters, update their entries when creating a new run.

- **Role-ARN:** This must be the ARN of an IAM role that has full SageMaker access in your AWS account. Use the ARN of `kfp-example-pod-role`. 

1510
• **Bucket**: This is the name of the Amazon S3 bucket that you uploaded the `kmeans_preprocessing.py` file to.

You can adjust any of the input parameters using the KFP UI and trigger your run again.

**Compile and deploy your pipeline**

After defining the pipeline in Python, you must compile the pipeline to an intermediate representation before you can submit it to the Kubeflow Pipelines service. The intermediate representation is a workflow specification in the form of a YAML file compressed into a tar.gz file. You need the KFP SDK to compile your pipeline.

**Install KFP SDK**

Run the following from the command line of your gateway node:

1. Install the KFP SDK following the instructions in the Kubeflow pipelines documentation.
2. Verify that the KFP SDK is installed with the following command:
   ```
   pip show kfp
   ```
3. Verify that `dsl-compile` has been installed correctly as follows:
   ```
   which dsl-compile
   ```

**Compile your pipeline**

You have three options to interact with Kubeflow Pipelines: KFP UI, KFP CLI, or the KFP SDK. The following sections illustrate the workflow using the KFP UI and CLI.

Complete the following from your gateway node to compile your pipeline.

1. Modify your Python file with your Amazon S3 bucket name and IAM role ARN.
2. Use the `dsl-compile` command from the command line to compile your pipeline as follows.
   Replace `<path-to-python-file>` with the path to your pipeline and `<path-to-output>` with the location where you want your tar.gz file to be.
   ```
   dsl-compile --py <path-to-python-file> --output <path-to-output>
   ```

**Upload and run the pipeline using the KFP CLI**

Complete the following steps from the command line of your gateway node. KFP organizes runs of your pipeline as experiments. You have the option to specify an experiment name. If you do not specify one, the run will be listed under **Default** experiment.

1. Upload your pipeline as follows:
   ```
   kfp pipeline upload --pipeline-name <pipeline-name> <path-to-output.tar.gz>
   ```

   Your output should look like the following. Take note of the ID.
   ```
   Pipeline 29c3ff21-49f5-4dfe-94f6-618c0e2420fe has been submitted
   ```
2. Create a run using the following command. The KFP CLI run command currently does not support specifying input parameters while creating the run. You need to update your parameters in the Python pipeline file before compiling. Replace `<experiment-name>` and `<job-name>` with any names. Replace `<pipeline-id>` with the ID of your submitted pipeline. Replace `<your-role-arn>` with the ARN of kfp-example-pod-role. Replace `<your-bucket-name>` with the name of the Amazon S3 bucket you created.

```
kfp run submit --experiment-name <experiment-name> --run-name <job-name> --pipeline-id <pipeline-id> role_arn="<your-role-arn>" bucket_name="<your-bucket-name>"
```

You can also directly submit a run using the compiled pipeline package created as the output of the `dsl-compile` command.

```
kfp run submit --experiment-name <experiment-name> --run-name <job-name> --package-file <path-to-output> role_arn="<your-role-arn>" bucket_name="<your-bucket-name>"
```

Your output should look like the following:

```
Creating experiment aws.
Run 95084a2c-f18d-4b77-a9da-eba00bf01e63 is submitted
+--------------------------+----------+-------------------+-------------------------------+
| run id                  | name     | status            | created at                    |
|--------------------------+----------+-------------------+-------------------------------+
| 95084a2c-f18d-4b77-a9da-eba00bf01e63 | sm-job   |                    | 2020-04-30T20:36:41+00:00     |
+--------------------------+----------+-------------------+-------------------------------+
```

3. Navigate to the UI to check the progress of the job.

**Upload and run the pipeline using the KFP UI**

1. On the left panel, choose the **Pipelines** tab.
2. In the upper-right corner, choose **+UploadPipeline**.
3. Enter the pipeline name and description.
4. Choose **Upload a file** and enter the path to the tar.gz file you created using the CLI or with the Python SDK.
5. On the left panel, choose the **Pipelines** tab.
6. Find the pipeline you created.
7. Choose **+CreateRun**.
8. Enter your input parameters.
9. Choose **Run**.
Running predictions

Once your classification pipeline is deployed, you can run classification predictions against the endpoint that was created by the Deploy component. Use the KFP UI to check the output artifacts for `sagemaker-deploy-model-endpoint_name`. Download the .tgz file to extract the endpoint name or check the SageMaker console in the region you used.

Configure permissions to run predictions

If you want to run predictions from your gateway node, skip this section.

1. To use any other machine to run predictions, assign the `sagemaker:InvokeEndpoint` permission to the IAM role or IAM user used by the client machine. This permission is used to run predictions.
2. On your gateway node, run the following to create a policy file:

   ```
   cat <<EoF > ./sagemaker-invoke.json
   {
   "Version": "2012-10-17",
   "Statement": [
   {
   "Effect": "Allow",
   "Action": ["sagemaker:InvokeEndpoint"],
   "Resource": "*"
   }
   ]
   }
   EoF
   ``

3. Attach the policy to the client node’s IAM role or IAM user.
4. If your client machine has an IAM role attached, run the following. Replace `<your-instance-IAM-role>` with the name of the client node’s IAM role. Replace `<path-to-sagemaker-invoke-json>` with the path to the policy file you created.

   ```
   aws iam put-role-policy --role-name <your-instance-IAM-role> --policy-name sagemaker-invoke-for-worker --policy-document file://<path-to-sagemaker-invoke-json>
   ```

5. If your client machine has IAM user credentials configured, run the following. Replace `<your_IAM_user_name>` with the name of the client node’s IAM user. Replace `<path-to-sagemaker-invoke-json>` with the path to the policy file you created.

   ```
   aws iam put-user-policy --user-name <your-IAM-user-name> --policy-name sagemaker-invoke-for-worker --policy-document file://<path-to-sagemaker-invoke-json>
   ```

Run predictions

1. Create a Python file from your client machine named `mnist-predictions.py` with the following content. Replace the `ENDPOINT_NAME` and `REGION` variables. This script loads the MNIST dataset, then creates a CSV from those digits and sends it to the endpoint for prediction. It then outputs the results.

   ```python
   import pickle, gzip, numpy, urllib.request, json
   from urllib.parse import urlparse
   import json
   import io
   import boto3
   ```
ENDPOINT_NAME='\'<endpoint-name>\'\'
REGION = '\'<region>\'\'

# Load the dataset
urllib.request.urlretrieve("http://deeplearning.net/data/mnist/mnist.pkl.gz", "mnist.pkl.gz")
with gzip.open('mnist.pkl.gz', 'rb') as f:
    train_set, valid_set, test_set = pickle.load(f, encoding='latin1')

# Simple function to create a csv from our numpy array
def np2csv(arr):
    csv = io.BytesIO()
    numpy.savetxt(csv, arr, delimiter=',', fmt='%g')
    return csv.getvalue().decode().rstrip()

runtime = boto3.Session(region_name=REGION).client('sagemaker-runtime')
payload = np2csv(train_set[0][30:31])
response = runtime.invoke_endpoint(EndpointName=ENDPOINT_NAME, ContentType='text/csv', Body=payload)
result = json.loads(response['Body'].read().decode())
print(result)

2. Run the Python file as follows:

    python mnist-predictions.py

View results and logs

When the pipeline is running, you can choose any component to check execution details, such as inputs and outputs. This lists the names of created resources.

If the KFP request is successfully processed and an SageMaker job is created, the component logs in the KFP UI provide a link to the job created in SageMaker. The CloudWatch logs are also provided if the job is successfully created.

If you run too many pipeline jobs on the same cluster, you may see an error message that indicates you do not have enough pods available. To fix this, log in to your gateway node and delete the pods created by the pipelines you are not using as follows:

    kubectl get pods -n kubeflow
    kubectl delete pods -n kubeflow <name-of-pipeline-pod>

Cleanup

When you're finished with your pipeline, you need to clean up your resources.

1. From the KFP dashboard, terminate your pipeline runs if they do not exit properly by choosing Terminate.
2. If the Terminate option doesn't work, log in to your gateway node and manually terminate all the pods created by your pipeline run as follows:

    kubectl get pods -n kubeflow
    kubectl delete pods -n kubeflow <name-of-pipeline-pod>

3. Using your AWS account, log in to the SageMaker service. Manually stop all training, batch transform, and HPO jobs. Delete models, data buckets, and endpoints to avoid incurring any additional costs. Terminating the pipeline runs does not stop the jobs in SageMaker.
Using Amazon Augmented AI for Human Review

This feature is not available in the China Regions.

When you use AI applications such as Amazon Rekognition, Amazon Textract, or your custom machine learning (ML) models, you can use Amazon Augmented AI to get human review of low confidence predictions or a random sample of predictions.

What is Amazon Augmented AI?

Amazon Augmented AI (Amazon A2I) makes it easy to build the workflows required for human review of ML predictions. Amazon A2I brings human review to all developers, removing the undifferentiated heavy lifting associated with building human review systems or managing large numbers of human reviewers.

Many machine learning applications require humans to review low-confidence predictions to ensure the results are correct. For example, extracting information from scanned mortgage application forms can require human review in some cases due to low-quality scans or poor handwriting. Building human review systems can be time-consuming and expensive because it involves implementing complex processes or workflows, writing custom software to manage review tasks and results, and in many cases, managing large groups of reviewers.

Amazon A2I makes it easy to build and manage human reviews for machine learning applications. Amazon A2I provides built-in human review workflows for common machine learning use cases, such as content moderation and text extraction from documents, which allows predictions from Amazon Rekognition and Amazon Textract to be reviewed easily. You can also create your own workflows for ML models built on SageMaker or any other tools. Using Amazon A2I, you can allow human reviewers to step in when a model is unable to make a high-confidence prediction or to audit its predictions on an ongoing basis.

Amazon A2I Use Case Examples

The following are examples of how you can use Amazon A2I to integrate a human review loop into your ML application. For each of these examples, you can find a Jupyter Notebook that demonstrates that workflow in Use Cases and Examples using Amazon A2I (p. 1534).

- **Use Amazon A2I with Amazon Textract** – Have humans review single page documents to review important form key value pairs, or have Amazon Textract randomly sample and send documents from your dataset to humans for review.
- **Use Amazon A2I with Amazon Rekognition** – Have humans review unsafe images for explicit adult or violent content if Amazon Rekognition returns a low confidence score, or have Amazon Rekognition randomly sample and send images from your dataset to humans for review.
- **Use Amazon A2I to review real time ML inferences** – Use Amazon A2I to review real time, low-confidence inferences made by a model deployed to an SageMaker hosted endpoint and incrementally train your model using Amazon A2I output data.
- **Use Amazon A2I with Amazon Comprehend** – Have humans review Amazon Comprehend inferences about text data such as sentiment analysis, text syntax, and entity detection.
- **Use Amazon A2I with Amazon Transcribe** – Have humans review Amazon Transcribe transcriptions of video or audio files. Use the results of transcription human review loops to create a custom vocabulary and improve future transcriptions of similar video or audio content.
- **Use Amazon A2I with Amazon Translate** – Have humans review low-confidence translations returned from Amazon Translate.
- **Use Amazon A2I to review tabular data** – Use Amazon A2I to integrate a human review loop into an ML application that uses tabular data.
Get Started with Amazon Augmented AI

To get started using Amazon Augmented AI, review the Core Components of Amazon A2I (p. 1516) and Prerequisites to Using Augmented AI (p. 1518). Then, use the following to learn how to use the Amazon A2I console and API.

- Demo: Get Started in the Amazon A2I Console (p. 1518)
- Demo: Get Started Using the Amazon A2I API (p. 1524)

Core Components of Amazon A2I

Review the following terms to familiarize yourself with the core components of Amazon A2I.

Task Types

The AI/ML workflow into which you integrate Amazon A2I defines an Amazon A2I task type.

Amazon A2I supports two built-in task types: Amazon Textract and Amazon Rekognition, and a custom task type.

You can use a custom task type to integrate Amazon A2I with other AWS services like Amazon Comprehend, Amazon Transcribe, and Amazon Translate, as well as your own custom machine learning workflows. To learn more, see Use Cases and Examples using Amazon A2I (p. 1534).
Human review workflow (flow definition)

You use a human review workflow to specify your human work team, to set up your worker UI using a worker task template, and to provide information about how workers should complete the review task.

For built-in task types, you also use the human review workflow to identify the conditions under which a human loop is triggered. For example, Amazon Rekognition can perform image content moderation using machine learning. You can use the human review workflow to specify that an image will be sent to a human for content moderation review if Amazon Rekognition’s confidence is too low.

You can use a human review workflow to create multiple human loops.

You can create a flow definition in the SageMaker console or with the SageMaker API. To learn more about both of these options, see Create a Human Review Workflow (p. 1540).

Work Team

A work team is a group of human workers to whom you send your human review tasks.

When you create a human review workflow, you specify a single work team.

Your work team can come from the Amazon Mechanical Turk workforce, a vendor-managed workforce, or your own private workforce. When you use the private workforce, you can create multiple work teams. Each work team can be used in multiple human review workflows. To learn how to create a workforce and work teams, see Create and Manage Workforces (p. 427).

Worker Task Template and Human Task UI

You use a worker task template to create a worker UI that is used for your human review tasks. The worker UI is referred to as a human task UI.

The human task UI displays your input data, such as documents or images, and instructions to workers. It also provides interactive tools that the worker uses to complete your tasks.

For built-in task types, you must use a the Amazon A2I worker task template provided for that task type.

Human Loops

A human loop is used to create a single human review job. For each human review job, you can choose the number of workers that are sent a task to review a single data object. For example, if you set the number of workers per object to 3 for an image classification labeling job, three workers will classify each input image. Increasing the number of workers per object can improve label accuracy.

A human loop is created using a human review workflow as follows:

- For built-in task types, the conditions specified in the human review workflow determine when the human loop is created.
- Human review tasks are sent to the work team specified in the human review workflow.
- The worker task template specified in the human review workflow is used to render the human task UI.

When do human loops get created?

When you use one of the built-in task types, the corresponding AWS service creates and starts a human loop on your behalf when the conditions specified in your human review workflow are met. For example:

- When you use Augmented AI with Amazon Textract, you can integrate Amazon A2I into a document review task using the API operation AnalyzeDocument. A human loop will get created everytime Amazon Textract returns inferences about key value pairs that meet the conditions you specify in your human review workflow.
When you use Augmented AI with Amazon Rekognition, you can integrate Amazon A2I into an image moderation task using the API operation `DetectModerationLabels`. A human loop will get created every time Amazon Rekognition returns inferences about image content that meet the conditions you specify in your human review workflow.

When using a *custom task type*, you start a human loop using the Amazon Augmented AI Runtime API. When you call `StartHumanLoop` in your custom application, a task is sent to human reviewers.

To learn how to create and start a human loop, see **Create and Start a Human Loop (p. 1558)**.

To generate these resources and create a human review workflow, Amazon A2I integrates multiple APIs including the Amazon Augmented AI Runtime Model, the SageMaker APIs, and APIs associated with your task type. To learn more, see **Use APIs in Amazon Augmented AI (p. 1591)**.

**Note**

AWS Region availability may differ when you use Augmented AI with other AWS services, such as Amazon Textract. Create Augmented AI resources in the same region that you use to interact with those AWS services. For AWS Region availability for all services, see the Region Table.

### Prerequisites to Using Augmented AI

Amazon A2I uses resources in IAM, SageMaker, and Amazon S3 to create and execute your human review workflows. To use Amazon A2I, you need the following resources. You can create some of these resources in the Amazon A2I console when you create a human review workflow. To learn how, see **Demo: Get Started in the Amazon A2I Console (p. 1518)**.

- One or more Amazon S3 buckets in the same AWS Region as the workflow for your input and output data. To create a bucket, follow the instructions in **Create a Bucket in the Amazon Simple Storage Service Console User Guide**.
- An IAM role with required permissions to create a human review workflow and an IAM user or role with permission to access Augmented AI. For more information, see **Permissions and Security in Amazon Augmented AI (p. 1584)**.
- A public, private, or vendor workforce for your human review workflows. If you plan to use a private workforce, you need to set one up ahead of time in the same AWS Region as your Amazon A2I workflow. To learn more about these workforce types, see **Create and Manage Workforces (p. 427)**.

**Important**

Click here to see the compliance programs that cover Amazon Augmented AI at this time. If you use Amazon Augmented AI in conjunction with other AWS services (such as Amazon Rekognition and Amazon Textract), please note that Amazon Augmented AI may not be in scope for the same compliance programs as those other services. You are responsible for how you use Amazon Augmented AI, including understanding how the service will process or store customer data, and any impact on the compliance of your data environment. You should discuss your workload objectives and goals with your AWS account team; they can help you evaluate whether the service is a good fit for your proposed use case and architecture.

### Demo: Get Started in the Amazon A2I Console

The following demo shows you how to get started using Amazon A2I in the Amazon A2I console.

The demo gives you the option to use Augmented AI with Amazon Textract for document review or Amazon Rekognition for image content review.

### Prerequisites

To get started using Amazon A2I, complete the following prerequisites.
• Create an Amazon S3 bucket in the same AWS Region as the workflow for your input and output data. For example if you are using Amazon A2I with Amazon Textract in us-east-1, create your bucket in us-east-1. To create a bucket, follow the instructions in Create a Bucket in the Amazon Simple Storage Service Console User Guide.

• Do one of the following:
  • If you want to complete the demo using Amazon Textract download this sample document and place it in your Amazon S3 bucket.
  • If you want to complete the demo using Amazon Rekognition, download the following image and place it in your Amazon S3 bucket.

Note
The Amazon A2I console is embedded in the SageMaker console.

Step 1: Create a Work Team

First, create a work team in the Amazon A2I console and add yourself as a worker so that you can preview the worker review task.

Important
This walkthrough uses a private work team. The Amazon A2I private workforce is configured in the Ground Truth area of the SageMaker console and is shared between Amazon A2I and Ground Truth.

To create a private workforce using worker emails

2. In the navigation pane, choose Labeling workforces under Ground Truth.
3. Choose Private, then choose Create private team.
4. Choose Invite new workers by email.
5. For this demo, enter your email and any others that you want to be able to preview the human task UI. You can paste or type a list of up to 50 email addresses, separated by commas, into the email addresses box.
6. Enter an organization name and contact email.
7. Optionally, choose an SNS topic to which to subscribe the team so workers are notified by email when new Ground Truth labeling jobs become available. Amazon SNS notifications are supported by Ground Truth and are not supported by Augmented AI. If you subscribe workers to SNS notifications, they only receive notifications about Ground Truth labeling jobs. They do not receive notifications about Augmented AI tasks.
8. Click the Create private team button.

If you add yourself to a private work team, you receive an email from no-reply@verificationemail.com with login information. Use the link in this email to reset your password and log in to your worker portal. This is where your human review tasks appear when you create a human loop.

Step 2: Create a Human Review Workflow

In this step, you create a human review workflow. Each human review workflow is created for a specific task type. This demo allows you to choose between the built-in task types: Amazon Rekognition and Amazon Textract.
To create a human review workflow:

1. Open the Augmented AI console at https://console.aws.amazon.com/a2i to access the Human review workflows page.
2. Select Create human review workflow.
3. In Workflow settings enter a workflow Name, S3 bucket, and the IAM role that you created for this demo, with the AWS managed policy AmazonAugmentedAIIntegratedAPIAccess attached.
4. For Task type select Textract - Key-value pair extraction or Rekognition - Image moderation.
5. Select the task type that you chose from the following table for instructions for that task type.

Amazon Textract - Key-value pair extraction

1. Select Trigger a human review for specific form keys based on the form key confidence score or when specific form keys are missing.
2. For Key name, enter Mail Address.
3. Set the identification confidence threshold between 0 and 99.
4. Set the qualification confidence threshold between 0 and 99.
5. Select Trigger a human review for all form keys identified by Amazon Textract with confidence scores in a specific range.
6. Set the identification confidence threshold between 0 and 90.
7. Set the qualification confidence threshold between 0 and 90.

This triggers a human review if Amazon Textract returns a confidence score that is less than 99 for Mail Address and its key, or if it returns a confidence score less than 90 for any key value pair detected in the document.
Amazon Textract form extraction - Conditions for invoking human review

1. Select Trigger human review for labels identified by Amazon Rekognition based on label confidence score.

2. Set the Threshold between 0 and 98.

This triggers a human review if Amazon Rekognition returns a confidence score that is less than 98 for an image moderation job.

Amazon Rekognition - Image moderation

1. Select Trigger human review for labels identified by Amazon Rekognition based on label confidence score.

2. Set the Threshold between 0 and 98.

This triggers a human review if Amazon Rekognition returns a confidence score that is less than 98 for an image moderation job.

6. Under Worker task template creation, select Create from a default template.

7. Enter a Template name.
8. In **Task description**, copy and paste the following into the text box:

   *Read the instructions carefully and complete the task.*

9. Under **Workers**, select **Private**.

10. Choose the private team that you created.

11. Select **Create**.

Once your human review workflow is created, it appears in the table on the **Human review workflows** page. When the **Status** is **Active**, copy and save the **Workflow ARN**. You will need this for the next step.

### Step 3: Start a Human Loop

You must use an API operation to start a human loop. There are a variety of language-specific SDKs that you can use to interact with these API operations. To see documentation for each of these SDKs, refer to the **See Also** section in the API documentation. For example:

For this demo, you will use one of the following APIs:

- If you chose the Amazon Textract task type, you will use the **AnalyzeDocument** operation.
- If you chose the Amazon Rekognition task type, you will use the **DetectModerationLabels** operation.

You can interact with these APIs using an SageMaker notebook instance (recommended for new users) or the AWS Command Line Interface (AWS CLI). Choose one of the following to learn more about these options:

- To learn more about and set up a notebook instance, see **Use Amazon SageMaker Notebook Instances** (p. 124).
- To learn more about and get started using the AWS CLI, see **What Is the AWS Command Line Interface?** in the AWS Command Line Interface User Guide.

Select your task type in the following table to see example requests for Amazon Textract and Amazon Rekognition using the AWS SDK for Python (Boto3).
Amazon Textract - Key-value pair extraction

The following example uses the AWS SDK for Python (Boto3) to call `analyze_document` in us-west-2. Replace the red, italicized text with your resources. Include the `DataAttributes` parameter if you are using the Amazon Mechanical Turk workforce. For more information, see `analyze_document` documentation in the AWS SDK for Python (Boto) API Reference.

```python
response = client.analyze_document(
    Document={"S3Object": {"Bucket": "AWSDOC-EXAMPLE-BUCKET", "Name": "document-name.pdf"},
        HumanLoopConfig={
            "HumanLoopName": "human-loop-name",
            "DataAttributes": {
                "ContentClassifiers": ["FreeOfPersonallyIdentifiableInformation", "FreeOfAdultContent"]
            }
        }
    },
    FeatureTypes=["FORMS"]
)
```

Amazon Rekognition - Image moderation

The following example uses the AWS SDK for Python (Boto3) to call `detect_moderation_labels` in us-west-2. Replace the red, italicized text with your resources. Include the `DataAttributes` parameter if you are using the Amazon Mechanical Turk workforce. For more information, see `detect_moderation_labels` documentation in the AWS SDK for Python (Boto) API Reference.

```python
response = client.detect_moderation_labels(
    Image={"S3Object": {"Bucket": "AWSDOC-EXAMPLE-BUCKET", "Name": "image-name.png"}},
    HumanLoopConfig={
            "HumanLoopName": "human-loop-name",
            "DataAttributes": {
                "ContentClassifiers": ["FreeOfPersonallyIdentifiableInformation", "FreeOfAdultContent"]
            }
        }
)
```

**Step 4: View Human Loop Status in Console**

When you start a human loop, you can view its status in the Amazon A2I console.

**To View your human loop status**

1. Open the Augmented AI console at [https://console.aws.amazon.com/a2i](https://console.aws.amazon.com/a2i) to access the Human review workflows page.
2. Select the human review workflow that you used to start your human loop.
3. In the Human loops section, you can see your human loop. View its status in the Status column.

**Step 5: Download Output Data**

Your output data is stored in the Amazon S3 bucket you specified when you created a human review workflow.

**To view your Amazon A2I output data**

1. Open the Amazon S3 console: [s3.console.aws.amazon.com](http://s3.console.aws.amazon.com).
2. Select the Amazon S3 bucket you specified when you created your human review workflow in step 2 of this demo.

3. Starting with the folder that is named after your human review workflow, navigate to your output data by clicking through the folders with the following naming convention:

   s3://output-bucket-specified-in-human-review-workflow/human-review-workflow-name/YYYY/MM/DD/hh/mm/ss/human-loop-name/output.json

4. Select output.json and select Download.

**Demo: Get Started Using the Amazon A2I API**

This walkthrough explains the API operations you can use to get started using Amazon A2I.

To use a Jupyter Notebook to execute these operations, select a Jupyter notebook from Use Cases and Examples using Amazon A2I (p. 1534) and use Use SageMaker Notebook Instance with Amazon A2I Jupyter Notebook (p. 1536) to learn how to use it in SageMaker notebook instance.

To learn more about the API operations you can use with Amazon A2I, see Use APIs in Amazon Augmented AI (p. 1591).

**Create a Private Work Team**

You can create a private work team and add yourself as a worker so that you can preview Amazon A2I.

If you are not familiar with Amazon Cognito, we recommend that you use the SageMaker console to create a private workforce and add yourself as a private worker. For instructions, see Step 1: Create a Work Team (p. 1519).

If you are familiar with Amazon Cognito you can use the following instructions to create a private work team using the SageMaker API. After you create a work team, note down the work team ARN (WorkteamArn).

To learn more about the private workforce, and other available configurations, see Use a Private Workforce (p. 430).

**Create a private workforce**

If you have not created a private workforce, you can do so using an Amazon Cognito user pool. Make sure that you have added yourself to this user pool. You can create a private work team using the AWS Python SDK (Boto3) function, create_workforce. For other language-specific SDKs, refer to the list in CreateWorkforce.

```python
response = client.create_workforce(
    CognitoConfig={
        "UserPool": "Pool_ID",
        "ClientId": "app-client-id"
    },
    WorkforceName="workforce-name"
)
```

**Create a private work team**

After you have created a private workforce in the AWS Region to configure and start your human loop, you can create a private work team using the AWS Python SDK (Boto3) function, create_workteam. For other language-specific SDKs, refer to the list in CreateWorkteam.
response = client.create_workteam(
    WorkteamName="work-team-name",
    WorkforceName= "workforce-name",
    MemberDefinitions=[
        {
            "CognitoMemberDefinition": {
                "UserPool": "<aws-region>_ID",
                "UserGroup": "user-group",
                "ClientId": "app-client-id"
            }
        }
    ]
)

Access your work team ARN as follows:

workteamArn = response["WorkteamArn"]

List private work teams in your account

If you have already created a private work team, you can list all work teams in a given AWS Region in your account using the AWS Python SDK (Boto3) function, list_workteams. For other language-specific SDKs, refer to the list in ListWorkteams.

response = client.list_workteams()

If you have numerous work teams in your account, you may want to use MaxResults, SortBy, and NameContains to filter your results.

Create a Human Review Workflow

You can create a human review workflow using the Amazon A2I operation CreateFlowDefinition. Before you create your human review workflow, you need to create a human task UI. You can do this with the CreateHumanTaskUi operation.

If you are using Amazon A2I with the Amazon Textract or Amazon Rekognition integrations, you can specify activation conditions using a JSON.

Create a Human Task UI

If you are creating a human review workflow to be used with Amazon Textract or Amazon Rekognition integrations, you need to use and modify pre-made worker task template. For all custom integrations, you can use your own, custom worker task template. Use the following table to see how to create a human task UI using a worker task template for the two built-in integrations. Replace the template with your own to customize this request.

Amazon Textract- Key-value pair extraction

To learn more about this template, see Custom Template Example for Amazon Textract (p. 1567).

template = r""
<script src="https://assets.crowd.aws/crowd-html-elements.js"></script>
{% capture s3_uri %}http://s3.amazonaws.com/{{ task.input.aiServiceRequest.document.s3Object.bucket }}/{{ task.input.aiServiceRequest.document.s3Object.name }}{% endcapture %}
<crowd-form>
Click on a key-value block to highlight the corresponding key-value pair in the document.

If it is a valid key-value pair, review the content for the value. If the content is incorrect, correct it.

The text of the value is incorrect, correct it.

A wrong value is identified, correct it.

If it is not a valid key-value relationship, choose No.

If you can't find the key in the document, choose Key not found.

Key and value displayed in one line.

Key and value displayed in two lines.

If the content of the value has multiple lines, enter all the text without line break. Include all value text even if it extends beyond the highlight box.

**Examples**

Key and value are often displayed next or below to each other.

If the content of a field is empty, choose Value is blank.

Key and value are often displayed next or below to each other.

If you can't find the key in the document, choose Key not found.

Key and value displayed in one line.

Key and value displayed in two lines.
<crowd-form>
  <crowd-rekognition-detect-moderation-labels
    categories='[
      {% for label in task.input.selectedAiServiceResponse.moderationLabels %}
        {
          name: "{{ label.name }}",
          parentName: "{{ label.parentName }}",
        },
      {% endfor %}
    ]'
    src="{{ s3_arn | grant_read_access }}"
    header="Review the image and choose all applicable categories."
  >
  <short-instructions header="Instructions">  
    <style>
      .instructions {
        white-space: pre-wrap;
      }
    </style>
    <p class="instructions">Review the image and choose all applicable categories. If no categories apply, choose None.
    </p>
    <b>Nudity</b>
    Visuals depicting nude male or female person or persons
    <b>Partial Nudity</b>
    Visuals depicting covered up nudity, for example using hands or pose
    <b>Revealing Clothes</b>
    Visuals depicting revealing clothes and poses
    <b>Physical Violence</b>
    Visuals depicting violent physical assault, such as kicking or punching
    <b>Weapon Violence</b>
    Visuals depicting violence using weapons like firearms or blades, such as shooting
    <b>Weapons</b>
    Visuals depicting weapons like firearms and blades
  </short-instructions>
  <full-instructions header="Instructions"></full-instructions>
</crowd-rekognition-detect-moderation-labels>
</crowd-form>

Custom Integration

The following is an example template that can be used in a custom integration. This template is used in this notebook, demonstrating a custom integration with Amazon Comprehend.

```template = r""
<script src="https://assets.crowd.aws/crowd-html-elements.js"></script>

<crowd-form>
  <crowd-classifier
    name="sentiment"
    categories=["Positive", "Negative", "Neutral", "Mixed"]
    initial-value="{{ task.input.initialValue }}"
    header="What sentiment does this text convey?"
  >
  <classification-target>
    {{ task.input.taskObject }}
  </classification-target>
</crowd-form>""```
Using the template specified above, you can create a template using the AWS Python SDK (Boto3) function, `create_human_task_ui`. For other language-specific SDKs, refer to the list in `CreateHumanTaskUi`.

```python
response = client.create_human_task_ui(
    HumanTaskUiName="human-task-ui-name",
    UiTemplate={
        "Content": template
    }
)
```

This response element contains the human task UI ARN. Save this as follows:

```python
humanTaskUiArn = response["HumanTaskUiArn"]
```

**Create JSON to specify activation conditions**

For Amazon Textract and Amazon Rekognition built-in integrations, you can save activation conditions in a JSON object and use this in your `CreateFlowDefinition` request.

Next, select a tab to see example activation conditions you can use for these built-in integrations. For additional information about activation condition options, see JSON Schema for Human Loop Activation Conditions in Amazon Augmented AI (p. 1545).

**Amazon Textract- Key-value pair extraction**

This example specifies conditions for specific keys (such as `Mail address`) in the document. If Amazon Textract's confidence falls outside of the thresholds set here, the document is sent to a human for review, with the specific keys that triggered the human loop prompted to the worker.

```python
import json

humanLoopActivationConditions = json.dumps(
    {  
        "Conditions": [
            {  
                "Or": [
                    {  
                        "ConditionType": "ImportantFormKeyConfidenceCheck",  
                        "ConditionParameters": {
                            "Key": "Mail address",  
                            "MinConfidence": 0.8,  
                            "MaxConfidence": 0.98
                        }
                    }
                ]
            }
        ]
    }
)
```
import json

humanLoopActivationConditions = json.dumps(
    {
        "Conditions": [
            {
                "Or": [
                    {
                        "ConditionType": "ModerationLabelConfidenceCheck",
                        "ConditionParameters": {
                            "ImportantFormKey": "*",
                            "KeyValueBlockConfidenceGreaterThan": 0,
                            "WordBlockConfidenceGreaterThan": 0
                        }
                    }
                ]
            }
        ]
    }
)

Amazon Rekognition - Image moderation

The human loop activation conditions used here are tailored towards Amazon Rekognition content moderation; they are based on the confidence thresholds for particular moderation labels, Suggestive and Female Swimwear Or Underwear.
Create a human review workflow

This section gives an example of the CreateFlowDefinition AWS Python SDK (Boto3) request using the resources created in the previous sections. For other language-specific SDKs, refer to the list in CreateFlowDefinition. Use the tabs in the following table to see the requests to create a human review workflow for Amazon Textract and Amazon Rekognition built-in integrations.

**Amazon Textract - Key-value pair extraction**

If you use the built-in integration with Amazon Textract, you must specify "AWS/Textract/AnalyzeDocument/Forms/V1" for "AwsManagedHumanLoopRequestSource" in HumanLoopRequestSource.

```python
response = client.create_flow_definition(
    FlowDefinitionName="human-review-workflow-name",
    HumanLoopRequestSource={
        "AwsManagedHumanLoopRequestSource": "AWS/Textract/AnalyzeDocument/Forms/V1"
    },
    HumanLoopActivationConfig={
        "HumanLoopActivationConditionsConfig": {
            "HumanLoopActivationConditions": humanLoopActivationConditions
        }
    },
    HumanLoopConfig={
        "WorkteamArn": workteamArn,
        "HumanTaskUiArn": humanTaskUiArn,
        "TaskTitle": "Document entry review",
        "TaskDescription": "Review the document and instructions. Complete the task",
        "TaskCount": 1,
        "TaskAvailabilityLifetimeInSeconds": 43200,
        "TaskTimeLimitInSeconds": 3600,
        "TaskKeywords": [
            "document review",
        ],
    },
    OutputConfig={
        "S3OutputPath": "s3://DOC-EXAMPLE-BUCKET/prefix/",
    },
    RoleArn="arn:aws:iam::<account-number>:role/<role-name>",
    Tags=[
        "Key": "string",
        "Value": "string"
    ]
)
```
Amazon Rekognition - Image moderation

If you use the built-in integration with Amazon Rekognition, you must specify "AWS/Rekognition/DetectModerationLabels/Image/V3" for "AwsManagedHumanLoopRequestSource" in HumanLoopRequestSource.

```python
response = client.create_flow_definition(
    FlowDefinitionName="human-review-workflow-name",
    HumanLoopRequestSource={
        "AwsManagedHumanLoopRequestSource": "AWS/Rekognition/DetectModerationLabels/Image/V3"
    },
    HumanLoopActivationConfig={
        "HumanLoopActivationConditionsConfig": {
            "HumanLoopActivationConditions": humanLoopActivationConditions
        }
    },
    HumanLoopConfig={
        "WorkteamArn": workteamArn,
        "HumanTaskUiArn": humanTaskUiArn,
        "TaskTitle": "Image content moderation",
        "TaskDescription": "Review the image and instructions. Complete the task",
        "TaskCount": 1,
        "TaskAvailabilityLifetimeIn Seconds": 43200,
        "TaskTimeLimitInSeconds": 3600,
        "TaskKeywords": [
            "content moderation",
        ],
    },
    OutputConfig={
        "S3OutputPath": "s3://DOC-EXAMPLE-BUCKET/prefix/"
    },
    RoleArn="arn:aws:iam::<account-number>:role/<role-name>",
    Tags=[
        {
            "Key": "string",
            "Value": "string"
        }
    ]
)
```

Custom Integration

If you use a custom integration, exclude the following parameters: HumanLoopRequestSource, HumanLoopActivationConfig.

```python
response = client.create_flow_definition(
    FlowDefinitionName="human-review-workflow-name",
    HumanLoopConfig={
        "WorkteamArn": workteamArn,
        "HumanTaskUiArn": humanTaskUiArn,
        "TaskTitle": "Image content moderation",
        "TaskDescription": "Review the image and instructions. Complete the task",
        "TaskCount": 1,
        "TaskAvailabilityLifetimeInSeconds": 43200,
        "TaskTimeLimitInSeconds": 3600,
        "TaskKeywords": [
```
After you create a human review workflow, you can retrieve the flow definition ARN from the response:

```python
humanReviewWorkflowArn = response["FlowDefinitionArn"]
```

## Create a Human Loop

The API operation you use to start a human loop depends on the Amazon A2I integration you use.

- If you use the Amazon Textract built-in integration, you use the `AnalyzeDocument` operation.
- If you use the Amazon Rekognition built-in integration, you use the `DetectModerationLabels` operation.
- If you use a custom integration, you use the `StartHumanLoop` operation.

Select your task type in the following table to see example requests for Amazon Textract and Amazon Rekognition using the AWS SDK for Python (Boto3).

### Amazon Textract- Key-value pair extraction

The following example uses the AWS SDK for Python (Boto3) to call `analyze_document` in us-west-2. Replace the red, italicized text with your resources. Include the `DataAttributes` parameter if you are using the Amazon Mechanical Turk workforce. For more information, see `analyze_document` documentation in the AWS SDK for Python (Boto) API Reference.

```python
response = client.analyze_document(
    Document="S3Object": {"Bucket": "AWSDOC-EXAMPLE-BUCKET", "Name": "document-name.pdf"},
    "HumanLoopName": "human-loop-name",
    "DataAttributes": {
        "ContentClassifiers": ["FreeOfPersonallyIdentifiableInformation", "FreeOfAdultContent"]
    }
)
FeatureTypes=['FORMS']
```

Human loops are only created if Amazon Textract's confidence for document analysis task meets the activation conditions you specified in your human review workflow. You can check the `response` element to determine if a human loop has been created. To see everything included in this response, see `HumanLoopActivationOutput`.
if "HumanLoopArn" in analyzeDocumentResponse["HumanLoopActivationOutput"]:  
    # A human loop has been started!  
    print(f"A human loop has been started with ARN: {analyzeDocumentResponse["HumanLoopActivationOutput"]['HumanLoopArn']}")

Amazon Rekognition - Image moderation

The following example uses the AWS SDK for Python (Boto3) to call `detect_moderation_labels` in us-west-2. Replace the red, italicized text with your resources. Include the `DataAttributes` parameter if you are using the Amazon Mechanical Turk workforce. For more information, see `detect_moderation_labels` documentation in the AWS SDK for Python (Boto) API Reference.

```
response = client.detect_moderation_labels(
    Image={'S3Object': {'Bucket': 'AWSDOC-EXAMPLE-BUCKET', 'Name': 'image-name.png'}},
    HumanLoopConfig={
        "HumanLoopName": "human-loop-name",
        "DataAttributes": {
            "ContentClassifiers": ['FreeOfPersonallyIdentifiableInformation', 'FreeOfAdultContent']
        }
    }
)
```

Human loops are only created if Amazon Rekognition's confidence for an image moderation task meets the activation conditions you specified in your human review workflow. You can check the `response` element to determine if a human loop has been created. To see everything included in this response, see `HumanLoopActivationOutput`.

```
if "HumanLoopArn" in response["HumanLoopActivationOutput"]:  
    # A human loop has been started!  
    print(f"A human loop has been started with ARN: {response["HumanLoopActivationOutput"]['HumanLoopArn']}")
```

Custom Integration

The following example uses the AWS SDK for Python (Boto3) to call `start_human_loop` in us-west-2. Replace the red, italicized text with your resources. Include the `DataAttributes` parameter if you are using the Amazon Mechanical Turk workforce. For more information, see `start_human_loop` documentation in the AWS SDK for Python (Boto) API Reference.

```
response = client.start_human_loop(
    HumanLoopName= "human-loop-name",
    HumanLoopInput={"InputContent": inputContentJson},
    DataAttributes={"ContentClassifiers": ["FreeOfPersonallyIdentifiableInformation","FreeOfAdultContent"]}
)
```

This example stores input content in the variable `inputContentJson`. Assume that the input content contains two elements: a text blurb and sentiment (such as Positive, Negative, or Neutral), and it is formatted as follows:
inputContent = {
    "initialValue": sentiment,
    "taskObject": blurb
}

The keys `initialValue` and `taskObject` must correspond to the keys used in the liquid elements of our worker task template. Refer to the custom template in Create a Human Task UI (p. 1525) to see an example.

To create `inputContentJson`, do the following.

```python
import json

inputContent = {
    "initialValue": sentiment,
    "taskObject": blurb
}

inputContentJson = json.dumps(inputContent)
```

A human loop starts each time you call `start_human_loop`. To check the status of your human loop, use `describe_human_loop`:

```python
human_loop_info = a2i.describe_human_loop(HumanLoopName="human_loop_name")
print(f"HumanLoop Status: {human_loop_info["HumanLoopStatus"]}")
print(f"HumanLoop Output Destination: {human_loop_info["HumanLoopOutput"]}")
```

Use Cases and Examples using Amazon A2I

You can use Amazon Augmented AI to incorporate a human review into your workflow for built-in task types, Amazon Textract and Amazon Rekognition, or your own custom tasks using a custom task type.

When you create a human review workflow using one of the built-in task types, you can specify conditions, such as confidence thresholds, that trigger a human review. The service (Amazon Rekognition or Amazon Textract) creates a human loop on your behalf when these conditions are met and supplies your input data directly to Amazon A2I to send to human reviewers. To learn more about the built-in task types, use the following:

- Use Amazon Augmented AI with Amazon Textract (p. 1536)
- Use Amazon Augmented AI with Amazon Rekognition (p. 1538)

When you use a custom task type, you create and start a human loop using the Amazon A2I Runtime API. Use the custom task type to incorporate a human review workflow with other AWS services or your own custom ML application.

- For more details, see Use Amazon Augmented AI with Custom Task Types (p. 1539)

The following table outlines a variety of Amazon A2I use cases that you can explore using SageMaker Jupyter notebooks. To get started with a Jupyter notebook, use the instructions in Use SageMaker Notebook Instance with Amazon A2I Jupyter Notebook (p. 1536). For more examples, see this GitHub repository.

<table>
<thead>
<tr>
<th>Use Case</th>
<th>Description</th>
<th>Task Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use Amazon A2I with Amazon Textract</td>
<td>Have humans review single-page documents to review important</td>
<td>Built-in</td>
</tr>
<tr>
<td>Use Case</td>
<td>Description</td>
<td>Task Type</td>
</tr>
<tr>
<td>----------------------------------------------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>-----------</td>
</tr>
<tr>
<td><strong>Use Amazon A2I with Amazon Textract</strong></td>
<td>form key value pairs, or have Amazon Textract randomly sample and send documents from your dataset to humans for review.</td>
<td></td>
</tr>
<tr>
<td><strong>Use Amazon A2I with Amazon Rekognition</strong></td>
<td>Have humans review unsafe images for explicit adult or violent content if Amazon Rekognition returns a low confidence score, or have Amazon Rekognition randomly sample and send images from your dataset to humans for review.</td>
<td>Built-in</td>
</tr>
<tr>
<td><strong>Use Amazon A2I with Amazon Comprehend</strong></td>
<td>Have humans review Amazon Comprehend inferences about text data such as sentiment analysis, text syntax, and entity detection.</td>
<td>Custom</td>
</tr>
<tr>
<td><strong>Use Amazon A2I with Amazon Transcribe</strong></td>
<td>Have humans review Amazon Transcribe transcriptions of video or audio files. Use the results of transcription human review loops to create a custom vocabulary and improve future transcriptions of similar video or audio content.</td>
<td>Custom</td>
</tr>
<tr>
<td><strong>Use Amazon A2I with Amazon Translate</strong></td>
<td>Have humans review low-confidence translations returned from Amazon Translate.</td>
<td>Custom</td>
</tr>
<tr>
<td><strong>Use Amazon A2I to review real time ML inferences</strong></td>
<td>Use Amazon A2I to review real time, low-confidence inferences made by a model deployed to an SageMaker hosted endpoint and incrementally train your model using Amazon A2I output data.</td>
<td>Custom</td>
</tr>
<tr>
<td><strong>Use Amazon A2I to review tabular data</strong></td>
<td>Use Amazon A2I to integrate a human review loop into an ML application that uses tabular data.</td>
<td>Custom</td>
</tr>
</tbody>
</table>

**Topics**
- Use SageMaker Notebook Instance with Amazon A2I Jupyter Notebook (p. 1536)
- Use Amazon Augmented AI with Amazon Textract (p. 1536)
- Use Amazon Augmented AI with Amazon Rekognition (p. 1538)
- Use Amazon Augmented AI with Custom Task Types (p. 1539)
Use SageMaker Notebook Instance with Amazon A2I Jupyter Notebook

For an end-to-end example that demonstrates how to integrate an Amazon A2I human review loop into a machine learning workflow, you can use a Jupyter notebook from this GitHub Repository in an SageMaker Notebook instance.

To use an Amazon A2I custom task type sample notebook in an Amazon SageMaker notebook Instance:

1. If you do not have an active SageMaker notebook instance, create one by following the instructions in Step 2: Create an Amazon SageMaker Notebook Instance (p. 60).
2. When your notebook instance is active, choose Open JupyterLab to the right of the notebook instance's name. It may take a few moments for JupyterLab to load.
3. Choose the icon to clone a GitHub repository into your workspace.
4. Enter the amazon-a2i-sample-jupyter-notebooks repository HTTPS URL.
5. Choose CLONE.
6. Open the notebook that you would like to run.
7. Follow the instructions in the notebook to configure your human review workflow and human loop and run the cells.
8. To avoid incurring unnecessary charges, when you are done with the demo, stop and delete your notebook instance in addition to any Amazon S3 buckets, IAM roles, and CloudWatch Events resources created during the walkthrough.

Use Amazon Augmented AI with Amazon Textract

Amazon Textract enables you to add document text detection and analysis to your applications. Amazon Augmented AI (Amazon A2I) directly integrated with Amazon Textract's AnalyzeDocument API operation. You can use AnalyzeDocument to analyze a document for relationships between detected items. When you add an Amazon A2I human review loop to an AnalyzeDocument request, Amazon A2I monitors the Amazon Textract results and sends a document to one or more human workers for review when the conditions specified in your flow definition are met. For example, if you want a human to review a specific key like "Full name:" and their associated input-values you can create a trigger to start a human review anytime the key "Full name:" is detected or when the inference confidence for that key falls within a range that you specify.

You can specify when Amazon Textract sends a task to a human worker for review when creating a human review workflow, or flow definition by specifying activation conditions.

You can set the following activation conditions when using the Amazon Textract task type:

- Trigger a human review for specific form keys based on the form key confidence score.
- Trigger a human review when specific form keys are missing.
- Trigger human review for all form keys identified by Amazon Textract with confidence scores in a specified range.
- Randomly send a sample of forms to humans for review.

When your activation condition depends on form key confidence scores, you can use two types of prediction-confidence to trigger human loops:
• **Identification confidence** – The confidence score for key-value pairs detected within a form.

• **Qualification confidence** – The confidence score for text contained within key and value respectively in a form.

In the image in the following section, **Full Name: Jane Doe** is the key-value pair, and **Full Name** and **Jane Doe** are the key and value respectively.

You can set these activation conditions using the Amazon SageMaker console when you create a human review workflow, or by creating a JSON for human loop activation conditions and specifying this as input in the `HumanLoopActivationConditions` parameter of `CreateFlowDefinition` API operation. To learn how to specify activation conditions in JSON format, see [JSON Schema for Human Loop Activation Conditions in Amazon Augmented AI](p. 1545) and [Use Human Loop Activation Conditions JSON Schema with Amazon Textract](p. 1546).

**Note**

When using Augmented AI with Amazon Textract, create Augmented AI resources in the same AWS Region you use to call `AnalyzeDocument`.

**Get Started: Integrate a Human Review into an Amazon Textract Analyze Document Job**

To integrate a human review into an Amazon Textract text detection and analysis job, you need to create a flow definition, and then use the Amazon Textract API to integrate that flow definition into your workflow. To learn how to create a flow definition using the SageMaker console or Augmented AI API, see the following topics:

- Create a Human Review Workflow (Console) (p. 1541)
- Create a Human Review Workflow (API) (p. 1543)

After you've created your flow definition, see [Using Augmented AI with Amazon Textract](p. 1546) to learn how to integrate your flow definition into your Amazon Textract task.

**End-to-end Demo Using Amazon Textract and Amazon A2I**

For an end-to-end example that demonstrates how to use Amazon Textract with Amazon A2I using the console, see [Demo: Get Started in the Amazon A2I Console](p. 1518).

To learn how to use the Amazon A2I API to create and start a human review, you can use Amazon Augmented AI (Amazon A2I) integration with Amazon Textract’s Analyze Document [Example] in an SageMaker Notebook instance. To get started, see [Use SageMaker Notebook Instance with Amazon A2I Jupyter Notebook](p. 1536).

**A2I Textract Worker Console Preview**

When they're assigned a review task in an Amazon Textract workflow, workers might see UI similar to the following:
You can customize this UI in the SageMaker console when you create your human review definition, or by creating and using a custom template. To learn more, see Create and Manage Worker Task Templates (p. 1564).

Use Amazon Augmented AI with Amazon Rekognition

Amazon Rekognition makes it easy to add image analysis to your applications. The Amazon Rekognition DetectModerationLabels API operation is directly integrated with Amazon A2I so that you can easily create a human loop to review unsafe images, such as explicit adult or violent content. You can use DetectModerationLabels to configure a human loop using a flow definition ARN. This enables Amazon A2I to analyze predictions made by Amazon Rekognition and send results to a human for review if they meet the conditions set in your flow definition.

You can set the following activation conditions when using the Amazon Rekognition task type:

- Trigger human review for labels identified by Amazon Rekognition based on the label confidence score.
- Randomly send a sample of images to humans for review.

You can set these activation conditions using the Amazon SageMaker console when you create a human review workflow, or by creating a JSON for human loop activation conditions and specifying this as input in the HumanLoopActivationConditions parameter of CreateFlowDefinition API operation. To learn how specify activation conditions in JSON format, see JSON Schema for Human Loop Activation Conditions in Amazon Augmented AI (p. 1545) and Use Human Loop Activation Conditions JSON Schema with Amazon Rekognition (p. 1552).

Note

When using Augmented AI with Amazon Rekognition, create Augmented AI resources in the same AWS Region you use to call DetectModerationLabels.

Get Started: Integrate a Human Review into an Amazon Rekognition Image Moderation Job

To integrate a human review into an Amazon Rekognition, see the following topics:

- Create a Human Review Workflow (Console) (p. 1541)
- Create a Human Review Workflow (API) (p. 1543)
After you've created your flow definition, see Using Augmented AI with Amazon Rekognition to learn how to integrate your flow definition into your Amazon Rekognition task.

**End-to-end Demo Using Amazon Rekognition and Amazon A2I**

For an end-to-end example that demonstrates how to use Amazon Rekognition with Amazon A2I using the console, see Demo: Get Started in the Amazon A2I Console (p. 1518).

To learn how to use the Amazon A2I API to create and start a human review, you can use Amazon Augmented AI (Amazon A2I) integration with Amazon Rekognition [Example] in an SageMaker Notebook instance. To get started, see Use SageMaker Notebook Instance with Amazon A2I Jupyter Notebook (p. 1536).

**A2I Rekognition Worker Console Preview**

When they're assigned a review task in an Amazon Rekognition workflow, workers might see UI similar to the following:

![Worker UI Preview](image)

You can customize this interface in the SageMaker console when you create your human review definition, or by creating and using a custom template. To learn more, see Create and Manage Worker Task Templates (p. 1564).

**Use Amazon Augmented AI with Custom Task Types**

You can use Amazon Augmented AI (Amazon A2I) to incorporate a human review (human loop) into any machine learning workflow using the custom task type. This option gives you the most flexibility to customize the conditions under which your data objects are sent to humans for review, as well as the look and feel of your worker UI.

When you use a custom task type, you create a custom human review workflow, and you specify the conditions under which a data object is sent for human review directly in your application.

Use the procedures on this page to learn how to integrate Amazon A2I into any machine learning workflow using the custom task type.

**To create a human loop using a flow definition, integrate it into your application, and monitor the results**

1. Complete the Amazon A2I Prerequisites to Using Augmented AI (p. 1518). Note the following:
Create a Human Review Workflow

Use an Amazon Augmented AI (Amazon A2I) human review workflow, or flow definition, to specify the following:

- For the Amazon Textract and Amazon Rekognition built-in task types, the conditions under which your human loop will be called.
- The workforce that your tasks will be sent to.
- The instructions that your workforce will receive, which is called a worker task template.

1. Create a Human Review Workflow

   Use an Amazon Augmented AI (Amazon A2I) human review workflow, or flow definition, to specify the following:

   - The path the Amazon Simple Storage Service (Amazon S3) bucket or buckets where you will store your input and output data.
   - The Amazon Resource Name (ARN) of an AWS Identity and Access Management (IAM) role with required permissions attached.
   - (Optional) If you plan to use a private workforce, the ARN of your workforce.

2. Using HTML elements, create a custom worker template which Amazon A2I uses to generate your worker task UI. To learn how to create a custom template, see Create Custom Worker Task Template (p. 1566).

3. Use the custom worker template from Step 2 to generate a worker task template in the Amazon SageMaker console. To learn how, see Create a Worker Task Template (p. 1565).

   In the next Step you will create a flow definition:

   - If you want to create a flow definition using the SageMaker API, note the ARN of this worker task template for the next step.
   - If you are creating a flow definition using the console, your template will automatically appear in Worker task template section when you choose Create human review workflow.

4. When creating your flow definition, provide the path to your S3 buckets, your IAM role ARN, and your worker template.

   - Learn how to create a flow definition using the SageMaker API: Create a Human Review Workflow (API) (p. 1543).
   - Learn how to create a flow definition using the SageMaker console: Create a Human Review Workflow (Console) (p. 1541).

5. Configure your human loop using the Amazon A2I Runtime API. To learn how, see Create and Start a Human Loop (p. 1558).

6. To control when human reviews are initiated in your application, specify conditions under which StartHumanLoop is called in your application. Human loop activation conditions, such as confidence thresholds that trigger the human loop, are not available when using Amazon A2I with custom task types. Every StartHumanLoop invocation results in a human review.

   Once you have started a human loop, you can manage and monitor your loops using the Amazon Augmented AI Runtime API and Amazon EventBridge (also known as Amazon CloudWatch Events). To learn more, see Monitor and Manage Your Human Loop (p. 1574).

End-to-end Demo Using Amazon A2I Custom Task Types

For an end-to-end examples that demonstrates how to integrate Amazon A2I into a variety of ML workflows, see the table in Use Cases and Examples using Amazon A2I (p. 1534). To get started using one of these notebooks, see Use SageMaker Notebook Instance with Amazon A2I Jupyter Notebook (p. 1536).
- The configuration of your worker tasks, including the number of workers that receive a task and time limits to complete tasks.
- Where your output data will be stored.

You can create a human review workflow in the SageMaker console or using the SageMaker `CreateFlowDefinition` operation. You can build a worker task template using the console for Amazon Textract and Amazon Rekognition task types while creating your flow definition.

**Important**

Human loop activation conditions, which trigger the human loop — for example, confidence thresholds — aren't available for Amazon A2I custom task types. When using the console to create a flow definition for a custom task type, you can't specify activation conditions. When using the Amazon A2I API to create a flow definition for a custom task type, you can't set the `HumanLoopActivationConditions` attribute of the `HumanLoopActivationConditionsConfig` parameter. To control when human reviews are initiated, specify conditions under which `StartHumanLoop` is called in your custom application. In this case, every `StartHumanLoop` invocation results in a human review. For more information, see Use Amazon Augmented AI with Custom Task Types (p. 1539).

**Prerequisites**

To create a human review workflow definition, you must have completed the prerequisites described in Prerequisites to Using Augmented AI (p. 1518).

If you use the API to create a flow definition for any task type, or if you use a custom task type when creating a flow definition in the console, first you will need to create a worker task template. For more information, see Create and Manage Worker Task Templates (p. 1564).

If you want to preview your worker task template while creating a flow definition for a built-in task type in the console, ensure that you grant the role that you use to create the flow definition permission to access the Amazon S3 bucket that contains your template artifacts using a policy like the one described in Enable Worker Task Template Previews (p. 1588).

**Topics**

- Create a Human Review Workflow (Console) (p. 1541)
- Create a Human Review Workflow (API) (p. 1543)
- JSON Schema for Human Loop Activation Conditions in Amazon Augmented AI (p. 1545)

**Create a Human Review Workflow (Console)**

Use this procedure to create a Amazon Augmented AI (Amazon A2I) human review workflow using the SageMaker console. If you are new to Amazon A2I, we recommend that you create a private work team using people in your organization, and use this work team's ARN when creating your flow definition. To learn how to set up a private workforce and create a work team, see Create a Private Workforce (Amazon SageMaker Console) (p. 431). If you have already set up a private workforce, see Create a Work Team Using the SageMaker Console (p. 434) to learn how to add a work team to that workforce.

If you are using Amazon A2I with one of the built-in task types, you can create worker instructions using a default worker task template provided by Augmented AI while creating a human review workflow in the console. To see samples of the default templates provided by Augmented AI, see the built-in task types in Use Cases and Examples using Amazon A2I (p. 1534).

**To create flow definition (console)**

2. In the navigation pane, under the Augmented AI section, choose Human review workflows and then choose Create human review workflow.

3. In Overview, do the following:
   a. For Name, enter a unique workflow name. The name must be lowercase, unique within the AWS Region in your account, and can have up to 63 characters. Valid characters include: a-z, 0-9, and - (hyphen).
   b. For S3 location for output, enter the S3 bucket where you want to store the human review results. The bucket must be located in the same AWS Region as the workflow.
   c. For IAM role, choose the role that has the required permissions. If you choose a built-in task type and want to preview your worker template in the console, provide a role with the type of policy described in Enable Worker Task Template Previews (p. 1588) attached.

4. For Task type, choose the task type that you want the human worker to perform.

5. If you chose the Amazon Rekognition or Amazon Textract task type, specify the conditions that will invoke human review.
   - For Amazon Rekognition image moderation tasks, choose an inference confidence score threshold interval that triggers human review.
   - For Amazon Textract tasks, you can trigger a human review when specific form keys are missing or when form key detection confidence is low. You can also trigger a human review if, after evaluating all of the form keys in the text, confidence is lower than your required threshold for any form key. You will see two variables that you can use to specify your confidence thresholds: Identification confidence and Qualification confidence. To learn more about these variables, see Use Amazon Augmented AI with Amazon Textract (p. 1536).
   - For both task types, you can randomly send a percentage of data objects (images or forms) and their labels to humans for review.

6. Configure and specify your worker task template:
   a. If you are using the Amazon Rekognition or Amazon Textract task type:
      - In the Create template section:
         - To create instructions for your workers using the Amazon A2I default template for Amazon Rekognition and Amazon Textract task types, choose Build from a default template.
         - If you choose Build from a default template, create your instructions under Worker task design:
            - Provide a Template name that is unique in the AWS Region you are in.
            - In the Instructions section, provide detailed instructions on how to complete your task. To help workers achieve greater accuracy, provide good and bad examples.
            - (Optional) In Additional instructions, provide your workers with additional information and instructions.

            For information on creating effective instructions, see Creating Good Worker Instructions (p. 1573).

            - To select a custom template that you've created, choose it from the Template menu and provide a Task description to briefly describe the task for your workers. To learn how to create a custom template, see Create a Worker Task Template (p. 1565).

   b. If you are using the custom task type:
      - In the Worker task template section, choose your template from the list. All of the templates that you have created in the SageMaker console appear in this list. To learn how to create a template for custom task types, see Create and Manage Worker Task Templates (p. 1564).
7. (Optional) Preview your worker template:

   For Amazon Rekognition and Amazon Textract task types, you have the option to choose see a sample worker task to preview your worker task UI.

   If you are creating a flow definition for a custom task type, you can preview your worker task UI using the RenderUiTemplate operation. For more information, see Preview a Worker Task Template (p. 1572).

8. For Workers, choose a workforce type.

9. Choose Create.

Next Steps

After you've created a human review workflow, it appears in the console under Human review workflows. To see your flow definition's Amazon Resource Name (ARN) and configuration details, choose the workflow by selecting its name.

If you are using a built-in task type, you can use the flow definition ARN to start a human loop using that AWS service's API (for example, the Amazon Textract API). For custom task types, you can use the ARN to start a human loop using the Amazon Augmented AI Runtime API. To learn more about both options, see Create and Start a Human Loop (p. 1558).

Create a Human Review Workflow (API)

To create a flow definition using the SageMaker API, you use the CreateFlowDefinition operation. After you complete the Prerequisites to Using Augmented AI (p. 1518), use the following procedure to learn how to use this API operation.

For an overview of the CreateFlowDefinition operation, and details about each parameter, see CreateFlowDefinition.

To create a flow definition (API)

1. For FlowDefinitionName, enter a unique name. The name must be unique within the AWS Region in your account, and can have up to 63 characters. Valid characters include: a-z, 0-9, and - (hyphen).
2. For RoleArn, enter the ARN of the role that you configured to grant access to your data sources.
3. For HumanLoopConfig, enter information about the workers and what they should see. For information about each parameter in HumanLoopConfig, see HumanLoopConfig.
4. (Optional) If you are using a built-in task type, provide conditions that trigger a human loop in HumanLoopActivationConfig. To learn how to create the input required for the HumanLoopActivationConfig parameter, see JSON Schema for Human Loop Activation Conditions in Amazon Augmented AI (p. 1545). If you do not specify conditions here, when you provide a flow definition to the AWS service associated with a built-in task type (for example, Amazon Textract or Amazon Rekognition), that service will send every task to a human worker for review.

   If you are using a custom task type, HumanLoopActivationConfig is disabled. To learn how to control when tasks are sent to human workers using a custom task type, see Use Amazon Augmented AI with Custom Task Types (p. 1539).
5. (Optional) If you are using a built-in task type, specify the integration source (for example, Amazon Rekognition or Amazon Textract) in the HumanLoopRequestSource parameter.
6. For OutputConfig, indicate where in Amazon Simple Storage Service (Amazon S3) to store the output of the human loop.
7. (Optional) Use Tags to enter key value pairs to help you categorize and organize a flow definition. Each tag consists of a key and a value, both of which you define.
The following is an example of a request to create an Amazon Rekognition human loop using the AWS Python SDK (Boto3). Replace 'AWS/Rekognition/DetectModerationLabels/Image/V3' with 'AWS/Textract/AnalyzeDocument/Forms/V1' to create a Amazon Textract human loop. For more information, see the Boto 3 Augmented AI Runtime documentation.

```python
response = client.create_flow_definition(
    FlowDefinitionName='string',
    HumanLoopRequestSource={
        'AwsManagedHumanLoopRequestSource': 'AWS/Rekognition/DetectModerationLabels/Image/V3',
    },
    HumanLoopActivationConfig={
        'HumanLoopActivationConditionsConfig': {
            'HumanLoopActivationConditions': 'string',
        },
    },
    HumanLoopConfig={
        'WorkteamArn': 'string',
        'HumanTaskUiArn': 'string',
        'TaskTitle': 'string',
        'TaskDescription': 'string',
        'TaskCount': 123,
        'TaskAvailabilityLifetimeInSeconds': 123,
        'TaskTimeLimitInSeconds': 123,
        'TaskKeywords': ['string'],
        'PublicWorkforceTaskPrice': {
            'AmountInUsd': {
                'Dollars': 123,
                'Cents': 123,
                'TenthFractionsOfACent': 123,
            },
        },
    },
    OutputConfig={
        'S3OutputPath': 'string',
        'KmsKeyId': 'string',
    },
    RoleArn='string',
    Tags=[
        {
            'Key': 'string',
            'Value': 'string',
        },
    ],
)
```

Next Steps

The return value of a successful call of the CreateFlowDefinition API operation is a flow definition Amazon Resource Name (ARN).

If you are using a built-in task type, you can use the flow definition ARN to start a human loop using that AWS service's API (i.e. the Amazon Textract API). For custom task types, you can use the ARN to start a human loop using the Amazon Augmented AI Runtime API. To learn more about both of these options, see Create and Start a Human Loop (p. 1558).
The HumanLoopActivationConditions is an input parameter of the CreateFlowDefinition API. This parameter is a JSON-formatted string. The JSON models the conditions under which a human loop is created, when those conditions are evaluated against the response from an integrating AI service API (such as Rekognition.DetectModerationLabels or Textract.AnalyzeDocument). This response is referred to as an inference. For example, Amazon Rekognition sends an inference of a moderation label with an associated confidence score. In this example, the inference is the model's best estimate of the appropriate label for an image. For Amazon Textract, inference is made on the association between blocks of text (key-value pairs), such as the association between Name: and Sue in a form as well as content within a block of text, or word block, such as ‘Name’.

The following is the schema for the JSON. At the top level, the HumanLoopActivationConditions has a JSON array, Conditions. Each member of this array is an independent condition that, if evaluated to true, will result in Amazon A2I creating a human loop. Each such independent condition can be a primitive condition or a complex condition. A simple condition has the following attributes:

- **ConditionType**: This attribute identifies the type of condition. Each AWS AI service API that integrates with Amazon A2I defines its own set of allowed ConditionTypes.

  - Rekognition DetectModerationLabels – This API supports the ModerationLabelConfidenceCheck and Sampling ConditionType values.
  
  - Textract AnalyzeDocument – This API supports the ImportantFormKeyConfidenceCheck, MissingImportantFormKey and Sampling ConditionType values.

- **ConditionParameters**: This is a JSON object that parameterizes the condition. The set of allowed attributes of this object is dependent on the value of the ConditionType. Each ConditionType defines its own set of ConditionParameters.

A member of the Conditions array can model a complex condition. This is accomplished by logically connecting primitive conditions using the And and Or logical operators and nesting the underlying primitive conditions. Up to two levels of nesting are supported.

```json
{  
"$schema": "http://json-schema.org/draft-07/schema#",  
"definitions": {  
  "Condition": {  
    "type": "object",  
    "properties": {  
      "ConditionType": {  
        "type": "string"  
      },  
      "ConditionParameters": {  
        "type": "object"  
      }  
    },  
    "required": [  
      "ConditionType"  
    ]  
  },  
  "OrConditionArray": {  
    "type": "object",  
    "properties": {  
      "Or": {  
        "type": "array",  
        "minItems": 2,  
        "items": {  
          "#ref": "#/definitions/ComplexCondition"  
        }  
      }  
    }  
  }  
}
```
Note

Human loop activation conditions aren't available for human review workflows that are integrated with custom task types. The HumanLoopActivationConditions parameter is disabled for custom task types.

Topics

- Use Human Loop Activation Conditions JSON Schema with Amazon Textract (p. 1546)
- Use Human Loop Activation Conditions JSON Schema with Amazon Rekognition (p. 1552)

Use Human Loop Activation Conditions JSON Schema with Amazon Textract

When used with Amazon A2I, the AnalyzeDocument operation supports the following inputs in the ConditionType parameter:

- ImportantFormKeyConfidenceCheck – Use this condition to create a human loop when inference confidence is within a specified range for document form keys and word blocks. A form key is any word in a document that is associated with an input. The input is called a value. Together, form keys and values are referred to as key-value pairs. A word block refers to the words that Amazon Textract
recognizes inside of a detected block of text. To learn more about Amazon Textract document blocks, see Documents and Block Objects in the Amazon Textract Developer Guide.

- **MissingImportantFormKey** – Use this condition to create a human loop when Textract did not identify the key or its associated aliases within the document.
- **Sampling** – Use this condition to specify a percentage of forms to send to humans for review, regardless of inference confidence scores. Use this condition to do the following:
  - Audit your ML model by randomly sampling all forms analyzed by your model and sending a specified percentage to humans for review.
  - Using the ImportantFormKeyConfidenceCheck condition, randomly sample a percentage of the inferences that met the conditions specified in ImportantFormKeyConfidenceCheck to start a human loop and send only the specified percentage to humans for review.

**Note**
If you send the same request to AnalyzeDocument multiple times, the result of Sampling will not change for the inference of that input. For example, if you make an AnalyzeDocument request once, and Sampling doesn't trigger a HumanLoop, subsequent requests to AnalyzeDocument with the same configuration will not trigger a human loop.

### ImportantFormKeyConfidenceCheck Inputs and Results

The ImportantFormKeyConfidenceCheck ConditionType supports the following ConditionParameters:

- **ImportantFormKey** – A string representing a key in a key-value pair detected by Amazon Textract that needs to be reviewed by human workers. If the value of this parameter is the special catch-all value (*), then all keys are considered to be matched to the condition. You can use this to model the case where any key-value pair satisfying certain confidence thresholds needs human review.
- **ImportantFormKeyAliases** – An array that represents alternate spellings or logical equivalents for the important form key.
- **KeyValueBlockConfidenceEquals**
- **KeyValueBlockConfidenceLessThan**
- **KeyValueBlockConfidenceLessThanEquals**
- **KeyValueBlockConfidenceGreaterThan**
- **KeyValueBlockConfidenceGreaterThanEquals**
- **WordBlockConfidenceEquals**
- **WordBlockConfidenceLessThan**
- **WordBlockConfidenceLessThanEquals**
- **WordBlockConfidenceGreaterThan**
- **WordBlockConfidenceGreaterThanEquals**

When you use the ImportantFormKeyConfidenceCheck ConditionType, Amazon A2I sends the key-value block and word block inferences of the key-value blocks and associated aliases that you specified in ImportantFormKey and ImportantFormKeyAliases for human review.

When creating a flow definition, if you use the default worker task template that is provided in the Human review workflows section of the Amazon SageMaker console, key-value and block inferences sent for human review by this activation condition are included in the worker UI. If you use a custom worker task template, you need to include the `{{ task.input.selectedAiServiceResponse.blocks }}` element to include initial-value input data (inferences) from Amazon Textract. For an example of a custom template that uses this input element, see Custom Template Example for Amazon Textract (p. 1567).
MissingImportantFormKey Inputs and Results

The MissingImportantFormKey ConditionType supports the following ConditionParameters:

- **ImportantFormKey** – A string representing a key in a key-value pair detected by Amazon Textract that needs to be reviewed by human workers.
- **ImportantFormKeyAliases** – An array that represents alternate spellings or logical equivalents for the important form key.

When you use the MissingImportantFormKey ConditionType, if the key in ImportantFormKey or aliases in ImportantFormKeyAliases are not included in Amazon Textract inference, that form will be sent to human for review and no predicted key-value pairs will be included. For example, if Amazon Textract only identified Address and Phone in a form, but was missing the ImportantFormKey Name (in the MissingImportantFormKey condition type) that form would be sent to humans for review without any of the form keys detected (Address and Phone).

If you use the default worker task template that is provided in the SageMaker console, a task will be created asking workers identify the key in ImportantFormKey and associated value. If you use a custom worker task template, you need to include the `<task.input.humanLoopContext>` custom HTML element to configure this task.

Sampling Inputs and Results

The Sampling ConditionType supports the RandomSamplingPercentage ConditionParameters. The input for RandomSamplingPercentage must be real number between 0.01 and 100. This number represents the percentage of data that qualifies for a human review and will be sent to humans for review. If you use the Sampling condition without any other conditions, this number represents the percentage of all resulting inferences made by the AnalyzeDocument operation from a single request that will be sent to humans for review.

If you specify the Sampling condition without any other condition type, all key-value and block inferences are sent to workers for review.

When creating a flow definition, if you use the default worker task template that is provided in the Human review workflows section of the SageMaker console, all key-value and block inferences sent for human review by this activation condition are included in the worker UI. If you use a custom worker task template, you need to include the `{{ task.input.selectedAiServiceResponse.blocks }}` element to include initial-value input data (inferences) from Amazon Textract. For an example of a custom template that uses this input element, see Custom Template Example for Amazon Textract (p. 1567).

Examples

While only one condition needs to evaluate to true to trigger a human loop, Amazon A2I will evaluate all conditions for each object analyzed by Amazon Textract. The human reviewers are asked to review the important form keys for all the conditions that evaluated to true.

**Example 1: Detect important form keys with confidence scores in a specified range trigger HumanLoop**

Following is an example of a HumanLoopActivationConditions JSON that triggers a HumanLoop if any one of the following three conditions is met:

- Textract AnalyzeDocument API returns a key-value pair whose key is one of Employee Name, Name, or EmployeeName, with the confidence of the key-value block being less than 60 and the confidences of each of the word blocks making up the key and value being less than 85.
• Textract AnalyzeDocument API returns a key-value pair whose key is one of Pay Date, PayDate, DateOfPay, or pay-date, with the confidence of the key-value block being less than 65 and the confidences of each of the Word blocks making up the key and value being less than 85.

• Textract AnalyzeDocument API returns a key-value pair whose key is one of Gross Pay, GrossPay, or GrossAmount, with the confidence of the key-value block being less than 60 and the confidences of each of the word blocks making up the key and value being less than 85.

```json
{
    "Conditions": [
        {
            "ConditionType": "ImportantFormKeyConfidenceCheck",
            "ConditionParameters": {
                "ImportantFormKey": "Employee Name",
                "ImportantFormKeyAliases": [
                    "Name",
                    "EmployeeName"
                ],
                "KeyValueBlockConfidenceLessThan": 60,
                "WordBlockConfidenceLessThan": 85
            }
        },
        {
            "ConditionType": "ImportantFormKeyConfidenceCheck",
            "ConditionParameters": {
                "ImportantFormKey": "Pay Date",
                "ImportantFormKeyAliases": [
                    "PayDate",
                    "DateOfPay",
                    "pay-date"
                ],
                "KeyValueBlockConfidenceLessThan": 65,
                "WordBlockConfidenceLessThan": 85
            }
        },
        {
            "ConditionType": "ImportantFormKeyConfidenceCheck",
            "ConditionParameters": {
                "ImportantFormKey": "Gross Pay",
                "ImportantFormKeyAliases": [
                    "GrossPay",
                    "GrossAmount"
                ],
                "KeyValueBlockConfidenceLessThan": 60,
                "WordBlockConfidenceLessThan": 85
            }
        }
    ]
}
```

**Example 2: Use ImportantFormKeyConfidenceCheck**

In the following example, if Amazon Textract detects a key-value pair whose confidence for the key-value block is less than 60 and is less than 90 for any underlying word blocks, a HumanLoop is created. The human reviewers are asked to review all the form key-value pairs that matched the confidence value comparisons.

```json
{
    "Conditions": [
        {
            "ConditionType": "ImportantFormKeyConfidenceCheck",
            "ConditionParameters": {
```

1549
Example 3: Use Sampling

In the following example, 5% of inferences resulting from an Amazon Textract AnalyzeDocument request will be sent to human workers for review. All detected key-value pairs returned by Amazon Textract are sent to workers for review.

```json
{
  "Conditions": [
    {
      "ConditionType": "Sampling",
      "ConditionParameters": {
        "RandomSamplingPercentage": 5
      }
    }
  ]
}
```

Example 4: Use MissingImportantFormKey

In the following example, if Mailing Address or its alias, Mailing Address: is missing from keys detected by Amazon Textract, a human review will be triggered. When using the default worker task template, the worker UI asks workers to identify the key Mailing Address or Mailing Address: and its associated value.

```json
{
  "ConditionType": "MissingImportantFormKey",
  "ConditionParameters": {
    "ImportantFormKey": "Mailing Address",
    "ImportantFormKeyAliases": ["Mailing Address:"
  }
}
```

Example 5: Use Sampling and ImportantFormKeyConfidenceCheck with the And operator

In this example, 5% of key-value pairs detected by Amazon Textract whose key is one of Pay Date, PayDate, DateOfPay, or pay-date, with the confidence of the key-value block less than 65 and the confidences of each of the word blocks making up the key and value less than 85 are sent to workers for review.

```json
{
  "Conditions": [
    {"And": [
      {"ConditionType": "Sampling",
       "ConditionParameters": {
         "RandomSamplingPercentage": 5
       }
      },
      {"ConditionType": "ImportantFormKeyConfidenceCheck",
       "ConditionParameters": {
```
Example 6: Use Sampling and ImportantFormKeyConfidenceCheck with the And operator

Use this example to configure your human review workflow to always send low confidence inferences of a specified key-value pair for human review and sample high confidence inference of a key-value pair at a specified rate.

In the following example, a human review is triggered in one of the following ways:

- Key-value pairs detected whose key is one of Pay Date, PayDate, DateOfPay, or pay-date, with key-value and word block confidences less than 60 will be sent for human review. Only the Pay Date form key (and its aliases) and associated values are sent to workers to review.
- 5% of key-value pairs detected whose key is one of Pay Date, PayDate, DateOfPay, or pay-date, with key-value and word block confidences greater than 90 will be sent for human review. Only the Pay Date form key (and its aliases) and associated values are sent to workers to review.

```json
{
  "Conditions": [
    {
      "Or": [
        {
          "ConditionType": "ImportantFormKeyConfidenceCheck",
          "ConditionParameters": {
            "ImportantFormKey": "Pay Date",
            "ImportantFormKeyAliases": [
              "PayDate",
              "DateOfPay",
              "pay-date"
            ],
            "KeyValueBlockConfidenceLessThan": 60,
            "WordBlockConfidenceLessThan": 60
          }
        },
        {
          "And": [
            {
              "ConditionType": "Sampling",
              "ConditionParameters": {
                "RandomSamplingPercentage": 5
              }
            },
            {
              "ConditionType": "ImportantFormKeyConfidenceCheck",
              "ConditionParameters": {
                "ImportantFormKey": "Pay Date",
                "ImportantFormKeyAliases": [
                  "PayDate",
                  "DateOfPay"
                ],
                "KeyValueBlockConfidenceLessThan": 65,
                "WordBlockConfidenceLessThan": 85
              }
            }
          ]
        }
      ]
    }
  ]
}
```
Example 7: Use Sampling and ImportantFormKeyConfidenceCheck with the Or operator

In the following example, the Amazon Textract AnalyzeDocument operation returns a key-value pair whose key is one of Pay Date, PayDate, DateOfPay, or pay-date, with the confidence of the key-value block less than 65 and the confidences of each of the word blocks making up the key and value less than 85. Additionally, 5% of all other forms will trigger a human loop. For each form randomly chosen, all key-value pairs detected for that form will be sent to humans for review.

```json
{
  "Conditions": [
    {
      "Or": [
        {
          "ConditionType": "Sampling",
          "ConditionParameters": {
            "RandomSamplingPercentage": 5
          }
        },
        {
          "ConditionType": "ImportantFormKeyConfidenceCheck",
          "ConditionParameters": {
            "ImportantFormKey": "Pay Date",
            "ImportantFormKeyAliases": [
              "PayDate",
              "DateOfPay",
              "pay-date"
            ],
            "KeyValueBlockConfidenceLessThan": 65,
            "WordBlockConfidenceLessThan": 85
          }
        }
      ]
    }
  ]
}
```

Use Human Loop Activation Conditions JSON Schema with Amazon Rekognition

When used with Amazon A2I, the Amazon Rekognition DetectModerationLabels operation supports the following inputs in the ConditionType parameters:

- **ModerationLabelConfidenceCheck** – Use this condition type to create a human loop when inference confidence is low for one or more specified labels.
- **Sampling** – Use this condition to specify a percentage of all inferences to send to humans for review. Use this condition to do the following:
Audit your ML model by randomly sampling all of your model's inferences and sending a specified percentage to humans for review.

Using the ModerationLabelConfidenceCheck condition, randomly sample a percentage of the inferences that met the conditions specified in ModerationLabelConfidenceCheck to start a human loop and send only the specified percentage to humans for review.

**Note**
If you send the same request to DetectModerationLabels multiple times, the result of Sampling will not change for the inference of that input. For example, if you make a DetectModerationLabels request once, and Sampling does not trigger a HumanLoop, subsequent requests to DetectModerationLabels with the same configuration won't trigger a human loop.

When creating a flow definition, if you use the default worker task template that is provided in the Human review workflows section of the Amazon SageMaker console, inferences sent for human review by these activation conditions are included in the worker UI when a worker opens your task. If you use a custom worker task template, you need to include the `<task.input.selectedAiServiceResponse.blocks>` custom HTML element to access these inferences. For an example of a custom template that uses this HTML element, see Custom Template Example for Amazon Rekognition (p. 1569).

**ModerationLabelConfidenceCheck Inputs**

For the ModerationLabelConfidenceCheck ConditionType, the following ConditionParameters are supported:

- **ModerationLabelName** – The exact (case-sensitive) name of a ModerationLabel detected by the Amazon Rekognition DetectModerationLabels operation. You can specify the special catch-all value (*) to denote any moderation label.
- **ConfidenceEquals**
- **ConfidenceLessThan**
- **ConfidenceLessThanEquals**
- **ConfidenceGreaterThan**
- **ConfidenceGreaterThanEquals**

When you use the ModerationLabelConfidenceCheck ConditionType, Amazon A2I sends label inferences for the labels that you specified in ModerationLabelName for human review.

**Sampling Inputs**

The Sampling ConditionType supports the RandomSamplingPercentage ConditionParameters. The input for the RandomSamplingPercentage parameter should be real number between 0.01 and 100. This number represents the percentage of inferences that qualifies for a human review that are sent to humans for review. If you use the Sampling condition without any other conditions, this number represents the percentage of all inferences that result from a single DetectModerationLabel request that are sent to humans for review.

**Examples**

**Example 1: Use ModerationLabelConfidenceCheck with the And operator**

The following example of a HumanLoopActivationConditions condition triggers a HumanLoop when one or more of the following conditions are met:

- Amazon Rekognition detects the Graphic Male Nudity moderation label with a confidence between 90 and 99.
• Amazon Rekognition detects the Graphic Female Nudity moderation label with a confidence between 80 and 99.

Note the use of the Or and And logical operators to model this logic.

Although only one of the two conditions under the Or operator need to evaluate to true for a HumanLoop to be created, Amazon Augmented AI evaluates all conditions. Human reviewers are asked to review the moderation labels for all the conditions that evaluated to true.

```json
{
  "Conditions": [{
    "Or": [{
      "And": [{
        "ConditionType": "ModerationLabelConfidenceCheck",
        "ConditionParameters": {
          "ModerationLabelName": "Graphic Male Nudity",
          "ConfidenceLessThanEquals": 99
        }
      }, {
        "ConditionType": "ModerationLabelConfidenceCheck",
        "ConditionParameters": {
          "ModerationLabelName": "Graphic Male Nudity",
          "ConfidenceGreaterThanEquals": 90
        }
      }]
    }, {
      "And": [{
        "ConditionType": "ModerationLabelConfidenceCheck",
        "ConditionParameters": {
          "ModerationLabelName": "Graphic Female Nudity",
          "ConfidenceLessThanEquals": 99
        }
      }, {
        "ConditionType": "ModerationLabelConfidenceCheck",
        "ConditionParameters": {
          "ModerationLabelName": "Graphic Female Nudity",
          "ConfidenceGreaterThanEquals": 80
        }
      }]
    }]
  }
}
```

**Example 2: Use ModerationLabelConfidenceCheck with the catch-all value (*)**

In the following example, if any moderation label with a confidence greater than or equal to 75 is detected, a HumanLoop is triggered. Human reviewers are asked to review all moderation labels with confidence scores greater than or equal to 75.

```json
{
  "Conditions": [
    {
      "ConditionType": "ModerationLabelConfidenceCheck",
      "ConditionParameters": {
        "ModerationLabelName": "*",
        "ConfidenceGreaterThanEquals": 75
      }
    }
  ]
}
```
Example 3: Use Sampling

In the following example, 5% of Amazon Rekognition inferences from a DetectModerationLabels request will be sent to human workers. When using the default worker task template provided in the SageMaker console, all moderation labels returned by Amazon Rekognition are sent to workers for review.

```json
{
  "Conditions": [
    {
      "ConditionType": "Sampling",
      "ConditionParameters": {
        "RandomSamplingPercentage": 5
      }
    }
  ]
}
```

Example 4: Use Sampling and ModerationLabelConfidenceCheck with the And operator

In this example, 5% of Amazon Rekognition inferences of the Graphic Male Nudity moderation label with a confidence greater than 50 will be sent workers for review. When using the default worker task template provided in the SageMaker console, only the Graphic Male Nudity label will be sent to workers for review.

```json
{
  "Conditions": [
    {
      "And": [
        {
          "ConditionType": "Sampling",
          "ConditionParameters": {
            "RandomSamplingPercentage": 5
          }
        },
        {
          "ConditionType": "ModerationLabelConfidenceCheck",
          "ConditionParameters": {
            "ModerationLabelName": "Graphic Male Nudity",
            "ConfidenceGreaterThan": 50
          }
        }
      ]
    }
  ]
}
```

Example 5: Use Sampling and ModerationLabelConfidenceCheck with the And operator

Use this example to configure your human review workflow to always send low confidence inferences of a specified label for human review and sample high confidence inference of a label at a specified rate.

In the following example, a human review is triggered in one of the following ways:

- Inferences for the Graphic Male Nudity moderation label the with confidence scores less than 60 are always sent for human review. Only the Graphic Male Nudity label is sent to workers to review.
• 5% of all inferences for the Graphic Male Nudity moderation label with confidence scores greater than 90 will be sent for human review. Only the Graphic Male Nudity label is sent to workers to review.

```json
{
    "Conditions": [
        {
            "Or": [
                {
                    "ConditionType": "ModerationLabelConfidenceCheck",
                    "ConditionParameters": {
                        "ModerationLabelName": "Graphic Male Nudity",
                        "ConfidenceLessThan": 60
                    }
                },
                {
                    "And": [
                        {
                            "ConditionType": "Sampling",
                            "ConditionParameters": {
                                "RandomSamplingPercentage": 5
                            }
                        },
                        {
                            "ConditionType": "ModerationLabelConfidenceCheck",
                            "ConditionParameters": {
                                "ModerationLabelName": "Graphic Male Nudity",
                                "ConfidenceGreaterThan": 90
                            }
                        }
                    ]
                }
            ]
        }
    ]
}
```

**Example 6: Use Sampling and ModerationLabelConfidenceCheck with the Or operator**

In the following example, a human loop is created if the Amazon Rekognition inference response contains the 'Graphic Male Nudity' label with inference confidence greater than 50. Additionally, 5% of all other inferences will trigger a human loop.

```json
{
    "Conditions": [
        {
            "Or": [
                {
                    "ConditionType": "Sampling",
                    "ConditionParameters": {
                        "RandomSamplingPercentage": 5
                    }
                },
                {
                    "ConditionType": "ModerationLabelConfidenceCheck",
                    "ConditionParameters": {
                        "ModerationLabelName": "Graphic Male Nudity",
                        "ConfidenceGreaterThan": 50
                    }
                }
            ]
        }
    ]
}
```
Delete a Human Review Workflow

When you delete a human review workflow or you delete your AWS account while a human loop is in process, your human review workflow status changes to Deleting. All associated human loops will automatically be stopped and deleted by Amazon A2I if workers have not started tasks created by those human loops. If human workers are already working on a task, that task will continue to be available until it is completed or expires. As long as workers are still working on a task, your human review workflow's status will be Deleting. If these tasks are completed, the results are stored in the Amazon S3 bucket specified in your flow definition.

Deleting a flow definition won't remove any worker answers from your Amazon S3 bucket. If the tasks are completed, but you deleted your AWS account, the results will be stored in the Augmented AI service bucket for thirty (30) days and then permanently deleted.

After all human loops have been deleted, the human review workflow will be permanently deleted. When a human review workflow has been deleted, you can reuse its name to create a new human review workflow.

You might want to delete a human review workflow for any of the following reasons:

- You have sent data to a set of human reviewers and you want to delete all non-started human loops because you do not want those workers to work on those tasks any longer.
- The worker task template used to generate your worker UI does not render correctly or is not functioning as expected.

After you delete a human review workflow, the following changes occur:

- The human review workflow no longer appears on the Human review workflows page in the Augmented AI area of the Amazon SageMaker console.
- When you use the human review workflow name as input to the API operations `DescribeFlowDefinition` or `DeleteFlowDefinition`, Augmented AI returns a ResourceNotFound error.
- When you use `ListFlowDefinitions`, deleted human review workflows won't be included in the results.
- When you use `ListHumanLoops`, Augmented AI returns a ResourceNotFoundException.

Delete a Flow Definition Using the Console or the SageMaker API

You can delete a human review workflow on the Human review workflows page in the Augmented AI area of the SageMaker console or by using the SageMaker API.

Flow definitions can only be deleted if their status is Active.

Delete a human review workflow (console)

1. Navigate to the Augmented AI console at https://console.aws.amazon.com/a2i/.
2. In the navigation pane, under the Augmented AI section, choose Human review workflows.
3. Choose the hyperlinked name of the human review workflow that you want to delete.
4. On the Summary page of your human review workflow, choose Delete in the upper-right corner.
5. In the dialog box asking you to confirm that you want to delete your human review workflow, choose Delete.

You're automatically redirected to the Human review workflows page. While your human review workflow is being deleted, the status Deleting appears in the status column for that workflow. After it's deleted, it won't appear in the list of workflows on this page.

Delete a human review workflow (API)

You can delete a human review workflow (flow definition) using the SageMaker DeleteFlowDefinition API operation. This API operation is supported through the AWS CLI and a variety of language specific SDKs. The following table shows example requests using SDK for Python (Boto3) and the AWS CLI to delete the human review workflow, example-flow-definition.

AWS SDK for Python (Boto3)

The following request example uses the SDK for Python (Boto3) to delete the human review workflow. For more information, see delete_flow_definition in the AWS SDK for Python (Boto) API Reference.

```python
import boto3
sagemaker_client = boto3.client('sagemaker')
response = sagemaker_client.delete_flow_definition(FlowDefinitionName='example-flow-definition')
```

AWS CLI

The following request example uses the AWS CLI to delete the human review workflow. For more information, see delete-flow-definition in the AWS CLI Command Reference.

```
$ aws sagemaker delete-flow-definition --flow-definition-name 'example-flow-definition'
```

If the action is successful, Augmented AI sends back an HTTP 200 response with an empty HTTP body.

Create and Start a Human Loop

A **human loop** starts your human review workflow and sends data review tasks to human workers. When you use one of the Amazon A2I built-in task types, the corresponding AWS service creates and starts a human loop on your behalf when the conditions specified in your flow definition are met. If no conditions were specified in your flow definition, a human loop is created for each object. When using Amazon A2I for a custom task, a human loop starts when StartHumanLoop is called in your application.

Use the following instructions to configure a human loop with Amazon Rekognition or Amazon Textract built-in task types and custom task types.

**Prerequisites**

To create and start a human loop, the AmazonAugmentedAIFullAccess policy must be attached to the AWS Identity and Access Management (IAM) user or role that configures or starts the human loop. This
will be the identity that you use to configure the human loop using `HumanLoopConfig` for built-in task types. For custom task types, this will be the identity that you use to call `StartHumanLoop`.

Additionally, when using a built-in task type, your IAM user or role must have permission to invoke API operations of the AWS service associated with your task type. For example, if using Amazon Rekognition with Augmented AI, you must attach permissions required to call `DetectModerationLabels`. For examples of identity-based policies you can use to grant these permissions, see Amazon Rekognition Identity-Based Policy Examples and Amazon Textract Identity-Based Policy Examples. You can also use the more general policy `AmazonAugmentedAllIntegratedAPIAccess` to grant these permissions. For more information, see Create an IAM User With Permissions to Invoke Amazon A2I, Amazon Textract, and Amazon Rekognition API Operations (p. 1587).

To create and start a human loop, you will need a flow definition ARN. To learn how to create a flow definition (or human review workflow), see Create a Human Review Workflow (p. 1540).

**Important**
Amazon A2I requires all S3 buckets that contain human loop input image data have a CORS policy attached. To learn more about this change, see CORS Permission Requirement (p. 1585).

## Create and Start a Human Loop for a Built-in Task Type

To start a human loop for a built-in task type jobs use the corresponding service's API to provide your input data and to configure the human loop. For Amazon Textract, you use the `AnalyzeDocument` API operation. For Amazon Rekognition, you use the `DetectModerationLabels` API operation. You can use the AWS CLI, or a language-specific SDK to create requests using these API operations.

**Important**
When you create a human loop using a built-in task type, you can use `DataAttributes` to specify a set of `ContentClassifiers` related to the input provided to the `StartHumanLoop` operation. Use content classifiers to declare that your content is free of personally identifiable information or adult content.

To use Amazon Mechanical Turk, ensure your data is free of personally identifiable information, including protected health information under HIPAA, and include the `FreeOfPersonallyIdentifiableInformation` content classifier. If you do not use this content classifier, SageMaker will not send your task to Mechanical Turk. If your data is free of adult content, also include the ‘FreeOfAdultContent’ classifier. If you do not use these content classifiers, SageMaker may restrict the Mechanical Turk workers that can view your task.

After you start your ML job using your built-in task type's AWS service API, Amazon A2I monitors the inference-results of that service. For example, when running a job with Amazon Rekognition, Amazon A2I checks the inference confidence score for each image and compares it to the confidence-thresholds specified in your flow definition. If the conditions to start a human review task are satisfied, or if you didn't specify conditions in your flow definition, a human review task is sent to workers.

## Create an Amazon Textract Human Loop

Amazon A2I integrates with Amazon Textract so that you can configure and start a human loop using the Amazon Textract API. To send a document file to Amazon Textract for text analysis, you use the Amazon Textract `AnalyzeDocument` API operation. To configure a human loop, set the `HumanLoopConfig` parameter when you configure `AnalyzeDocument`. To learn how, see Step 3 in Running `AnalyzeDocument` with Amazon A2I in the Amazon Textract Developer Guide.

After you run the `AnalyzeDocument` with a human loop configured, Amazon A2I monitors the results from `AnalyzeDocument` and checks it against the flow definition's activation conditions. If Amazon Textract inference confidence score for one or more key value pairs meets the conditions for review,
Amazon A2I starts a human review loop and includes the `HumanLoopActivationOutput` object in the `AnalyzeDocument` response.

### Create an Amazon Rekognition Human Loop

Amazon A2I integrates with Amazon Rekognition so that you can configure and start a human loop using the Amazon Rekognition API. To send images to Amazon Rekognition for content moderation, you use the Amazon Rekognition `DetectModerationLabels` API operation. To configure a human loop, set the `HumanLoopConfig` parameter when you configure `DetectModerationLabels`. To learn how, see Step 3 in Running `DetectModerationLabels` with Amazon A2I in the Amazon Rekognition Developer Guide.

After you run the `DetectModerationLabels` with a human loop configured, Amazon A2I monitors the results from `DetectModerationLabels` and checks it against the flow definition’s activation conditions. If the Amazon Rekognition inference confidence score for an image meets the conditions for review, Amazon A2I starts a human review loop and includes the `HumanLoopActivationOutput` response object in the `DetectModerationLabels` response.

### Create and Start a Human Loop for a Custom Task Type

To configure a human loop for a custom human review task, use the `StartHumanLoop` operation within your application. This section provides an example of a human loop request using the AWS SDK for Python (Boto3) and the AWS Command Line Interface (AWS CLI). For documentation on other language specific SDK’s that support `StartHumanLoop`, use the See Also section of `StartHumanLoop` in the Amazon Augmented AI Runtime API documentation.

#### Prerequisites

To complete this procedure, you need:

- Input data formatted as a string representation of a JSON-formatted file.
- The Amazon Resource Name (ARN) of your flow definition
- A flow definition ARN.

#### To configure the human loop

1. For `DataAttributes`, specify a set of `ContentClassifiers` related to the input provided to the `StartHumanLoop` operation. Use content classifiers to declare that your content is free of personally identifiable information or adult content.

   To use Amazon Mechanical Turk, ensure your data is free of personally identifiable information, including protected health information under HIPAA, and include the `FreeOfPersonallyIdentifiableInformation` content classifier. If you do not use this content classifier, SageMaker will not send your task to Mechanical Turk. If your data is free of adult content, also include the 'FreeOfAdultContent' classifier. If you do not use these content classifiers, SageMaker may restrict the Mechanical Turk workers that can view your task.

2. For `FlowDefinitionArn`, enter the Amazon Resource Name (ARN) of your flow definition.

3. For `HumanLoopInput`, enter your input data as a string representation of a JSON-formatted file. Structure your input data and custom worker task template so that your input data is properly displayed to human workers when you start your human loop. See to learn how to preview your custom worker task template.

4. For `HumanLoopName`, enter a name for the human loop. The name must be unique within the Region in your account, and can have up to 63 characters. Valid characters: a-z, 0-9, and - (hyphen).
To start a human loop

- To start a human loop, submit a request similar to the following examples using your preferred language specific SDK.

AWS SDK for Python (Boto3)

The following request example uses the SDK for Python (Boto3). For more information, see Boto 3 Augmented AI Runtime in the AWS SDK for Python (Boto) API Reference.

```python
response = client.start_human_loop(
    HumanLoopName='string',
    FlowDefinitionArn='string',
    HumanLoopInput={
        'InputContent': 'string'
    },
    DataAttributes={
        'ContentClassifiers': [
            'FreeOfPersonallyIdentifiableInformation',
            'FreeOfAdultContent',
        ]
    }
)
```

AWS CLI

The following request example uses the AWS CLI. For more information, see start-human-loop in the AWS CLI Command Reference.

```
# InputContent=string
ContentClassifiers="FreeOfPersonallyIdentifiableInformation","FreeOfAdultContent"
start-human-loop --human-loop-name example-humanloop
    --human-loop-input InputContent
    --data-attributes ContentClassifiers
```

When you successfully start a human loop by invoking StartHumanLoop directly, the response will include a HumanLoopARN and a HumanLoopActivationResults object which will be set to NULL. You can use this the human loop name to monitor and manage your human loop.

**Next Steps:**

After starting a human loop, you can manage and monitor it with the Amazon Augmented AI Runtime API and Amazon CloudWatch Events. To learn more, see Monitor and Manage Your Human Loop (p. 1574).

**Delete a Human Loop**

When you delete a human loop, the status changes to Deleting. When the human loop is deleted, the associated human review task is no longer available to workers. You might want to delete a human loop in one of the following circumstances:

- The worker task template used to generate your worker UI does not render correctly or is not functioning as expected.
- A single data object was accidentally sent to workers multiple times.
- You no longer need a data object reviewed by a human.
If the status of a human loop is InProgress, you must stop the human loop before deleting it. When you stop a human loop, the status changes to Stopping while it is being stopped. When the status changes to Stopped, you can delete the human loop.

If human workers are already working on a task when you stop the associated human loop, that task continues to be available until it is completed or expires. As long as workers are still working on a task, your human loop's status is Stopping. If these tasks are completed, the results are stored in the Amazon S3 bucket URI specified in your human review workflow. If the worker leaves the task without submitting work, it is stopped and the worker can't return to the task. If no worker has started working on the task, it is stopped immediately.

If you delete the AWS account used to create the human loop, it is stopped and deleted automatically.

Human Loop Data Retention and Deletion

When a human worker completes a human review task, the results are stored in the Amazon S3 output bucket you specified in the human review workflow used to create the human loop. Deleting or stopping a human loop does not remove any worker answers from your S3 bucket.

Additionally, Amazon A2I temporarily stores human loop input and output data internally for the following reasons:

- If you configure your human loops so that a single data object is sent to multiple workers for review, Amazon A2I does not write output data to your S3 bucket until all workers have completed the review task. Amazon A2I stores partial answers—answers from individual workers—internally so that it can write full results to your S3 bucket.
- If you report a low quality human review result, Amazon A2I can investigate and respond to your issue.
- If you lose access to or delete the output S3 bucket specified in the human review workflow used to create a human loop, and the task has already been sent to one or more workers, Amazon A2I needs a place to temporarily store human review results.

Amazon A2I deletes this data internally 30 days after a human loop's status changes to one of the following: Deleted, Stopped, or Completed. In other words, data is deleted 30 days after the human loop has been completed, stopped, or deleted. Additionally, this data is deleted after 30 days if you close the AWS account used to create associated human loops.

Stop and Delete a Flow Definition Using the Console or the Amazon A2I API

You can stop and delete a human loop in the Augmented AI console or by using the SageMaker API. When the human loop has been deleted, the status changes to Deleted.

Delete a human loop (console)

1. Navigate to the Augmented AI console at https://console.aws.amazon.com/a2i/.
2. In the navigation pane, under the Augmented AI section, choose Human review workflows.
3. Choose the hyperlinked name of the human review workflow you used to create the human loop you want to delete.
4. In the Human loops section at the bottom of the page, select the human loop you want to stop and delete.
5. If the human loop status is Completed, Stopped, or Failed, select Delete.

   If the human loop Status is InProgress, select Stop. When the status changes to Stopped, select Delete.
Delete a human loop (API)

1. Check the status of your human loop using the Augmented AI Runtime API operation `DescribeHumanLoop`. See examples using this operation in the following table.

**AWS SDK for Python (Boto3)**

The following example uses the SDK for Python (Boto3) to describe the human loop named `example-human-loop`. For more information, see `describe_human_loop` in the [AWS SDK for Python (Boto) API Reference](https://boto3.amazonaws.com/v1/documentation/api/latest/reference/services/sagemaker-a2i-runtime.html#SageMakerA2IRuntime.Client.describe_human_loop).

```python
import boto3

a2i_runtime_client = boto3.client('sagemaker-a2i-runtime')
response = a2i_runtime_client.describe_human_loop(HumanLoopName='example-human-loop')
human_loop_status = response['HumanLoopStatus']
print(f'example-human-loop status is: {human_loop_status}')
```

**AWS CLI**

The following example uses the AWS CLI to describe the human loop named `example-human-loop`. For more information, see `describe-human-loop` in the [AWS CLI Command Reference](https://docs.aws.amazon.com/cli/latest/reference/sagemaker-a2i-runtime/describe-human-loop.html).

```
$ aws sagemaker-a2i-runtime describe-human-loop --human-loop-name 'example-human-loop'
```

2. If the flow definition status is Completed, Stopped, or Failed, delete the flow definition using the Augmented AI Runtime API operation `DeleteHumanLoop`.

**AWS SDK for Python (Boto3)**

The following example uses the SDK for Python (Boto3) to delete the human loop named `example-human-loop`. For more information, see `delete_human_loop` in the [AWS SDK for Python (Boto) API Reference](https://boto3.amazonaws.com/v1/documentation/api/latest/reference/services/sagemaker-a2i-runtime.html#SageMakerA2IRuntime.Client.delete_human_loop).

```python
import boto3

a2i_runtime_client = boto3.client('sagemaker-a2i-runtime')
response = a2i_runtime_client.delete_human_loop(HumanLoopName='example-human-loop')
```

**AWS CLI**

The following example uses the AWS CLI to delete the human loop named `example-human-loop`. For more information, see `delete-human-loop` in the [AWS CLI Command Reference](https://docs.aws.amazon.com/cli/latest/reference/sagemaker-a2i-runtime/delete-human-loop.html).

```
$ aws sagemaker-a2i-runtime delete-human-loop --human-loop-name 'example-human-loop'
```

If the human loop status is InProgress, stop the human loop using `StopHumanLoop` and then use `DeleteHumanLoop` to delete it.

**AWS SDK for Python (Boto3)**

The following example uses the SDK for Python (Boto3) to describe the human loop named `example-human-loop`. For more information, see `stop_human_loop` in the [AWS SDK for Python (Boto) API Reference](https://boto3.amazonaws.com/v1/documentation/api/latest/reference/services/sagemaker-a2i-runtime.html#SageMakerA2IRuntime.Client.stop_human_loop).

```python
import boto3

a2i_runtime_client = boto3.client('sagemaker-a2i-runtime')
response = a2i_runtime_client.stop_human_loop(HumanLoopName='example-human-loop')
```

**AWS CLI**

The following example uses the AWS CLI to stop the human loop named `example-human-loop`. For more information, see `stop-human-loop` in the [AWS CLI Command Reference](https://docs.aws.amazon.com/cli/latest/reference/sagemaker-a2i-runtime/stop-human-loop.html).

```
$ aws sagemaker-a2i-runtime stop-human-loop --human-loop-name 'example-human-loop'
```
Create and Manage Worker Task Templates

You can create a task UI for your workers by creating a worker task template. A worker task template is an HTML file that is used to display your input data and instructions to help workers complete your task.

For Amazon Rekognition or Amazon Textract task types, you can customize a pre-made worker task template using a graphical user interface (GUI) and avoid interacting with HTML code. For this option, use the instructions in Create a Human Review Workflow (Console) (p. 1541) to create a human review workflow and customize your worker task template in the Amazon SageMaker console. Once you create a template using these instructions, it will appear on the Worker task templates page of the Augmented AI console.

If you are creating a human review workflow for a custom task type, you must create a custom worker task template using HTML code. For more information, see Create Custom Worker Task Template (p. 1566).

If you create your template using HTML, you must use this template to generate an Amazon A2I human task UI Amazon Resource Name (ARN) in the Amazon A2I console. This ARN has the following format: arn:aws:sagemaker:<aws-region>:<aws-account-number>:human-task-ui/<template-name>. This ARN is associated with a worker task template resource that you can use in one or more human review workflows (flow definitions).

Generate a human task UI ARN using a worker task template by following the instructions found in Create a Worker Task Template (p. 1565) or by using the CreateHumanTaskUi API operation.

Topics
- Create and Delete a Worker Task Templates (p. 1564)
- Create Custom Worker Task Template (p. 1566)
- Creating Good Worker Instructions (p. 1573)

Create and Delete a Worker Task Templates

You can use a worker template to customize the interface and instructions that your workers see when working on your tasks. Use the instructions on this page to create a worker task template in the Augmented AI area of the Amazon SageMaker console. A starter template is provided for Amazon Textract and Amazon Rekognition tasks. To learn how to customize your template using HTML crowd elements, see Create Custom Worker Task Template (p. 1566).

When you create a worker template in the Worker task templates page of the Augmented AI area of the SageMaker console, a worker task template ARN will be generated. Use this ARN as the input to
Create and Delete Worker Task Templates

HumanTaskUiArn when you create a flow definition using the API operation CreateFlowDefinition. You can choose this template when creating a human review workflow on the Human review workflows page of the console.

If you are creating a worker task template resource for an Amazon Textract or Amazon Rekognition task type, you can preview the worker UI that will be generated from your template on the Worker task templates console page. You will need to attach the policy described in Enable Worker Task Template Previews (p. 1588) to the IAM role that you use to preview the template.

Create a Worker Task Template

You can create a worker task template using the SageMaker console and using the SageMaker API operation CreateHumanTaskUi.

Create a worker task template (console)

1. Open the Amazon A2I console at https://console.aws.amazon.com/a2i.
2. Under Amazon Augmented AI in the left navigation pane, choose Worker task templates.
3. Choose Create template.
4. In Template name, enter a unique name.
5. (Optional) Enter an IAM role that grants A2I the permissions necessary to call services on your behalf.
6. In Template type, choose a template type from the drop-down menu. If you are creating a template for a Textract-form extraction or Rekognition-image moderation task, choose the appropriate option.
7. Enter your custom template elements as follows:
   - If you selected the Amazon Textract or Amazon Rekognition task template, the Template editor autopopulates with a default template that you can customize.
   - If you are using a custom template, enter your predefined template in the editor.
8. (Optional) To complete this step, you must provided an IAM role ARN with permission to read Amazon S3 objects that get rendered on your user interface in Step 5.
   
   You can only preview your template if you are creating templates for Amazon Textract or Amazon Rekognition.
   
   Choose See preview to preview the interface and instructions that workers will see. This is an interactive preview. After you complete the sample task and choose Submit, you see the resulting output from the task that you just performed.
   
   If you are creating a worker task template for a custom task type, you can preview your worker task UI using RenderUiTemplate. For more information, see Preview a Worker Task Template (p. 1572).
9. When you’re satisfied with your template, choose Create.

After you’ve created your template, you can select that template when you create a human review workflow in the console. Your template also appears in the Amazon Augmented AI section of the SageMaker console under Worker task templates. Choose your template to view its ARN. Use this ARN when using the API operation CreateFlowDefinition.

Create a worker task template using a worker task template (API)

To generate a worker task template using the SageMaker API operation CreateHumanTaskUi, specify a name for your UI in HumanTaskUiName and input your HTML template in Content under UiTemplate. Find documentation on language-specific SDKs that support this API operation in the See Also section of the CreateHumanTaskUi.
Delete a Worker Task Template

Once you have created a worker task template, it can be deleted using the SageMaker console or using the SageMaker API operation `DeleteHumanTaskUi`.

When you delete a worker task template, you will not be able to use human review workflows (flow definitions) that were created using that template to start human loops. Any human loops that have already been created using the worker task template that you delete will continue to be processed until completion and will not be impacted.

Delete a worker task template (console)

1. Open the Amazon A2I console at https://console.aws.amazon.com/a2i.
2. Under Amazon Augmented AI in the left navigation pane, choose Worker task templates.
3. Select the template that you want to delete.
4. Select Delete.
5. A modal will appear to confirm your choice. Select Delete.

Delete a worker task template (API)

To delete a worker task template using the SageMaker API operation `DeleteHumanTaskUi`, specify a name of your UI in `HumanTaskUiName`.

Create Custom Worker Task Template

_Crowd HTML Elements_ are web components that provide a number of task widgets and design elements that you can tailor to the question you want to ask. You can use these crowd elements to create a custom worker template and integrate it with an Amazon Augmented AI (Amazon A2I) human review workflow to customize the worker console and instructions.

For a list of all HTML crowd elements available to Amazon A2I users, see [Crowd HTML Elements Reference](p. 448). To see examples of templates, see the AWS Github repository, which contains over 60 sample custom task templates.

Develop Templates Locally

When in the console to test how your template process incoming data, you can test the look and feel of your template's HTML and custom elements in your browser by adding the following code to the top of your HTML file.

```
<script src="https://assets.crowd.aws/crowd-html-elements.js"></script>
```

This loads the necessary code to render the custom HTML elements. Use this code if you want to develop your template's look and feel in your preferred editor instead of in the console.

This code won't parse your variables. You might want to replace them with sample content while developing locally.

Use External Assets

Amazon Augmented AI custom templates enable you to embed external scripts and style sheets. For example, the following header embeds a `text/css` style sheet name `stylesheet` located at https://www.example.com/my-enhancement-styles.css into the custom template.
Example

```html
<script src="https://www.example.com/my-enhancement-script.js"></script>
<link rel="stylesheet" type="text/css" href="https://www.example.com/my-enhancement-styles.css" />
```

If you encounter errors, ensure that your originating server is sending the correct MIME type and encoding headers with the assets.

For example, the MIME and encoding types for remote scripts is `application/javascript; CHARSET=UTF-8`.

The MIME and encoding type for remote stylesheets is `text/css; CHARSET=UTF-8`.

Track Your Variables

When building a custom template, you must add variables to it to represent the pieces of data that might change from task to task, or worker to worker. If you're starting with one of the sample templates, you need to make sure you're aware of the variables it already uses.

For example, for a custom template that integrates an Augmented AI human review loop with a Amazon Textract text review task, `{{ task.input.selectedAiServiceResponse.blocks }}` is used for initial-value input data. For Amazon Augmented AI (Amazon A2I) integration with Amazon Rekognition, `{{ task.input.selectedAiServiceResponse.moderationLabels }}` is used. For a custom task type, you need to determine the input parameter for your task type. Use `{{ task.input.customInputValuesForStartHumanLoop }}` where you specify `customInputValuesForStartHumanLoop`.

Custom Template Example for Amazon Textract

All custom templates begin and end with the `<crowd-form> </crowd-form>` elements. Like standard HTML `<form>` elements, all of your form code should go between these elements.

For a Amazon Textract document analysis task, use the `<crowd-textract-document-analysis>` element. It uses the following attributes:

- `src` – Specifies the URL of the image file to be annotated.
- `initialValue` – Sets initial values for attributes found in the worker UI.
- `blockTypes` (required) – Determines the kind of analysis that the workers can do. Only `KEY_VALUE_SET` is currently supported.
- `keys` (required) – Specifies new keys and the associated text value that the worker can add.
- `no-key-edit` (required) – Prevents the workers from editing the keys of annotations passed through `initialValue`.
- `no-geometry-edit` – Prevents workers from editing the polygons of annotations passed through `initialValue`.

For children of the `<crowd-textract-document-analysis>` element, you must have two regions. You can use arbitrary HTML and CSS elements in these regions.

- `<full-instructions>` – Instructions that are available from the View full instructions link in the tool. You can leave this blank, but we recommend that you provide complete instructions to get better results.
- `<short-instructions>` – A brief description of the task that appears in the tool's sidebar. You can leave this blank, but we recommend that you provide complete instructions to get better results.
An Amazon Textract template would look similar to the following.

Example

```html
<script src="https://assets.crowd.aws/crowd-html-elements.js"></script>

{% capture s3_arn %}http://s3.amazonaws.com/
{{ task.input.aiServiceRequest.document.s3Object.bucket }}/
{{ task.input.aiServiceRequest.document.s3Object.name }}{% endcapture %}

<crowd-form>
<crowd-textract-analyze-document
  src="{% s3_arn | grant_read_access %}
  initial-value="{{ task.input.selectedAiServiceResponse.blocks }}"
  header="Review the key-value pairs listed on the right and correct them if they don't match the following document."
  no-key-edit
  no-geometry-edit
  keys="{{ task.input.humanLoopContext.importantFormKeys }}"
  block-types="['KEY_VALUE_SET']">
  <short-instructions header="Instructions">
  <style>
    .instructions {
      white-space: pre-wrap;
    }
    .instructionsImage {
      display: inline-block;
      max-width: 100%;
    }
  </style>
  <p class='instructions'>Choose a key-value block to highlight the corresponding key-value pair in the document.

  If it is a valid key-value pair, review the content for the value. If the content is incorrect, correct it.

  The text of the value is incorrect, correct it.
  <img class='instructionsImage' src="https://example-site/correct-value-text.png" />

  A wrong value is identified, correct it.
  <img class='instructionsImage' src="https://example-site/correct-value.png" />

  If it is not a valid key-value relationship, choose No.
  <img class='instructionsImage' src="https://example-site/not-a-key-value-pair.png" />

  If you can’t find the key in the document, choose Key not found.
  <img class='instructionsImage' src="https://example-site/key-is-not-found.png" />

  If the content of a field is empty, choose Value is blank.
  <img class='instructionsImage' src="https://example-site/value-is-blank.png" />

  <b>Examples</b>
  Key and value are often displayed next to or below each other.
  <img class='instructionsImage' src="https://example-site/sample-key-value-pair-1.png" />

  Key and value displayed in one line.
  <img class='instructionsImage' src="https://example-site/sample-key-value-pair-2.png" />

  Key and value displayed in two lines.
  <img class='instructionsImage' src="https://example-site/sample-key-value-pair-2.png" />

  If the content of the value has multiple lines, enter all the text without a line break. Include all value text even if it extends beyond the highlight box.
  <img class='instructionsImage' src="https://assets.crowd.aws/images/a2i-console/multiple-lines.png" />
</p>
</short-instructions>
</crowd-textract-analyze-document>
</crowd-form>
```
Custom Template Example for Amazon Rekognition

All custom templates begin and end with the `<crowd-form>` elements. Like standard HTML `<form>` elements, all of your form code should go between these elements. For an Amazon Rekognition custom task template, use the `<crowd-rekognition-detect-moderation-labels>` element. This element supports the following attributes:

- **categories** – An array of strings or an array of objects where each object has a name field.
  - If the categories come in as objects, the following applies:
    - The displayed categories are the value of the name field.
    - The returned answer contains the full objects of any selected categories.
  - If the categories come in as strings, the following applies:
    - The returned answer is an array of all the strings that were selected.
- **exclusion-category** – By setting this attribute, you create a button underneath the categories in the UI. When a user presses the button, all categories are deselected and disabled. If the worker presses the button again, you re-enable users to choose categories. If the worker submits the task by selecting the Submit button after you pressing the button, that task will return an empty array.

For children of the `<crowd-textract-document-analysis>` element, you must have three regions.

- **<full-instructions>** – Instructions that are available from the View full instructions link in the tool. You can leave this blank, but we recommend that you provide complete instructions to get better results.
- **<short-instructions>** – Brief description of the task that appears in the tool's sidebar. You can leave this blank, but we recommend that you provide complete instructions to get better results.

A template using these elements would look similar to the following.

```html
<full-instructions header="Instructions"></full-instructions>
</crowd-textract-analyze-document>
</crowd-form>

<script src="https://assets.crowd.aws/crowd-html-elements.js"></script>
{% capture s3_arn %}http://s3.amazonaws.com/{{ task.input.aiServiceRequest.image.s3Object.bucket }}/{{ task.input.aiServiceRequest.image.s3Object.name }}{% endcapture %}

<crowd-form>
  <crowd-rekognition-detect-moderation-labels
categories='[
    {% for label in task.input.selectedAiServiceResponse.moderationLabels %}
      {
        name: "{{ label.name }}",
        parentName: "{{ label.parentName }}",
      },
    {% endfor %}
]
src="{{ s3_arn | grant_read_access }}"
header="Review the image and choose all applicable categories."
>
  <short-instructions header="Instructions">
    <style>
      .instructions {
        white-space: pre-wrap;
      }
    </style>
  </short-instructions>
</crowd-form>
```
Review the image and choose all applicable categories. If no categories apply, choose None.

- Nudity
  Visuals depicting nude male or female person or persons
- Graphic Male Nudity
  Visuals depicting full frontal male nudity, often close ups
- Graphic Female Nudity
  Visuals depicting full frontal female nudity, often close ups
- Sexual Activity
  Visuals depicting various types of explicit sexual activities and pornography
- Illustrated Nudity or Sexual Activity
  Visuals depicting animated or drawn sexual activity, nudity, or pornography
- Adult Toys
  Visuals depicting adult toys, often in a marketing context
- Female Swimwear or Underwear
  Visuals depicting female person wearing only swimwear or underwear
- Male Swimwear Or Underwear
  Visuals depicting male person wearing only swimwear or underwear
- Partial Nudity
  Visuals depicting covered up nudity, for example using hands or pose
- Revealing Clothes
  Visuals depicting revealing clothes and poses, such as deep cut dresses
- Graphic Violence or Gore
  Visuals depicting prominent blood or bloody injuries
- Physical Violence
  Visuals depicting violent physical assault, such as kicking or punching
- Weapon Violence
  Visuals depicting violence using weapons like firearms or blades, such as shooting
- Weapons
  Visuals depicting weapons like firearms and blades
- Self Injury
  Visuals depicting self-inflicted cutting on the body, typically in distinctive patterns using sharp objects
- Emaciated Bodies
  Visuals depicting extremely malnourished human bodies
- Corpses
  Visuals depicting human dead bodies
- Hanging
  Visuals depicting death by hanging
Add Automation with Liquid

The custom template system uses Liquid for automation. Liquid is an open-source inline markup language. For more information and documentation, see the Liquid homepage.

In Liquid, the text between single curly braces and percent symbols is an instruction or tag that creates control flow. Text between double curly braces is a variable or object that outputs its value.

Use Variable Filters

In addition to the standard Liquid filters and actions, Amazon Augmented AI (Amazon A2I) offers a few additional filters. You apply filters by placing a pipe (\|) character after the variable name, and then specifying a filter name. To chain filters use the following format.

Example

```liquid
{{ <content> | <filter> | <filter> }}
```

Autoescape and Explicit Escape

By default, inputs are HTML-escaped to prevent confusion between your variable text and HTML. You can explicitly add the escape filter to make it more obvious to someone reading the source of your template that escaping is being done.

escape_once

escape_once ensures that if you've already escaped your code, it doesn't get re-escaped again. For example, to ensure that `&amp;` doesn't become `&amp;amp;`.

skip_autoescape

skip_autoescape is useful when your content is meant to be used as HTML. For example, you might have a few paragraphs of text and some images in the full instructions for a bounding box.

Use skip_autoescape sparingly. As a best practice for templates, avoid passing in functional code or markup with skip_autoescape unless you are absolutely sure that you have strict control over what's being passed. If you're passing user input, you could be opening your workers up to a cross-site scripting attack.

to_json

to_json encodes data that you provide to JavaScript Object Notation (JSON). If you provide an object, it serializes it.

grant_read_access

grant_read_access takes an Amazon Simple Storage Service (Amazon S3) URI and encodes it into an HTTPS URL with a short-lived access token for that resource. This makes it possible to display photo, audio, or video objects stored in S3 buckets that are not otherwise publicly accessible to workers.

Example Example of the to_json and grant_read_access filters

Input

```liquid
auto-escape: {{ "Have you read 'James & the Giant Peach'?" }}
explicit escape: {{ "Have you read 'James & the Giant Peach'?" | escape }}
explicit escape_once: {{ "Have you read 'James & the Giant Peach'?" | escape_once }}
skip_autoescape: {{ "Have you read 'James & the Giant Peach'?" | skip_autoescape }}
to_json: {{ jsObject | to_json }}
grant_read_access: {{ s3://examplebucket/myphoto.png | grant_read_access }}
```
Example

Output

```
avo-escape: Have you read 'James & the Giant Peach'?  
extic escape: Have you read 'James & the Giant Peach'?   
extic escape_once: Have you read 'James & the Giant Peach'?   
skip_autoescape: Have you read 'James & the Giant Peach'?   
to_json: { "point_number": 8, "coords": [ 59, 76 ] }   
grant_read_access: https://s3.amazonaws.com/examplebucket/myphoto.png?<access token and other params>
```

Example Example of an automated classification template.

To automate this simple text classification sample, include the Liquid tag `{{ task.input.source }}`. This example uses the crowd-classifier (p. 462) element.

```html
<script src="https://assets.crowd.aws/crowd-html-elements.js"></script>
<crowd-form>
  <crowd-classifier
    name="tweetFeeling"
    categories="["positive", "negative", "neutral", "cannot determine"]"
    header="Which term best describes this tweet?"
  >
    <classification-target>
      {{ task.input.source }}
    </classification-target>
  <full-instructions header="Analyzing a sentiment">
    Try to determine the feeling the author of the tweet is trying to express.
    If none seems to match, choose "other."
  </full-instructions>

  <short-instructions>
    Pick the term that best describes the sentiment of the tweet.
  </short-instructions>
</crowd-classifier>
</crowd-form>
```

Preview a Worker Task Template

To preview a custom worker task template, use the SageMaker RenderUiTemplate operation. You can use the RenderUiTemplate operation with the AWS CLI or your preferred AWS SDK. For documentation on the supported language specific SDK's for this API operation use the See Also section of the RenderUiTemplate.

Prerequisites

To preview your worker task template, the AWS Identity and Access Management (IAM) role Amazon Resource Name (ARN), or RoleArn, that you use must have permission to access to the S3 objects that are used by the template. To learn how to configure your role or user see Enable Worker Task Template Previews (p. 1588).

To preview your worker task template using the RenderUiTemplate operation:

1. Provide a RoleArn of the role with required policies attached to preview your custom template.
2. In the Input parameter of Task, provide A JSON object that contains values for the variables defined in the template. These are the variables that are substituted for the task.input.source.
variable. For example, if you define a variable task.input.text in your template, you can supply the variable in the JSON object as "text": "sample text".

3. In the **Content** parameter of **UiTemplate**, insert your template.

Once you've configured **RenderUiTemplate**, use your preferred SDK or the AWS CLI to submit a request to render your template. If your request was successful, the response will include **RenderedContent**, a Liquid template that renders the HTML for the worker UI.

**Important**

To preview your template, you need an IAM role with permissions to read Amazon S3 objects that get rendered on your user interface. For a sample policy that you can attach to your IAM role to grant these permissions, see [Enable Worker Task Template Previews](p. 1588).

## Creating Good Worker Instructions

Creating good instructions for your human review jobs improves your worker's accuracy in completing their task. You can modify the default instructions that are provided in the console when creating a human review workflow, or you can use the console to create a custom worker template and include your instructions in this template. The instructions are shown to the worker on the UI page where they complete their labeling task.

### Create Good Worker Instructions

There are three kinds of instructions in the Amazon Augmented AI console:

- **Task Description** – The description should provide a succinct explanation of the task.
- **Instructions** – These instructions are shown on the same webpage where workers complete a task. These instructions should provide an easy reference to show the worker the correct way to complete the task.
- **Additional Instructions** – These instructions are shown in a dialog box that appears when a worker chooses **View full instructions**. We recommend that you provide detailed instructions for completing the task, and include several examples showing edge cases and other difficult situations for labeling objects.

### Add Example Images to Your Instructions

Images provide useful examples for your workers. To add a publicly accessible image to your instructions, do the following:

1. Place the cursor where the image should go in the instructions editor.
2. Choose the image icon in the editor toolbar.
3. Enter the URL of your image.

If your instruction image is in an S3 bucket that isn't publicly accessible, do the following:

- For the image URL, enter: `{ { 'https://s3.amazonaws.com/your-bucket-name/image-file-name' | grant_read_access } }`.

This renders the image URL with a short-lived, one-time access code that's appended so the worker's browser can display it. A broken image icon is displayed in the instructions editor, but previewing the tool displays the image in the rendered preview. See [grant_read_access](p. 1571) for more information about the `grant_read_access` element.
Monitor and Manage Your Human Loop

Once you've started a human review loop, you can check the results of and manage the loop using the Amazon Augmented AI Runtime API. Additionally, Amazon A2I integrates with Amazon EventBridge (also known as Amazon CloudWatch Events) to alert you when a human review loop changes status.

Use the procedures below to learn how to use the Amazon A2I Runtime API to monitor and manage your human loops. See Use Amazon CloudWatch Events in Amazon Augmented AI (p. 1589) to learn how Amazon A2I integrates with Amazon EventBridge.

To check your output data:

1. Check the results of your human loop by calling the DescribeHumanLoop operation. The result of this API operation contains information about the reason for and outcome of the loop activation.
2. Check the output data from your human loop in Amazon Simple Storage Service (Amazon S3). The path to the data uses the following pattern where YYYY/MM/DD/hh/mm/ss represents the human loop creation date with year (YYYY), month (MM) and day (DD) and the creation time with hour (hh), minute (mm) and second (ss).

   s3://customer-output-bucket-specified-in-flow-definition/flow-definition-name/YYYY/MM/DD/hh/mm/ss/human-loop-name/output.json

You can integrate this structure with AWS Glue or Amazon Athena to partition and analyze your output data. For more information, see Managing Partitions for ETL Output in AWS Glue.

To learn more about Amazon A2I output data format, see Amazon A2I Output Data (p. 1574).

To stop and delete your human loop:

1. Once a human loop has been started, you can stop your human loop by calling the StopHumanLoop operation using the HumanLoopName. If a human loop was successfully stopped, the server sends back an HTTP 200 response.
2. To delete a human loop for which the status equals Failed, Completed, or Stopped, use the DeleteHumanLoop operation.

To list human loops:

1. You can list all active human loops by calling the ListHumanLoops operation. You can filter human loops by the creation date of the loop using the CreationTimeAfter and CreateTimeBefore parameters.
2. If successful, ListHumanLoops will return HumanLoopSummaries and NextToken objects in the response element. HumanLoopSummaries contains information about a single human loop. For example, it will list a loop's status and if applicable, failure reason.

   Use the string returned in NextToken as an input in a subsequent call to ListHumanLoops to see the next page of human loops.

Amazon A2I Output Data

When your machine learning workflow sends Amazon A2I a data object, a human loop is created and human reviewers receive a task to review that data object. The output data from each human review task is stored in the Amazon Simple Storage Service (Amazon S3) output bucket you specify in your human review workflow. The path to the data uses the following pattern where YYYY/MM/DD/hh/mm/ss
represents the human loop creation date with year (YYYY), month (MM) and day (DD) and the creation
time with hour (hh), minute (mm) and second (ss).

```
s3://customer-output-bucket-specified-in-flow-definition/flow-definition-name/YYYY/MM/DD/hh/mm/ss/human-loop-name/output.json
```

The content of your output data depends on the type of task type (built-in or custom) and the type of
workforce you use. Your output data always includes the response from the human worker. Additionally,
output data may also include metadata about the human loop, the human reviewer (worker), and the
data object.

Use the following sections to learn more about Amazon A2I output data format for different task types
and workforces.

### Output Data From Built-In Task Types

Amazon A2I built-in task types include Amazon Textract and Amazon Rekognition. In addition to human
responses, the output data from one of these tasks includes details about the reason the human loop
was created and information about the integrated service used to create the human loop. Use the
following table to learn more about the output data schema for all built-in task types. The value for each
of these parameters depends on the service you use with Amazon A2I. Refer to the second table in this
section for more information about these service-specific values.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value Type</th>
<th>Example Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>awsManagedHumanLoopRequestSource</td>
<td>String</td>
<td>AWS/Rekognition/DetectModerationLabels/Image/V3 or AWS/Textract/AnalyzeDocument/Forms/V1</td>
<td>The API operation and associated AWS services that requested Amazon A2I to create the human loop. This is the API operation you use to configure your Amazon A2I human loop.</td>
</tr>
<tr>
<td>flowDefinitionArn</td>
<td>String</td>
<td>arn:aws:sagemaker:us-west-2:11112223333:flow-definition/flow-definition-name</td>
<td>The Amazon Resource Number (ARN) of the human review workflow (flow definition) used to create the human loop.</td>
</tr>
<tr>
<td>humanAnswers</td>
<td>List of JSON objects</td>
<td><code>{ &quot;answerContent&quot;: { &quot;AWS/Rekognition/DetectModerationLabels/Image/V3&quot;: { &quot;moderationLabels&quot;: [...] } },</code> or <code>{ &quot;answerContent&quot;: { }</code></td>
<td>A list of JSON objects that contain worker responses in answerContent. This object also contains submission details and, if a private workforce was used, worker metadata. To learn more, see Track Worker Activity (p. 1583). For human loop output data produced from</td>
</tr>
</tbody>
</table>
### Output Data From Built-In Task Types

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value Type</th>
<th>Example Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>humanLoopName</td>
<td>String</td>
<td>'human-loop-name'</td>
<td>The name of the human loop.</td>
</tr>
<tr>
<td>inputContent</td>
<td>JSON object</td>
<td>{</td>
<td>The input content the AWS service sent to Amazon A2I when it requested a human loop be created.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;aiServiceRequest&quot;: { },</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;aiServiceResponse&quot;: { },</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;humanTaskActivationConditionResults&quot;: { },</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;selectedAiServiceResponse&quot;: { }</td>
<td></td>
</tr>
<tr>
<td>aiServiceRequest</td>
<td>JSON object</td>
<td>{</td>
<td>The original request sent to the AWS service integrated with Amazon A2I. For example, if you use Amazon Rekognition with Amazon A2I, this includes the request made through the API operation DetectModerationLabels. For Amazon Textract integrations, this includes the request made through AnalyzeDocument.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;document&quot;: { },</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;featureTypes&quot;: [ ],</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;humanLoopConfig&quot;: { }</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>or</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>{</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;image&quot;: { },</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;humanLoopConfig&quot;: { }</td>
<td></td>
</tr>
<tr>
<td>Parameter</td>
<td>Value Type</td>
<td>Example Values</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------</td>
<td>------------</td>
<td>----------------</td>
<td>-------------</td>
</tr>
<tr>
<td>aiServiceResponse</td>
<td>JSON object</td>
<td>{</td>
<td>The full response from the AWS service. This is the data that is used to determine if a human review is required. This object may contain metadata about the data object that is not shared with human reviewers.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;moderationLabels&quot;: [...],&quot;moderationModelVersion&quot;: &quot;3.0&quot;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>or</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>{  &quot;blocks&quot;: [...],  &quot;documentMetadata&quot;: {}}</td>
<td></td>
</tr>
<tr>
<td>selectedAiServiceResponse</td>
<td>JSON object</td>
<td>{</td>
<td>The subset of the aiServiceResponse that matches the activation conditions in ActivationConditions. All data objects listed in aiServiceResponse are listed in selectedAiServiceResponse when inferences are randomly sampled, or all inferences triggered activation conditions.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;moderationLabels&quot;: [...],&quot;moderationModelVersion&quot;: &quot;3.0&quot;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>or</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>{  &quot;blocks&quot;: [...],  &quot;documentMetadata&quot;: {}}</td>
<td></td>
</tr>
<tr>
<td>Parameter</td>
<td>Value Type</td>
<td>Example Values</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>-----------------</td>
<td>-------------------------------------------------------------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>humanTaskActivationConditionResults</td>
<td>JSON object</td>
<td>{ &quot;Conditions&quot;: [...] }</td>
<td>A JSON object in inputContent that contains the reason a human loop was created. This includes a list of the activation conditions (Conditions) included in your human review workflow (flow definition), and the evaluation result for each condition—this result is either true or false. To learn more about activation conditions, see JSON Schema for Human Loop Activation Conditions in Amazon Augmented AI (p. 1545).</td>
</tr>
</tbody>
</table>

Select a tab on the following table to learn about the task type-specific parameters and see an example output-data code block for each of the built-in task types.

**Amazon Textract Task Type Output Data**

When you use the Amazon Textract built-in integration, you see 'AWS/Textract/AnalyzeDocument/Forms/V1' as the value for awsManagedHumanLoopRequestSource in your output data.

The answerContent parameter contains a Block object that includes human responses for all blocks sent to Amazon A2I.

The aiServiceResponse parameter also includes a Block object with Amazon Textract's response to the original request sent using to AnalyzeDocument.

To learn more about the parameters you see in the block object, refer to Block in the Amazon Textract Developer Guide.

The following is an example of the output data from an Amazon A2I human review of Amazon Textract document analysis inferences.

```json
{
  "awsManagedHumanLoopRequestSource": "AWS/Textract/AnalyzeDocument/Forms/V1",
  "humanAnswers": [
    {
      "answerContent": {
        "AWS/Textract/AnalyzeDocument/Forms/V1": {
          "blocks": [...]
        }
      },
      "submissionTime": "2020-09-28T19:17:59.880Z",
      "workerId": "111122223333",
```
"workerMetadata": {
"identityData": {
"identityProviderType": "Cognito",
"sub": "c6a8eb7-9944-42e9-a6b9-111122223333"
}
},
"humanLoopName": "human-loop-name",
"inputContent": {
"aiServiceRequest": {
"document": {
"s3Object": {
"bucket": "DOC-EXAMPLE-BUCKET1",
"name": "document-demo.jpg"
}
},
"featureTypes": [
"TABLES",
"FORMS"
],
"humanLoopConfig": {
"dataAttributes": {
"contentClassifiers": [
"FreeOfPersonallyIdentifiableInformation"
]
},
"humanLoopName": "human-loop-name"
},
"aiServiceResponse": {
"blocks": [...],
"documentMetadata": {
"pages": 1
}
},
"humanTaskActivationConditionResults": {
"Conditions": [
{
"EvaluationResult": true,
"Or": [
"ConditionParameters": {
"ImportantFormKey": "Mail address",
"ImportantFormKeyAliases": [
"Mail Address:",
"Mail address:",
"Mailing Add:",
"Mailing Addresses"
],
"KeyValueBlockConfidenceLessThan": 100,
"WordBlockConfidenceLessThan": 100
},
"ConditionType": "ImportantFormKeyConfidenceCheck",
"EvaluationResult": true
},
"ConditionParameters": {
"ImportantFormKey": "Mail address",
"ImportantFormKeyAliases": [
"Mail Address:",
"Mail address:"
],
"KeywordConfidenceLessThan": 100,
"WordBlockConfidenceLessThan": 100
},
"ConditionType": "KeywordConfidenceCheck",
"EvaluationResult": true
}
}
Amazon SageMaker Developer Guide
Output Data From Built-In Task Types

Amazon Rekognition Task Type Output Data

When you use the Amazon Textract built-in integration, you see the string 'AWS/Rekognition/DetectModerationLabels/Image/V3' as the value for awsManagedHumanLoopRequestSource in your output data.

The answerContent parameter contains a moderationLabels object that contains human responses for all moderation labels sent to Amazon A2I.

The aiServiceResponse parameter also includes a moderationLabels object with Amazon Rekognition’s response to the original request sent to DetectModerationLabels.

To learn more about the parameters you see in the block object, refer to ModerationLabel in the Amazon Rekognition Developer Guide.

The following is an example of the output data from an Amazon A2I human review of Amazon Rekognition image moderation inferences.

```json
{
    "awsManagedHumanLoopRequestSource": "AWS/Rekognition/DetectModerationLabels/Image/V3",
    "humanAnswers": [
        {
            "answerContent": {
                "AWS/Rekognition/DetectModerationLabels/Image/V3": {
                    "moderationLabels": [...]
                }
            },
            "submissionTime": "2020-09-28T19:22:35.508Z",
            "workerId": "ef7294f850a3d9d1",
            "workerMetadata": {
                "identityData": {
                    "identityProviderType": "Cognito",
                    "sub": "c6a8eb7-9944-42e9-a6b9-111122223333"
                }
            }
        }
    ],
    "humanLoopName": "human-loop-name",
    "inputContent": {
        "aiServiceRequest": {
            "humanLoopConfig": {
```
Output Data From Custom Task Types

When you add Amazon A2I to a custom human review workflow, you see the following parameters in the output data returned from human review tasks.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>flowDefinitionArn</td>
<td>String</td>
<td>The Amazon Resource Number (ARN) of the human review</td>
</tr>
<tr>
<td>humanLoopName</td>
<td></td>
<td></td>
</tr>
<tr>
<td>image.bucket</td>
<td></td>
<td></td>
</tr>
<tr>
<td>image.name</td>
<td></td>
<td></td>
</tr>
<tr>
<td>moderationLabels</td>
<td></td>
<td></td>
</tr>
<tr>
<td>moderationModelVersion</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EvaluationResult</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ConditionParameters</td>
<td></td>
<td></td>
</tr>
<tr>
<td>confidenceLessThan</td>
<td></td>
<td></td>
</tr>
<tr>
<td>suggested</td>
<td></td>
<td></td>
</tr>
<tr>
<td>confidenceGreaterThan</td>
<td></td>
<td></td>
</tr>
<tr>
<td>female swimwear or underwear</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ConditionType</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EvaluationResult</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ConditionParameters</td>
<td></td>
<td></td>
</tr>
<tr>
<td>confidenceLessThan</td>
<td></td>
<td></td>
</tr>
<tr>
<td>suggested</td>
<td></td>
<td></td>
</tr>
<tr>
<td>confidenceGreaterThan</td>
<td></td>
<td></td>
</tr>
<tr>
<td>female swimwear or underwear</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ConditionType</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EvaluationResult</td>
<td></td>
<td></td>
</tr>
<tr>
<td>moderationLabels</td>
<td></td>
<td></td>
</tr>
<tr>
<td>confidence</td>
<td></td>
<td></td>
</tr>
<tr>
<td>name</td>
<td></td>
<td></td>
</tr>
<tr>
<td>parentName</td>
<td></td>
<td></td>
</tr>
<tr>
<td>moderationModelVersion</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The following is an example of output data from a custom integration with Amazon A2I and Amazon Transcribe. In this example, the `inputContent` consists of:

- A path to an `.mp4` file in Amazon S3 and the video title
- The transcription returned from Amazon Transcribe (parsed from Amazon Transcribe output data)
- A start and end time used by the worker task template to clip the `.mp4` file and show workers a relevant portion of the video

```json
{
    "humanAnswers": [
        {
            "answerContent": {
                "transcription": "use lambda to turn your notebook"
            },
            "submissionTime": "2020-06-18T17:08:26.246Z",
            "workerId": "ef7294f850a3d9d1",
            "workerMetadata": {
                "identityData": {
                    "identityProviderType": "Cognito",
                    "sub": "c6aa8eb7-9944-42e9-a6b9-111122223333"
                }
            }
        }
    ],
    "humanLoopName": "human-loop-name",
    "inputContent": {
        "audioPath": "s3://DOC-EXAMPLE-BUCKET1/a2i_transcribe_demo/Fully-Managed Notebook Instances with Amazon SageMaker - a Deep Dive.mp4",
        "end_time": 950.27,
        "original_words": "but definitely use Lambda to turn your ",
        "start_time": 948.51,
    }
}
```
Track Worker Activity

Amazon A2I provides information that you can use to track individual workers in task output data. To identify the worker that worked on the human review task, use the following from the output data in Amazon S3:

- The `acceptanceTime` is the time that the worker accepted the task. The format of this date and time stamp is `YYYY-MM-DDTHH:MM:SS.mmmZ` for the year (`YYYY`), month (`MM`), day (`DD`), hour (`HH`), minute (`MM`), second (`SS`) and millisecond (`mmm`). The date and time are separated by a `T`.
- The `submissionTime` is the time that the worker submitted their annotations using the `Submit` button. The format of this date and time stamp is `YYYY-MM-DDTHH:MM:SS.mmmZ` for the year (`YYYY`), month (`MM`), day (`DD`), hour (`HH`), minute (`MM`), second (`SS`) and millisecond (`mmm`). The date and time are separated by a `T`.
- `timeSpentInSeconds` reports the total time, in seconds, that a worker worked on that task. This is the `submissionTime` subtracted from the `acceptanceTime`.
- The `workerId` is unique to each worker.
- If you use a private workforce, in `workerMetadata`, you see the following.
  - The `identityProviderType` is the service used to manage the private workforce.
  - The `issuer` is the Cognito user pool or OIDC Identity Provider (IdP) issuer associated with the work team assigned to this human review task.
  - A unique `sub` identifier refers to the worker. If you create a workforce using Amazon Cognito, you can retrieve details about this worker (such as the name or user name) using this ID using Amazon Cognito. To learn how, see Managing and Searching for User Accounts in Amazon Cognito Developer Guide.

The following is an example of the output you may see if you use Amazon Cognito to create a private workforce. This is identified in the `identityProviderType`.

```json
"submissionTime": "2020-12-28T18:28:39Z",
"acceptanceTime": "2020-12-28T18:59:58Z",
"timeSpentInSeconds": 43,
"workerId": "a12b3cdefg4h5i67",
"workerMetadata": {
  "identityData": {
    "identityProviderType": "Cognito",
    "sub": "aaaaaaaa-bbbb-cccc-dddd-eeeeeeeeeeee"
  }
}
```

The following is an example of the output you may see if you use your own OIDC IdP to create a private workforce:

```json
"workerId": "a12b3cdefg4h5i67",
"workerMetadata": {
  "identityData": {
    "identityProviderType": "Oidc",
    "issuer": "https://example-oidc-ipd.com/adfs",
    "sub": "aaaaaaaa-bbbb-cccc-dddd-eeeeeeeeeeee"
  }
}
```
Permissions and Security in Amazon Augmented AI

When using Amazon Augmented AI (Amazon A2I) to create a human review workflow for your ML/AI application, you create and configure resources in Amazon SageMaker such as a human workforce and worker task templates. To configure and start a human loop, you will either integrate Amazon A2I with other AWS services such as Amazon Textract or Amazon Rekognition or use the Amazon Augmented AI Runtime API. To create a human review workflow and start a human loop, you will need to attach certain policies to your AWS Identity and Access Management (IAM) role or user. Specifically:

• When you start a human loop using image input data on or after January 12th, 2020, you must add a CORS header policy to the S3 bucket that contains your input data. See CORS Permission Requirement (p. 1585) to learn more.

• When you create a flow definition, you need to provide a role that grants Amazon A2I permission to access Amazon S3 both for reading objects that will be rendered in a human task UI and for writing the results of the human review.

This role will also need to have a trust policy attached to give SageMaker permission to assume the role. This allows Amazon A2I to perform actions in accordance with permissions that you attach to the role.

See Add Permissions to the IAM Role Used to Create a Flow Definition (p. 1585) for example policies that you can modify and attach to the role you use to create a flow definition. These are the policies that will be attached to the IAM role that is created in the Human review workflows section of the Amazon A2I area of the SageMaker console.

• To create and start human loops, you either use an API operation from a built-in task type (such as DetectModerationLabel or AnalyzeDocument) or the Amazon A2I Runtime API operation StartHumanLoop in a custom ML application. You need to attach the AmazonAugmentedAIFullAccess managed policy to the IAM user that invokes these API operations to grant permission to these services to use Amazon A2I operations. To learn how, see Create an IAM User That Can Invoke Amazon A2I API Operations (p. 1587).

This policy does not grant permission to invoke the API operations of the AWS service associated with built-in task types. For example, AmazonAugmentedAIFullAccess does not grant permission to call Amazon Rekognition API operation DetectModerationLabel or Amazon Textract API operation AnalyzeDocument. You can use the more general policy, AmazonAugmentedAIIntegratedAPIAccess, to grant these permissions. For more information, see Create an IAM User With Permissions to Invoke Amazon A2I, Amazon Textract, and Amazon Rekognition API Operations (p. 1587). This is a good option when you want to grant an IAM user broad permissions to use Amazon A2I and integrated AWS services' API operations.

If you want to configure more granular permissions, see Amazon Rekognition Identity-Based Policy Examples and Amazon Textract Identity-Based Policy Examples for identity-based policies you can use to grant permission to use these individual services.

• To preview your custom worker task UI template, you need an IAM role with permissions to read Amazon S3 objects that get rendered on your user interface. See a policy example in Enable Worker Task Template Previews (p. 1588).

Topics
• CORS Permission Requirement (p. 1585)
• Add Permissions to the IAM Role Used to Create a Flow Definition (p. 1585)
CORS Permission Requirement

Earlier in 2020, widely used browsers like Chrome and Firefox changed their default behavior for rotating images based on image metadata, referred to as EXIF data. Previously, images would always display in browsers exactly how they are stored on disk, which is typically unrotated. After the change, images now rotate according to a piece of image metadata called orientation value. This has important implications for the entire machine learning (ML) community. For example, if the EXIF orientation is not considered, applications that are used to annotate images may display images in unexpected orientations and result in incorrect labels.

It is estimated that, starting with Chrome 88 on January 19th, 2021, AWS can no longer automatically prevent the rotation of images because the web standards group W3C has decided that the ability to control rotation of images violates the web's Same Origin Policy. Therefore, to ensure human workers annotate your input images in a predictable orientation when you submit requests to create a human loop, you must add a CORS header policy to the S3 buckets that contain your input images by January 12th, 2021.

Important
If you do not add a CORS configuration to the S3 buckets that contains your input data by January 12th, 2021, human review tasks for those input data objects will fail.

You can add a CORS policy to an S3 bucket that contains input data in the S3 console. To set the required CORS headers on the S3 bucket that contain your input images in the S3 console, follow the directions detailed in How do I add cross-domain resource sharing with CORS?. Use the following CORS configuration code for the buckets that hosts your images. (Choose JSON or XML based on your preference; they both accomplish the same configuration.)

**JSON**

```
[
    "AllowedHeaders": [],
    "AllowedMethods": ["GET"],
    "AllowedOrigins": ["*"],
    "ExposeHeaders": []
]
```

**XML**

```
<CORSConfiguration>
  <CORSRule>
    <AllowedOrigin>*</AllowedOrigin>
    <AllowedMethod>GET</AllowedMethod>
  </CORSRule>
</CORSConfiguration>
```

Add Permissions to the IAM Role Used to Create a Flow Definition

To create a flow definition, attach the policies in this section to the role that you use when creating a human review workflow in the SageMaker console, or using the CreateFlowDefinition API operation.
If you are using the console to create a human review workflow, enter the role Amazon Resource Name (ARN) in the IAM role field when creating a human review workflow in the console.

When creating a flow definition using the API, attach these policies to the role that is passed to the RoleArn parameter of the CreateFlowDefinition operation.

When you create a human review workflow (flow definition), Amazon A2I invokes Amazon S3 to complete your task. To grant Amazon A2I permission to retrieve and store your files in your Amazon S3 bucket, create the following policy and attach it to your role. For example, if the images, documents, and other files that you are sending for human review are stored in an S3 bucket named my_input_bucket, and if you want the human reviews to be stored in a bucket named my_output_bucket, you would create the following policy.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": ["s3:GetObject"],
            "Resource": ["arn:aws:s3:::my_input_bucket/*"]
        },
        {
            "Effect": "Allow",
            "Action": ["s3:PutObject"],
            "Resource": ["arn:aws:s3:::my_output_bucket/*"]
        }
    ]
}
```

In addition, the IAM role must have the following trust policy to give SageMaker permission to assume the role. To learn more about IAM trust policies, see Resource-Based Policies section of Policies and Permissions in the AWS Identity and Access Management documentation.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "AllowSageMakerToAssumeRole",
            "Effect": "Allow",
            "Principal": {
                "Service": "sagemaker.amazonaws.com"
            },
            "Action": "sts:AssumeRole"
        }
    ]
}
```

For more information about creating and managing IAM roles and policies, see the following topics in the AWS Identity and Access Management User Guide:

- To create IAM role, see Creating a Role to Delegate Permissions to an IAM User.
- To learn how to create IAM policies, see Creating IAM Policies.
Create an IAM User That Can Invoke Amazon A2I API Operations

To use Amazon A2I to create and start human loops for Amazon Rekognition, Amazon Textract, or the Amazon A2I runtime API, you must use an IAM user that has permissions to invoke Amazon A2I operations. To do this, use the IAM console to attach the AmazonAugmentedAIFullAccess managed policy to a new or existing IAM user.

This policy grants permission to an IAM user to invoke API operations from the SageMaker API for flow definition creation and management and the Amazon Augmented AI Runtime API for human loop creation and management. To learn more about these API operations, see Use APIs in Amazon Augmented AI.

AmazonAugmentedAIFullAccess does not grant permissions to use Amazon Rekognition or Amazon Textract API operations.

Note
You can also attach the AmazonAugmentedAIFullAccess to an IAM role that is used to create and start a human loop.

To create the required IAM user

2. Choose Users and choose an existing user, or create a new user by choosing Add user. To learn how to create a new user, see Creating an IAM User in Your AWS Account in the AWS Identity and Access Management User Guide.
   - If you chose to attach the policy to an existing user, choose Add permissions.
   - While creating a new user, follow the next step on the Set permissions page.
3. Choose Attach existing policies directly.
4. In the Search bar, enter AmazonAugmentedAIFullAccess and check the box next to that policy.

   To enable this IAM user to create a flow definition with the public work team, also attach the AmazonSageMakerMechanicalTurkAccess managed policy.
5. After attaching the policy or policies:
   a. If you are using an existing user, choose Next: Review, and then choose Add permissions.
   b. If you are creating a new user, choose Next: Tags and complete the process of creating your user.

For more information, see Adding and Removing IAM Identity Permissions in the AWS Identity and Access Management User Guide.

Create an IAM User With Permissions to Invoke Amazon A2I, Amazon Textract, and Amazon Rekognition API Operations

To create an IAM user that has permission to invoke the API operations used by the built-in task types (that is, DetectModerationLables for Amazon Rekognition and AnalyzeDocument for Amazon
Enable Worker Task Template Previews

To customize the interface and instructions that your workers see when working on your tasks, you create a worker task template. You can create the template using the CreateHumanTaskUi operation or the SageMaker console.

To preview your template, you need an IAM role with the following permissions to read Amazon S3 objects that get rendered on your user interface.

```json
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": ["s3:GetObject"],
            "Resource": [
                "arn:aws:s3:::my_input_bucket/*"
            ]
        }
    ]
}
```
For Amazon Rekognition and Amazon Textract task types, you can preview your template using the Amazon Augmented AI section of the SageMaker console. For custom task types, you preview your template by invoking the `RenderUiTemplate` operation. To preview your template, follow the instructions for your task type:

- Amazon Rekognition and Amazon Textract task types – In the SageMaker console, use the role’s Amazon Resource Name (ARN) in the procedure documented in Create a Worker Task Template (p. 1565).
- Custom task types – In the `RenderUiTemplate` operation, use the role’s ARN in the `RoleArn` parameter.

**Additional Permissions and Security Resources**

- the section called “Control Access to SageMaker Resources by Using Tags” (p. 1644).
- the section called “SageMaker Identity-Based Policies” (p. 1622)
- the section called “Control Creation of SageMaker Resources with Condition Keys” (p. 1635)
- the section called “Amazon SageMaker API Permissions Reference” (p. 1663)
- Security (p. 1611)

**Use Amazon CloudWatch Events in Amazon Augmented AI**

Amazon Augmented AI uses Amazon CloudWatch Events (CloudWatch Events) to alert you when a human review loop changes status. When a review loop changes to the Completed, Failed, or Stopped status, Augmented AI sends an event to CloudWatch Events similar to the following:

```json
{
  "version": "0",
  "id": "12345678-1111-2222-3333-12345EXAMPLE",
  "detail-type": "SageMaker A2I HumanLoop Status Change",
  "source": "aws.sagemaker",
  "account": "111111111111",
  "time": "2019-11-14T17:49:25Z",
  "region": "us-east-1",
  "resources": ["arn:aws:sagemaker:us-east-1:111111111111:human-loop/humanloop-nov-14-1"],
  "detail": {
    "creationTime": "2019-11-14T17:36:40Z",
    "failureCode": null,
    "failureReason": null,
    "flowDefinitionArn": "arn:aws:sagemaker:us-east-1:111111111111:flow-definition/flowdef-nov-12",
    "humanLoopName": "humanloop-nov-14-1",
    "humanLoopOutput": {
      "outputS3Uri": "s3://customer-output-bucket-specified-in-flow-definition/flowdef-nov-12/2019/11/14/17/36/36/humanloop-nov-14-1/output.json"
    },
    "humanLoopStatus": "Completed"
  }
}
```

The details in the JSON output include the following:
creationTime

The timestamp when Augmented AI created the human loop.

failureCode

A failure code denoting a specific type of failure.

failureReason

The reason why a human loop has failed. The failure reason is only returned when the human review loop status is failed.

flowDefinitionArn

The Amazon Resource Name (ARN) of the flow definition, or human review workflow.

humanLoopArn

The Amazon Resource Name (ARN) of the human loop.

humanLoopName

The name of the human loop.

humanLoopOutput

An object containing information about the output of the human loop.

outputS3Uri

The location of the Amazon S3 object where Augmented AI stores your human loop output.

humanLoopStatus

The status of the human loop.

Send Events from Your Human Loop to CloudWatch Events

To configure a CloudWatch Events rule to get status updates, or events, for your Amazon A2I human loops, use the AWS Command Line Interface (AWS CLI) `put-rule` command. When using the `put-rule` command, specify the following to receive human loop statuses:

- \"source\":[\"aws.sagemaker\"]
- \"detail-type\":[\"SageMaker A2I HumanLoop Status Change\"]

To configure a CloudWatch Events rule to watch for all status changes, use the following command and replace the placeholder text. For example, replace "A2IHumanLoopStatusChanges" with a unique CloudWatch Events rule name and "arn:aws:iam::111122223333:role/MyRoleForThisRule" with the Amazon Resource Number (ARN) of an IAM role with an events.amazonaws.com trust policy attached. Replace `region` with the AWS Region you want to create the rule in.

```
aws events put-rule --name "A2IHumanLoopStatusChanges"
--event-pattern "{"source\":["aws.sagemaker"],"detail-type\":["SageMaker A2I HumanLoop Status Change"]}"
--role-arn "arn:aws:iam::111122223333:role/MyRoleForThisRule"
--region "region"
```

To learn more about the `put-rule` request, see Event Patterns in CloudWatch Events in the Amazon CloudWatch Events User Guide.
Set Up a Target to Process Events

To process events, you need to set up a target. For example, if you want to receive an email when a human loop status changes, use a procedure in Setting Up Amazon SNS Notifications in the Amazon CloudWatch User Guide to set up an Amazon SNS topic and subscribe your email to it. Once you have create a topic, you can use it to create a target.

To add a target to your CloudWatch Events rule

1. Open the CloudWatch console: https://console.aws.amazon.com/cloudwatch/home
2. In the navigation pane, choose Rules.
3. Choose the rule that you want to add a target to.
4. Choose Actions, and then choose Edit.
5. Under Targets, choose Add Target and choose the AWS service you want to act when a human loop status change event is detected.
6. Configure your target. For instructions, see the topic for configuring a target in the AWS documentation for that service.
7. Choose Configure details.
8. For Name, enter a name and, optionally, provide details about the purpose of the rule in Description.
9. Make sure that the check box next to State is selected so that your rule is listed as Enabled.
10. Choose Update rule.

Use Human Review Output

After you receive human review results, you can analyze the results and compare them to machine learning predictions. The JSON that is stored in the Amazon S3 bucket contains both the machine learning predictions and the human review results.

More Information

Automating Amazon SageMaker with Amazon EventBridge (p. 1715)

Use APIs in Amazon Augmented AI

You can create a human review workflow or a worker task template programmatically. The APIs you use depend on whether you are creating a Amazon Rekognition, Amazon Textract, or custom task type. This topic provides links to API reference documentation for each task type and programming task.

The following APIs can be used with Augmented AI:

Amazon Augmented AI

Use the Augmented AI API to start, stop, and delete human review loops. You can also list all human review loops and return information about human review loops in your account.

Learn more about human review loop APIs in the Amazon Augmented AI Runtime API Reference.

Amazon Rekognition

Use the HumanLoopConfig parameter of the DetectModerationLabels API to trigger a human review workflow using Amazon Rekognition.
Amazon SageMaker

Use the Amazon SageMaker API to create a FlowDefinition, also known as a human review workflow. You can also create a HumanTaskUi, or worker task template.

For more information, see the CreateFlowDefinition or the CreateHumanTaskUi API documentation.

Amazon Textract

Use the HumanLoopConfig parameter of the AnalyzeDocument API to trigger a human review workflow using Amazon Textract.

Programmatic Walkthroughs

The following walkthroughs and tutorials provide example code and step-by-step instructions for creating human review workflows and worker task templates programmatically.

- the section called “Create a Human Review Workflow (API)” (p. 1543)
- Using Amazon Augmented AI with Amazon Rekognition in the Amazon Rekognition Developer Guide
- Using Amazon Augmented AI with Amazon Textract AnalyzeDocument in the Amazon Textract Developer Guide
Buy and Sell Amazon SageMaker Algorithms and Models in AWS Marketplace

This feature is not available in the China Regions.

Amazon SageMaker integrates with AWS Marketplace, enabling developers to charge other SageMaker users for the use of their algorithms and model packages. AWS Marketplace is a curated digital catalog that makes it easy for customers to find, buy, deploy, and manage third-party software and services that customers need to build solutions and run their businesses. AWS Marketplace includes thousands of software listings in popular categories, such as security, networking, storage, machine learning, business intelligence, database, and DevOps. It simplifies software licensing and procurement with flexible pricing options and multiple deployment methods.

For information, see AWS Marketplace Documentation.

Topics

- SageMaker Algorithms (p. 1593)
- SageMaker Model Packages (p. 1593)
- Sell Amazon SageMaker Algorithms and Model Packages (p. 1607)
- Find and Subscribe to Algorithms and Model Packages on AWS Marketplace (p. 1610)
- Use Algorithm and Model Package Resources (p. 1600)

SageMaker Algorithms

An algorithm enables you to perform end-to-end machine learning. It has two logical components: training and inference. Buyers can use the training component to create training jobs in SageMaker and build a machine learning model. SageMaker saves the model artifacts generated by the algorithm during training to an Amazon S3 bucket. For more information, see Train a Model with Amazon SageMaker (p. 8).

Buyers use the inference component with the model artifacts generated during a training job to create a deployable model in their SageMaker account. They can use the deployable model for real-time inference by using SageMaker hosting services. Or, they can get inferences for an entire dataset by running batch transform jobs. For more information, see Deploy a Model in Amazon SageMaker (p. 11).

SageMaker Model Packages

Buyers use a model package to build a deployable model in SageMaker. They can use the deployable model for real-time inference by using SageMaker hosting services. Or, they can get inferences for an entire dataset by running batch transform jobs. For more information, see Deploy a Model in Amazon SageMaker (p. 11). As a seller, you can build your model artifacts by training in SageMaker, or you can use your own model artifacts from a model that you trained outside of SageMaker. You can charge buyers for inference.
Use your own algorithms and models with the AWS Marketplace

The following sections show how to create algorithm and model package resources that you can use locally and publish to the AWS Marketplace.

Topics
- Create Algorithm and Model Package Resources (p. 1594)
- Use Algorithm and Model Package Resources (p. 1600)

Create Algorithm and Model Package Resources

After your training and/or inference code is packaged in Docker containers, create algorithm and model package resources that you can use in your Amazon SageMaker account and, optionally, publish on AWS Marketplace.

Topics
- Create an Algorithm Resource (p. 1594)
- Create a Model Package Resource (p. 1597)

Create an Algorithm Resource

To create an algorithm resource that you can use to run training jobs in Amazon SageMaker and publish on AWS Marketplace specify the following information:

- The Docker containers that contains the training and, optionally, inference code.
- The configuration of the input data that your algorithm expects for training.
- The hyperparameters that your algorithm supports.
- Metrics that your algorithm sends to Amazon CloudWatch during training jobs.
- The instance types that your algorithm supports for training and inference, and whether it supports distributed training across multiple instances.
- Validation profiles, which are training jobs that SageMaker uses to test your algorithm's training code and batch transform jobs that SageMaker runs to test your algorithm's inference code.

To ensure that buyers and sellers can be confident that products work in SageMaker, we require that you validate your algorithms before listing them on AWS Marketplace. You can list products in the AWS Marketplace only if validation succeeds. To validate your algorithms, SageMaker uses your validation profile and sample data to run the following validations tasks:

1. Create a training job in your account to verify that your training image works with SageMaker.
2. If you included inference code in your algorithm, create a model in your account using the algorithm's inference image and the model artifacts produced by the training job.
3. If you included inference code in your algorithm, create a transform job in your account using the model to verify that your inference image works with SageMaker.

When you list your product on AWS Marketplace, the inputs and outputs of this validation process persist as part of your product and are made available to your buyers. This helps buyers understand and evaluate the product before they buy it. For example, buyers can inspect the input data that you used, the outputs generated, and the logs and metrics emitted by your code. The more comprehensive your validation specification, the easier it is for customers to evaluate your product.
Note
In your validation profile, provide only data that you want to expose publicly.

Validation can take up to a few hours. To see the status of the jobs in your account, in the SageMaker console, see the Training jobs and Transform jobs pages. If validation fails, you can access the scan and validation reports from the SageMaker console. If any issues are found, you will have to create the algorithm again.

Note
To publish your algorithm on AWS Marketplace, at least one validation profile is required.

You can create an algorithm by using either the SageMaker console or the SageMaker API.

Topics
• Create an Algorithm Resource (Console) (p. 1595)
• Create an Algorithm Resource (API) (p. 1597)

Create an Algorithm Resource (Console)

To create an algorithm resource (console)
2. Choose Algorithms, then choose Create algorithm.
3. On the Training specifications page, provide the following information:
   a. For Algorithm name, type a name for your algorithm. The algorithm name must be unique in your account and in the AWS region. The name must have 1 to 64 characters. Valid characters are a-z, A-Z, 0-9, and - (hyphen).
   b. Type a description for your algorithm. This description appears in the SageMaker console and in the AWS Marketplace.
   c. For Training image, type the path in Amazon ECR where your training container is stored.
   d. For Support distributed training, Choose Yes if your algorithm supports training on multiple instances. Otherwise, choose No.
   e. For Support instance types for training, choose the instance types that your algorithm supports.
   f. For Channel specification, specify up to 8 channels of input data for your algorithm. For example, you might specify 3 input channels named train, validation, and test. For each channel, specify the following information:
      i. For Channel name, type a name for the channel. The name must have 1 to 64 characters. Valid characters are a-z, A-Z, 0-9, and - (hyphen).
      ii. To require the channel for your algorithm, choose Channel required.
      iii. Type a description for the channel.
      iv. For Supported input modes, choose Pipe mode if your algorithm supports streaming the input data, and File mode if your algorithm supports downloading the input data as a file. You can choose both.
      v. For Supported content types, type the MIME type that your algorithm expects for input data.
      vi. For Supported compression type, choose Gzip if your algorithm supports Gzip compression. Otherwise, choose None.
      vii. Choose Add channel to add another data input channel, or choose Next if you are done adding channels.
4. On the **Tuning specifications** page, provide the following information:

a. **For Hyperparameter specification**, specify the hyperparameters that your algorithm supports by editing the JSON object. For each hyperparameter that your algorithm supports, construct a JSON block similar to the following:

```json
{
  "DefaultValue": "5",
  "Description": "The first hyperparameter",
  "IsRequired": true,
  "IsTunable": false,
  "Name": "intRange",
  "Range": { 
    "IntegerParameterRangeSpecification": { 
      "MaxValue": "10",
      "MinValue": "1"
    },
    "Type": "Integer"
  }
}
```

In the JSON, supply the following:

i. **For DefaultValue**, specify a default value for the hyperparameter, if there is one.
ii. **For Description**, specify a description for the hyperparameter.
iii. **For IsRequired**, specify whether the hyperparameter is required.
iv. **For IsTunable**, specify `true` if this hyperparameter can be tuned when a user runs a hyperparameter tuning job that uses this algorithm. For information, see [Perform Automatic Model Tuning](p. 1023).

v. **For Name**, specify a name for the hyperparameter.
vi. **For Range**, specify one of the following:

   - **IntegerParameterRangeSpecification** - the values of the hyperparameter are integers. Specify minimum and maximum values for the hyperparameter.

   - **ContinuousParameterRangeSpecification** - the values of the hyperparameter are floating-point values. Specify minimum and maximum values for the hyperparameter.

   - **CategoricalParameterRangeSpecification** - the values of the hyperparameter are categorical values. Specify a list of all of the possible values.

vii. **For Type**, specify Integer, Continuous, or Categorical. The value must correspond to the type of Range that you specified.

b. **For Metric definitions**, specify any training metrics that you want your algorithm to emit. SageMaker uses the regular expression that you specify to find the metrics by parsing the logs from your training container during training. Users can view these metrics when they run training jobs with your algorithm, and they can monitor and plot the metrics in Amazon CloudWatch. For information, see [Monitor and Analyze Training Jobs Using Metrics](p. 1127). For each metric, provide the following information:

i. **For Metric name**, type a name for the metric.
ii. **For Regex**, type the regular expression that SageMaker uses to parse training logs so that it can find the metric value.

iii. **For Objective metric support** choose **Yes** if this metric can be used as the objective metric for a hyperparameter tuning job. For information, see [Perform Automatic Model Tuning](p. 1023).

iv. Choose **Add metric** to add another metric, or choose **Next** if you are done adding metrics.
Create an Algorithm Resource (API)

To create an algorithm resource by using the SageMaker API, call the CreateAlgorithm API.

Create a Model Package Resource

To create a model package resource that you can use to create deployable models in Amazon SageMaker and publish on AWS Marketplace specify the following information:

- The Docker container that contains the inference code, or the algorithm resource that was used to train the model.
- The location of the model artifacts. Model artifacts can either be packaged in the same Docker container as the inference code or stored in Amazon S3.
• The instance types that your model package supports for both real-time inference and batch transform jobs.

• Validation profiles, which are batch transform jobs that SageMaker runs to test your model package's inference code.

Before listing model packages on AWS Marketplace, you must validate them. This ensures that buyers and sellers can be confident that products work in Amazon SageMaker. You can list products on AWS Marketplace only if validation succeeds.

The validation procedure uses your validation profile and sample data to run the following validations tasks:

1. Create a model in your account using the model package's inference image and the optional model artifacts that are stored in Amazon S3.

   **Note**
   A model package is specific to the region in which you create it. The S3 bucket where the model artifacts are stored must be in the same region where your created the model package.

2. Create a transform job in your account using the model to verify that your inference image works with SageMaker.

3. Create a validation profile.

   **Note**
   In your validation profile, provide only data that you want to expose publicly.

Validation can take up to a few hours. To see the status of the jobs in your account, in the SageMaker console, see the Transform jobs pages. If validation fails, you can access the scan and validation reports from the SageMaker console. After fixing issues, recreate the algorithm. When the status of the algorithm is COMPLETED, find it in the SageMaker console and start the listing process

   **Note**
   To publish your model package on AWS Marketplace, at least one validation profile is required.

You can create an model package either by using the SageMaker console or by using the SageMaker API.

**Topics**

- Create a Model Package Resource (Console) (p. 1598)
- Create a Model Package Resource (API) (p. 1599)

**Create a Model Package Resource (Console)**

**To create a model package in the SageMaker console:**

2. Choose **Model packages**, then choose **Create model package**.
3. On the **Inference specifications** page, provide the following information:
   a. For **Model package name**, type a name for your model package. The model package name must be unique in your account and in the AWS region. The name must have 1 to 64 characters. Valid characters are a-z, A-Z, 0-9, and - (hyphen).
   b. Type a description for your model package. This description appears in the SageMaker console and in the AWS Marketplace.
   c. For **Inference specification options**, choose **Provide the location of the inference image and model artifacts** to create a model package by using an inference container and model artifacts.
Choose **Provide the algorithm used for training and its model artifacts** to create a model package from an algorithm resource that you created or subscribe to from AWS Marketplace.

d. If you chose **Provide the location of the inference image and model artifacts** for **Inference specification options**, provide the following information for **Container definition** and **Supported resources**:

   i. For **Location of inference image**, type the path to the image that contains your inference code. The image must be stored as a Docker container in Amazon ECR.

   ii. For **Location of model data artifacts**, type the location in S3 where your model artifacts are stored.

   iii. For **Container DNS host name**, type the name of the DNS host to use for your container.

   iv. For **Supported instance types for real-time inference**, choose the instance types that your model package supports for real-time inference from SageMaker hosted endpoints.

   v. For **Supported instance types for batch transform jobs**, choose the instance types that your model package supports for batch transform jobs.

   vi. **Supported content types**, type the content types that your model package expects for inference requests.

   vii. For **Supported response MIME types**, type the MIME types that your model package uses to provide inferences.

e. If you chose **Provide the algorithm used for training and its model artifacts** for **Inference specification options**, provide the following information:

   i. For **Algorithm ARN**, type the Amazon Resource Name (ARN) of the algorithm resource to use to create the model package.

   ii. For **Location of model data artifacts**, type the location in S3 where your model artifacts are stored.

e. Choose **Next**.

4. On the **Validation and scanning** page, provide the following information:

   a. For **Publish this model package on AWS Marketplace**, choose **Yes** to publish the model package on AWS Marketplace.

   b. For **Validate this model package**, choose **Yes** if you want SageMaker to run batch transform jobs that you specify to test the inference code of your model package.

      **Note**
      To publish your model package on AWS Marketplace, your model package must be validated.

   c. For **IAM role**, choose an IAM role that has the required permissions to run batch transform jobs in SageMaker, or choose **Create a new role** to allow SageMaker to create a role that has the AmazonSageMakerFullAccess managed policy attached. For information, see SageMaker Roles (p. 1647).

   d. For **Validation profile**, specify the following:

      • A name for the validation profile.

      • A **Transform job definition**. This is a JSON block that describes a batch transform job. This is in the same format as the TransformJobDefinition input parameter of the CreateAlgorithm API.

5. Choose **Create model package**.

**Create a Model Package Resource (API)**

To create a model package by using the SageMaker API, call the CreateModelPackage API.
Use Algorithm and Model Package Resources

You can create algorithms and model packages as resources in your Amazon SageMaker account, and you can find and subscribe to algorithms and model packages on AWS Marketplace.

Use algorithms to:

- Run training jobs. For information, see Use an Algorithm to Run a Training Job (p. 1601).
- Run hyperparameter tuning jobs. For information, see Use an Algorithm to Run a Hyperparameter Tuning Job (p. 1603).
- Create model packages. After you use an algorithm resource to run a training job or a hyperparameter tuning job, you can use the model artifacts that these jobs output along with the algorithm to create a model package. For information, see Create a Model Package Resource (p. 1597).

**Note**
If you subscribe to an algorithm on AWS Marketplace, you must create a model package before you can use it to get inferences by creating hosted endpoint or running a batch transform job.

Use model packages to:

- Create models that you can use to get real-time inference or run batch transform jobs. For information, see Use a Model Package to Create a Model (p. 1606).
• Create hosted endpoints to get real-time inference. For information, see Step 6.1: Deploy the Model to SageMaker Hosting Services (p. 68).
• Create batch transform jobs. For information, see Step 6.2: Deploy the Model with Batch Transform (p. 70).

Topics
• Use an Algorithm to Run a Training Job (p. 1601)
• Use an Algorithm to Run a Hyperparameter Tuning Job (p. 1603)
• Use a Model Package to Create a Model (p. 1606)

Use an Algorithm to Run a Training Job

You can create use an algorithm resource to create a training job by using the Amazon SageMaker console, the low-level Amazon SageMaker API, or the Amazon SageMaker Python SDK.

Topics
• Use an Algorithm to Run a Training Job (Console) (p. 1601)
• Use an Algorithm to Run a Training Job (API) (p. 1602)
• Use an Algorithm to Run a Training Job (Amazon SageMaker Python SDK) (p. 1602)

Use an Algorithm to Run a Training Job (Console)

To use an algorithm to run a training job (console)

2. Choose Algorithms.
3. Choose an algorithm that you created from the list on the My algorithms tab or choose an algorithm that you subscribed to on the AWS Marketplace subscriptions tab.
4. Choose Create training job.
   The algorithm you chose will automatically be selected.
5. On the Create training job page, provide the following information:
   a. For Job name, type a name for the training job.
   b. For IAM role, choose an IAM role that has the required permissions to run training jobs in SageMaker, or choose Create a new role to allow SageMaker to create a role that has the AmazonSageMakerFullAccess managed policy attached. For information, see SageMaker Roles (p. 1647).
   c. For Resource configuration, provide the following information:
      i. For Instance type, choose the instance type to use for training.
      ii. For Instance count, type the number of ML instances to use for the training job.
      iii. For Additional volume per instance (GB), type the size of the ML storage volume that you want to provision. ML storage volumes store model artifacts and incremental states.
      iv. For Encryption key, if you want Amazon SageMaker to use an AWS Key Management Service key to encrypt data in the ML storage volume attached to the training instance, specify the key.
      v. For Stopping condition, specify the maximum amount of time in seconds, minutes, hours, or days, that you want the training job to run.
d. For **VPC**, choose a Amazon VPC that you want to allow your training container to access. For more information, see Give SageMaker Training Jobs Access to Resources in Your Amazon VPC (p. 1694).

e. For **Hyperparameters**, specify the values of the hyperparameters to use for the training job.

f. For **Input data configuration**, specify the following values for each channel of input data to use for the training job. You can see what channels the algorithm you're using for training sports, and the content type, supported compression type, and supported input modes for each channel, under **Channel specification** section of the **Algorithm summary** page for the algorithm.

i. For **Channel name**, type the name of the input channel.
ii. For **Content type**, type the content type of the data that the algorithm expects for the channel.
iii. For **Compression type**, choose the data compression type to use, if any.
iv. For **Record wrapper**, choose RecordIO if the algorithm expects data in the RecordIO format.

v. For **S3 data type**, **S3 data distribution type**, and **S3 location**, specify the appropriate values. For information about what these values mean, see S3DataSource.

vi. For **Input mode**, choose File to download the data from to the provisioned ML storage volume, and mount the directory to a Docker volume. Choose PipeTo stream data directly from Amazon S3 to the container.

vii. To add another input channel, choose **Add channel**. If you are finished adding input channels, choose **Done**.

g. For **Output location**, specify the following values:

i. For **S3 output path**, choose the S3 location where the training job stores output, such as model artifacts.

   **Note**
   You use the model artifacts stored at this location to create a model or model package from this training job.

ii. For **Encryption key**, if you want SageMaker to use a AWS KMS key to encrypt output data at rest in the S3 location.

h. For **Tags**, specify one or more tags to manage the training job. Each tag consists of a key and an optional value. Tag keys must be unique per resource.

i. Choose **Create training job** to run the training job.

### Use an Algorithm to Run a Training Job (API)

To use an algorithm to run a training job by using the SageMaker API, specify either the name or the Amazon Resource Name (ARN) as the **AlgorithmName** field of the **AlgorithmSpecification** object that you pass to **CreateTrainingJob**. For information about training models in SageMaker, see Train a Model with Amazon SageMaker (p. 8).

### Use an Algorithm to Run a Training Job (Amazon SageMaker Python SDK)

Use an algorithm that you created or subscribed to on AWS Marketplace to create a training job, create an **AlgorithmEstimator** object and specify either the Amazon Resource Name (ARN) or the name of the algorithm as the value of the **algorithm_arn** argument. Then call the **fit** method of the estimator. For example:

```python
from sagemaker import AlgorithmEstimator
data_path = os.path.join(DATA_DIR, 'marketplace', 'training')
```
Use an Algorithm to Run a Hyperparameter Tuning Job

A hyperparameter tuning job finds the best version of a model by running many training jobs on your dataset using the algorithm and ranges of hyperparameters that you specify. It then chooses the hyperparameter values that result in a model that performs the best, as measured by a metric that you choose. For more information, see Perform Automatic Model Tuning (p. 1023).

You can use an algorithm resource to create a hyperparameter tuning job by using the Amazon SageMaker console, the low-level Amazon SageMaker API, or the Amazon SageMaker Python SDK.

Topics
• Use an Algorithm to Run a Hyperparameter Tuning Job (Console) (p. 1603)
• Use an Algorithm to Run a Hyperparameter Tuning Job (API) (p. 1605)
• Use an Algorithm to Run a Hyperparameter Tuning Job (Amazon SageMaker Python SDK) (p. 1605)

Use an Algorithm to Run a Hyperparameter Tuning Job (Console)

To use an algorithm to run a hyperparameter tuning job (console)

2. Choose Algorithms.
3. Choose an algorithm that you created from the list on the My algorithms tab or choose an algorithm that you subscribed to on the AWS Marketplace subscriptions tab.
4. Choose Create hyperparameter tuning job.
   The algorithm you chose will automatically be selected.
5. On the Create hyperparameter tuning job page, provide the following information:
   a. For Warm start, choose Enable warm start to use the information from previous hyperparameter tuning jobs as a starting point for this hyperparameter tuning job. For more information, see Run a Warm Start Hyperparameter Tuning Job (p. 1041).
      i. Choose Identical data and algorithm if your input data is the same as the input data for the parent jobs of this hyperparameter tuning job, or choose Transfer learning to use additional or different input data for this hyperparameter tuning job.
      ii. For Parent hyperparameter tuning job(s), choose up to 5 hyperparameter tuning jobs to use as parents to this hyperparameter tuning job.
   b. For Hyperparameter tuning job name, type a name for the tuning job.
   c. For IAM role, choose an IAM role that has the required permissions to run hyperparameter tuning jobs in SageMaker, or choose Create a new role to allow SageMaker to create a role that has the AmazonSageMakerFullAccess managed policy attached. For information, see SageMaker Roles (p. 1647).
d. For **VPC**, choose a Amazon VPC that you want to allow the training jobs that the tuning job launches to access. For more information, see *Give SageMaker Training Jobs Access to Resources in Your Amazon VPC* (p. 1694).

e. Choose **Next**.

f. For **Objective metric**, choose the metric that the hyperparameter tuning job uses to determine the best combination of hyperparameters, and choose whether to minimize or maximize this metric. For more information, see *View the Best Training Job* (p. 1039).

g. For **Hyperparameter configuration**, choose ranges for the tunable hyperparameters that you want the tuning job to search, and set static values for hyperparameters that you want to remain constant in all training jobs that the hyperparameter tuning job launches. For more information, see *Define Hyperparameter Ranges* (p. 1026).

h. Choose **Next**.

i. For **Input data configuration**, specify the following values for each channel of input data to use for the hyperparameter tuning job. You can see what channels the algorithm you're using for hyperparameter tuning supports, and the content type, supported compression type, and supported input modes for each channel, under **Channel specification** section of the **Algorithm summary** page for the algorithm.

i. For **Channel name**, type the name of the input channel.

ii. For **Content type**, type the content type of the data that the algorithm expects for the channel.

iii. For **Compression type**, choose the data compression type to use, if any.

iv. For **Record wrapper**, choose RecordIO if the algorithm expects data in the RecordIO format.

v. For **S3 data type**, **S3 data distribution type**, and **S3 location**, specify the appropriate values. For information about what these values mean, see *S3DataSource*.

vi. For **Input mode**, choose **File** to download the data from to the provisioned ML storage volume, and mount the directory to a Docker volume. Choose **Pipe** to stream data directly from Amazon S3 to the container.

vii. To add another input channel, choose **Add channel**. If you are finished adding input channels, choose **Done**.

j. For **Output location**, specify the following values:

i. For **S3 output path**, choose the S3 location where the training jobs that this hyperparameter tuning job launches store output, such as model artifacts.

   **Note**
   You use the model artifacts stored at this location to create a model or model package from this hyperparameter tuning job.

ii. For **Encryption key**, if you want SageMaker to use a AWS KMS key to encrypt output data at rest in the S3 location.

k. For **Resource configuration**, provide the following information:

i. For **Instance type**, choose the instance type to use for each training job that the hyperparameter tuning job launches.

ii. For **Instance count**, type the number of ML instances to use for each training job that the hyperparameter tuning job launches.

iii. For **Additional volume per instance (GB)**, type the size of the ML storage volume that you want to provision each training job that the hyperparameter tuning job launches. ML storage volumes store model artifacts and incremental states.

iv. For **Encryption key**, if you want Amazon SageMaker to use an AWS Key Management Service key to encrypt data in the ML storage volume attached to the training instances, specify the key.
l. For **Resource limits**, provide the following information:

   i. For **Maximum training jobs**, specify the maximum number of training jobs that you want the hyperparameter tuning job to launch. A hyperparameter tuning job can launch a maximum of 500 training jobs.

   ii. For **Maximum parallel training jobs**, specify the maximum number of concurrent training jobs that the hyperparameter tuning job can launch. A hyperparameter tuning job can launch a maximum of 10 concurrent training jobs.

   iii. For **Stopping condition**, specify the maximum amount of time in seconds, minutes, hours, or days, that you want each training job that the hyperparameter tuning job launches to run.

m. For **Tags**, specify one or more tags to manage the hyperparameter tuning job. Each tag consists of a key and an optional value. Tag keys must be unique per resource.

n. Choose **Create jobs** to run the hyperparameter tuning job.

### Use an Algorithm to Run a Hyperparameter Tuning Job (API)

To use an algorithm to run a hyperparameter tuning job by using the SageMaker API, specify either the name or the Amazon Resource Name (ARN) of the algorithm as the `AlgorithmName` field of the `AlgorithmSpecification` object that you pass to `CreateHyperParameterTuningJob`. For information about hyperparameter tuning in SageMaker, see Perform Automatic Model Tuning (p. 1023).

### Use an Algorithm to Run a Hyperparameter Tuning Job (Amazon SageMaker Python SDK)

Use an algorithm that you created or subscribed to on AWS Marketplace to create a hyperparameter tuning job, create an `AlgorithmEstimator` object and specify either the Amazon Resource Name (ARN) or the name of the algorithm as the value of the `algorithm_arn` argument. Then initialize a `HyperparameterTuner` object with the `AlgorithmEstimator` you created as the value of the `estimator` argument. Finally, call the `fit` method of the `AlgorithmEstimator`. For example:

```python
from sagemaker import AlgorithmEstimator
from sagemaker.tuner import HyperparameterTuner
data_path = os.path.join(DATA_DIR, 'marketplace', 'training')
algo = AlgorithmEstimator(
    role='SageMakerRole',
    train_instance_count=1,
    train_instance_type='ml.c4.xlarge',
    sagemaker_session=sagemaker_session,
    base_job_name='test-marketplace')
train_input = algo.sagemaker_session.upload_data(
    path=data_path, key_prefix='integ-test-data/marketplace/train')
algo.set_hyperparameters(max_leaf_nodes=10)
tuner = HyperparameterTuner(estimator=algo, base_tuning_job_name='some-name',
                            objective_metric_name='validation:accuracy',
                            hyperparameter_ranges=hyperparameter_ranges,
                            max_jobs=2, max_parallel_jobs=2)
tuner.fit({'training': train_input}, include_cls_metadata=False)
tuner.wait()
```
Use a Model Package to Create a Model

Use a model package to create a deployable model that you can use to get real-time inferences by creating a hosted endpoint or to run batch transform jobs. You can create a deployable model from a model package by using the Amazon SageMaker console, the low-level SageMaker API, or the Amazon SageMaker Python SDK.

Topics
- Use a Model Package to Create a Model (Console) (p. 1606)
- Use a Model Package to Create a Model (API) (p. 1606)
- Use a Model Package to Create a Model (Amazon SageMaker Python SDK) (p. 1607)

Use a Model Package to Create a Model (Console)

To create a deployable model from a model package (console)

2. Choose Model packages.
3. Choose a model package that you created from the list on the My model packages tab or choose a model package that you subscribed to on the AWS Marketplace subscriptions tab.
4. Choose Create model.
5. For Model name, type a name for the model.
6. For IAM role, choose an IAM role that has the required permissions to call other services on your behalf, or choose Create a new role to allow SageMaker to create a role that has the AmazonSageMakerFullAccess managed policy attached. For information, see SageMaker Roles (p. 1647).
7. For VPC, choose a Amazon VPC that you want to allow the model to access. For more information, see Give SageMaker Hosted Endpoints Access to Resources in Your Amazon VPC (p. 1697).
8. Leave the default values for Container input options and Choose model package.
9. For environment variables, provide the names and values of environment variables you want to pass to the model container.
10. For Tags, specify one or more tags to manage the model. Each tag consists of a key and an optional value. Tag keys must be unique per resource.
11. Choose Create model.

After you create a deployable model, you can use it to set up an endpoint for real-time inference or create a batch transform job to get inferences on entire datasets. For information about hosted endpoints in SageMaker, see Step 6.1: Deploy the Model to SageMaker Hosting Services (p. 68). For information about batch transform jobs, see Step 6.2: Deploy the Model with Batch Transform (p. 70).

Use a Model Package to Create a Model (API)

To use a model package to create a deployable model by using the SageMaker API, specify the name or the Amazon Resource Name (ARN) of the model package as the ModelPackageName field of the ContainerDefinition object that you pass to the CreateModel API.

After you create a deployable model, you can use it to set up an endpoint for real-time inference or create a batch transform job to get inferences on entire datasets. For information about hosted endpoints in SageMaker, see Step 6.1: Deploy the Model to SageMaker Hosting Services (p. 68). For information about batch transform jobs, see Step 6.2: Deploy the Model with Batch Transform (p. 70).
Use a Model Package to Create a Model (Amazon SageMaker Python SDK)

To use a model package to create a deployable model by using the SageMaker Python SDK, initialize a `ModelPackage` object, and pass the Amazon Resource Name (ARN) of the model package as the `model_package_arn` argument. For example:

```python
from sagemaker import ModelPackage
model = ModelPackage(role='SageMakerRole',
                    model_package_arn='training-job-scikit-decision-trees-1542660466-6f92',
                    sagemaker_session=sagemaker_session)
```

After you create a deployable model, you can use it to set up an endpoint for real-time inference or create a batch transform job to get inferences on entire datasets. For information about hosted endpoints in SageMaker, see Step 6.1: Deploy the Model to SageMaker Hosting Services (p. 68). For information about batch transform jobs, see Step 6.2: Deploy the Model with Batch Transform (p. 70).

Sell Amazon SageMaker Algorithms and Model Packages

Selling Amazon SageMaker algorithms and model packages is a three-step process:

1. Develop your algorithm or model, and package it in a Docker container. For information, see Develop Algorithms and Models in Amazon SageMaker (p. 1608).
2. Create an algorithm or model package resource in SageMaker. For information, see Create Algorithm and Model Package Resources (p. 1594).
3. Register as a seller on AWS Marketplace and list your algorithm or model package on AWS Marketplace. For information about registering as a seller, see Getting Started as a Seller in the User Guide for AWS Marketplace Providers. For information about listing and monetizing your algorithms and model packages, see Listing Algorithms and Model Packages in AWS Marketplace for Machine Learning in the User Guide for AWS Marketplace Providers.
Develop Algorithms and Models in Amazon SageMaker

Before you can create algorithm and model package resources to use in Amazon SageMaker or list on AWS Marketplace, you have to develop them and package them in Docker containers.

**Note**
When algorithms and model packages are created for listing on AWS Marketplace, SageMaker scans the containers for security vulnerabilities on supported operating systems. Only the following operating system versions are supported:

- Debian: 6.0, 7, 8, 9, 10
- CentOS: 5, 6, 7
- Oracle Linux: 5, 6, 7
- Alpine: 3.3, 3.4, 3.5
- Amazon Linux

Develop Algorithms in SageMaker

An algorithm should be packaged as a docker container and stored in Amazon ECR to use it in SageMaker. The Docker container contains the training code used to run training jobs and, optionally, the inference code used to get inferences from models trained by using the algorithm.

For information about developing algorithms in SageMaker and packaging them as containers, see Using Docker containers with SageMaker (p. 1356). For a complete example of how to create an algorithm container, see the sample notebook at https://github.com/awslabs/amazon-sagemaker-examples/blob/master/advanced_functionality/scikitbring_your_own/scikitbring_your_own.ipynb. You can also find the sample notebook in a SageMaker notebook instance. The notebook is in the Advanced Functionality section, and is named scikitbring_your_own.ipynb. For information about using the sample notebooks in a notebook instance, see Example Notebooks (p. 134).

Always thoroughly test your algorithms before you create algorithm resources to publish on AWS Marketplace.

**Note**
When a buyer subscribes to your containerized product, the Docker containers run in an isolated (internet-free) environment. When you create your containers, do not rely on making outgoing calls over the internet. Calls to AWS services are also not allowed.
Develop Models in SageMaker

A deployable model in SageMaker consists of inference code, model artifacts, an IAM role that is used to access resources, and other information required to deploy the model in SageMaker. Model artifacts are the results of training a model by using a machine learning algorithm. The inference code must be packaged in a Docker container and stored in Amazon ECR. You can either package the model artifacts in the same container as the inference code, or store them in Amazon S3.

You create a model by running a training job in SageMaker, or by training a machine learning algorithm outside of SageMaker. If you run a training job in SageMaker, the resulting model artifacts are available in the ModelArtifacts field in the response to a call to the DescribeTrainingJob operation. For information about how to develop a SageMaker model container, see Use Your Own Inference Code (p. 1387). For a complete example of how to create a model container from a model trained outside of SageMaker, see the sample notebook at https://github.com/awslabs/amazon-sagemaker-examples/blob/master/advanced_functionality/xgboostbring_your_own_model/xgboostbring_your_own_model.ipynb. You can also find the sample notebook in a SageMaker notebook instance. The notebook is in the Advanced Functionality section, and is named xgboostbring_your_own_model.ipynb. For information about using the sample notebooks in a notebook instance, see Example Notebooks (p. 134).

Always thoroughly test your models before you create model packages to publish on AWS Marketplace.

Note
When a buyer subscribes to your containerized product, the Docker containers run in an isolated (internet-free) environment. When you create your containers, do not rely on making outgoing calls over the internet. Calls to AWS services are also not allowed.

List Your Algorithm or Model Package on AWS Marketplace

After creating and validating your algorithm or model in Amazon SageMaker, list your product on AWS Marketplace. The listing process makes your products available in the AWS Marketplace and the SageMaker console.

To list products on AWS Marketplace, you must be a registered seller. To register, use the self-registration process from the AWS Marketplace Management Portal (AMMP). For information, see Getting Started as a Seller in the User Guide for AWS Marketplace Providers. When you start the product listing process from the Amazon SageMaker console, we check your seller registration status. If you have not registered, we direct you to do so.

To start the listing process, do one of the following:

- From the SageMaker console, choose the product, choose Actions, and choose Publish new ML Marketplace listing. This carries over your product reference, the Amazon Resource Name (ARN), and directs you to the AMMP to create the listing.
- Go to ML listing process, manually enter the Amazon Resource Name (ARN), and start your product listing. This process carries over the product metadata that you entered when creating the product in SageMaker. For an algorithm listing, the information includes the supported instance types and hyperparameters. In addition, you can enter a product description, promotional information, and support information as you would with other AWS Marketplace products.
Find and Subscribe to Algorithms and Model Packages on AWS Marketplace

With AWS Marketplace, you can browse and search for hundreds of machine learning algorithms and models in a broad range of categories, such as computer vision, natural language processing, speech recognition, text, data, voice, image, video analysis, fraud detection, predictive analysis, and more.

To find algorithms on AWS Marketplace

1. Open the Amazon SageMaker console at https://console.aws.amazon.com/sagemaker/.
2. Choose Algorithms, then choose Find algorithms.

This takes you to the AWS Marketplace algorithms page. For information about finding and subscribing to algorithms on AWS Marketplace, see Machine Learning Products in the AWS Marketplace User Guide for AWS Consumers.

To find model packages on AWS Marketplace

2. Choose Model packages, then choose Find model packages.

This takes you to the AWS Marketplace model packages page. For information about finding and subscribing to model packages on AWS Marketplace, see Machine Learning Products in the AWS Marketplace User Guide for AWS Consumers.

Use Algorithms and Model Packages

For information about using algorithms and model packages that you subscribe to in SageMaker, see Use Algorithm and Model Package Resources (p. 1600).

Note

When you create a training job, inference endpoint, and batch transform job from an algorithm or model package that you subscribe to on AWS Marketplace, the training and inference containers do not have access to the internet. Because the containers do not have access to the internet, the seller of the algorithm or model package does not have access to your data.
Security in Amazon SageMaker

Cloud security at AWS is the highest priority. As an AWS customer, you benefit from a data center and network architecture that is built to meet the requirements of the most security-sensitive organizations.

Security is a shared responsibility between AWS and you. The shared responsibility model describes this as security of the cloud and security in the cloud:

- **Security of the cloud** – AWS is responsible for protecting the infrastructure that runs AWS services in the AWS Cloud. AWS also provides you with services that you can use securely. Third-party auditors regularly test and verify the effectiveness of our security as part of the AWS compliance programs. To learn about the compliance programs that apply to Amazon SageMaker, see AWS Services in Scope by Compliance Program.
- **Security in the cloud** – Your responsibility is determined by the AWS service that you use. You are also responsible for other factors including the sensitivity of your data, your company's requirements, and applicable laws and regulations.

This documentation helps you understand how to apply the shared responsibility model when using SageMaker. The following topics show you how to configure SageMaker to meet your security and compliance objectives. You also learn how to use other AWS services that help you to monitor and secure your SageMaker resources.

Topics
- Access Control (p. 1611)
- Data Protection in Amazon SageMaker (p. 1613)
- Identity and Access Management for Amazon SageMaker (p. 1618)
- Logging and Monitoring (p. 1677)
- Compliance Validation for Amazon SageMaker (p. 1677)
- Resilience in Amazon SageMaker (p. 1678)
- Infrastructure Security in Amazon SageMaker (p. 1678)

Access Control

Amazon SageMaker Studio notebooks and SageMaker notebook instances differ in their runtime environments. The following topics describe how to control root access to these notebooks.

Topics
- Access control and SageMaker Studio notebooks (p. 1611)
- Control root access to a SageMaker notebook instance (p. 1613)

Access control and SageMaker Studio notebooks

Amazon SageMaker Studio uses filesystem and container permissions for access control and isolation of Studio users and notebooks. This is one of the major differences between Studio notebooks and SageMaker notebook instances. This topic describes how permissions are set up to avoid security threats, what SageMaker does by default, and how the customer can customize the permissions. For more
information about Studio notebooks and their runtime environment, see Use Amazon SageMaker Studio Notebooks (p. 84).

SageMaker app permissions

A run-as user is a POSIX user/group which is used to run the JupyterServer app and KernelGateway apps inside the container.

The run-as user for the JupyterServer app is sagemaker-user (1000) by default. This user has sudo permissions to enable the installation of dependencies such as yum packages.

The run-as user for the KernelGateway apps is root (0) by default. This user is able to install dependencies using pip/apt-get/conda.

Due to the previously mentioned user remapping, neither user is able to access resources or make changes to the host instance.

User remapping

SageMaker performs user-remapping to map a user inside the container to a user on the host instance outside the container. The range of user IDs (0 - 65535) in the container are mapped to non-privileged user IDs above 65535 on the instance. For example, sagemaker-user (1000) inside the container might map to user (200001) on the instance, where the number in parentheses is the user ID. If the customer creates a new user/group inside the container, it won't be privileged on the host instance regardless of the user/group ID. The root user of the container is also mapped to a non-privileged user on the instance. For more information, see Isolate containers with a user namespace.

Custom image permissions

Customers can bring their own custom SageMaker images. These images can specify a different run-as user/group to launch the KernelGateway app. The customer can implement fine grained permission control inside the image, for example, to disable root access or perform other actions. The same user remapping applies here. For more information, see Bring your own SageMaker image (p. 102).

Container isolation

Docker keeps a list of default capabilities that the container can use. SageMaker doesn’t add additional capabilities. SageMaker adds specific route rules to block requests to Amazon EFS and the instance metadata service (IMDS) from the container. Customers can’t change these route rules from the container. For more information, see Runtime privilege and Linux capabilities.

App metadata access

Metadata used by running apps are mounted to the container with read-only permission. Customers aren’t able to modify this metadata from the container. For the available metadata, see Get Notebook and App Metadata (p. 92).

User isolation on EFS

When you onboard to Studio, SageMaker creates an Amazon Elastic File System (EFS) volume for your domain that is shared by all Studio users in the domain. Each user gets their own private home directory on the EFS volume. This home directory is used to store the user's notebooks, Git repositories, and other data. To prevent other users in the domain from accessing the user's data, SageMaker creates a globally unique user ID for the user's profile and applies it as a POSIX user/group ID for the user's home directory.

EBS access

An Amazon Elastic Block Store (Amazon EBS) volume is attached to the host instance and shared across all images. It's used for the root volume of the notebooks and stores temporary data that's generated.
inside the container. The storage isn't persisted when the instance running the notebooks is deleted. The root user inside the container can't access the EBS volume.

**Control root access to a SageMaker notebook instance**

By default, when you create a notebook instance, users that log into that notebook instance have root access. Data science is an iterative process that might require the data scientist to test and use different software tools and packages, so many notebook instance users need to have root access to be able to install these tools and packages. Because users with root access have administrator privileges, users can access and edit all files on a notebook instance with root access enabled.

If you don't want users to have root access to a notebook instance, when you call `CreateNotebookInstance` or `UpdateNotebookInstance` operations, set the `RootAccess` field to `Disabled`. You can also disable root access for users when you create or update a notebook instance in the Amazon SageMaker console. For information, see Step 2: Create an Amazon SageMaker Notebook Instance (p. 60).

**Note**
Lifecycles configurations need root access to be able to set up a notebook instance. Because of this, lifecycles configurations associated with a notebook instance always run with root access even if you disable root access for users.

**Note**
For security reasons, Rootless Docker is installed on root-disabled notebook instances instead of regular Docker. For more information, see Run the Docker daemon as a non-root user (Rootless mode).

**Data Protection in Amazon SageMaker**

The AWS shared responsibility model applies to data protection in Amazon SageMaker. As described in this model, AWS is responsible for protecting the global infrastructure that runs all of the AWS Cloud. You are responsible for maintaining control over your content that is hosted on this infrastructure. This content includes the security configuration and management tasks for the AWS services that you use. For more information about data privacy, see the Data Privacy FAQ. For information about data protection in Europe, see the AWS Shared Responsibility Model and GDPR blog post on the AWS Security Blog.

For data protection purposes, we recommend that you protect AWS account credentials and set up individual user accounts with AWS Identity and Access Management (IAM). That way each user is given only the permissions necessary to fulfill their job duties. We also recommend that you secure your data in the following ways:

- Use multi-factor authentication (MFA) with each account.
- Use SSL/TLS to communicate with AWS resources. We recommend TLS 1.2 or later.
- Set up API and user activity logging with AWS CloudTrail.
- Use AWS encryption solutions, along with all default security controls within AWS services.
- Use advanced managed security services such as Amazon Macie, which assists in discovering and securing personal data that is stored in Amazon S3.
- If you require FIPS 140-2 validated cryptographic modules when accessing AWS through a command line interface or an API, use a FIPS endpoint. For more information about the available FIPS endpoints, see Federal Information Processing Standard (FIPS) 140-2.

We strongly recommend that you never put sensitive identifying information, such as your customers’ account numbers, into free-form fields such as a **Name** field. This includes when you work with Amazon
SageMaker or other AWS services using the console, API, AWS CLI, or AWS SDKs. Any data that you enter into Amazon SageMaker or other services might get picked up for inclusion in diagnostic logs. When you provide a URL to an external server, don't include credentials information in the URL to validate your request to that server.

Topics
- Protect Data at Rest Using Encryption (p. 1614)
- Protecting Data in Transit with Encryption (p. 1615)
- Key Management (p. 1617)
- Internetwork Traffic Privacy (p. 1617)

Protect Data at Rest Using Encryption

To protect your Amazon SageMaker Studio notebooks and SageMaker notebook instances, along with your model-building data and model artifacts, SageMaker uses the AWS Key Management Service (AWS KMS) to encrypt the notebooks and data. SageMaker uses AWS managed customer master keys (CMKs) by default. For more control, you can specify your own customer managed CMKs. For more information on AWS KMS, see What is AWS Key Management Service?.

Topics
- Studio notebooks (p. 1614)
- Notebook instances and SageMaker jobs (p. 1614)

Studio notebooks

In Amazon SageMaker Studio, your SageMaker Studio notebooks and data can be stored in the following locations:

- An S3 bucket – When you onboard to Studio and enable shareable notebook resources, SageMaker shares notebook snapshots and metadata in an Amazon Simple Storage Service (Amazon S3) bucket.
- An EFS volume – When you onboard to Studio, SageMaker attaches an Amazon Elastic File System (Amazon EFS) volume to your domain for storing your Studio notebooks and data files. The EFS volume persists after the domain is deleted.
- An EBS volume – When you open a notebook in Studio, an Amazon Elastic Block Store (Amazon EBS) is attached to the instance that the notebook runs on. The EBS volume persists for the duration of the instance.

SageMaker uses the AWS Key Management Service (AWS KMS) to encrypt the S3 bucket and both volumes with an AWS managed customer master key (CMK) by default. For more control, you can specify your own customer managed CMK when you onboard to Studio or through the SageMaker API. For more information, see Onboard to Amazon SageMaker Studio (p. 34) and CreateDomain.

In the CreateDomain API, you use the S3KmsKeyId parameter to specify the custom CMK for shareable notebooks. You use the KmsKeyId parameter to specify the custom CMK for the EFS and EBS volumes. The same CMK is used for both volumes. The custom CMK for shareable notebooks can be the same CMK as used for the volumes or a different CMK.

Notebook instances and SageMaker jobs

To encrypt the machine learning (ML) storage volume that is attached to notebooks, processing jobs, training jobs, hyperparameter tuning jobs, batch transform jobs, and endpoints, you can pass a AWS KMS key to SageMaker. If you don't specify a KMS key, SageMaker encrypts storage volumes with a transient
key and discards it immediately after encrypting the storage volume. For notebook instances, if you don’t specify a KMS key, SageMaker encrypts both OS volumes and ML data volumes with a system-managed KMS key.

You can use an AWS managed KMS key to encrypt all instance OS volumes. You can encrypt all ML data volumes for all SageMaker instances with a KMS key that you specify. ML storage volumes are mounted as follows:

- **Notebooks** - `/home/ec2-user/SageMaker`
- **Processing** - `/opt/ml/processing` and `/tmp/`
- **Training** - `/opt/ml/` and `/tmp/`
- **Batch** - `/opt/ml/` and `/tmp/`
- **Endpoints** - `/opt/ml/` and `/tmp/`

Processing, batch transform, and training job containers and their storage are ephemeral in nature. When the job completes, output is uploaded to Amazon S3 using AWS KMS encryption (with an optional KMS key you specify) and the instance is torn down.

**Important**
Sensitive data that needs to be encrypted with a KMS key for compliance reasons should be stored in the ML storage volume or in Amazon S3, both of which can be encrypted using a KMS key you specify.

When you open a notebook instance, SageMaker saves it and any files associated with it in the SageMaker folder in the ML storage volume by default. When you stop a notebook instance, SageMaker creates a snapshot of the ML storage volume. Any customizations to the operating system of the stopped instance, such as installed custom libraries or operating system level settings, are lost. Consider using a lifecycle configuration to automate customizations of the default notebook instance. When you terminate an instance, the snapshot and the ML storage volume are deleted. Any data that you need to persist beyond the lifespan of the notebook instance should be transferred to an Amazon S3 bucket.

**Note**
Certain Nitro-based SageMaker instances include local storage, depending on the instance type. Local storage volumes are encrypted using a hardware module on the instance. You can’t use a KMS key on an instance type with local storage. For a list of instance types that support local instance storage, see [Instance Store Volumes](#). For more information about storage volumes on Nitro-based instances, see [Amazon EBS and NVMe on Linux Instances](#). For more information about local instance storage encryption, see [SSD Instance Store Volumes](#).

### Protecting Data in Transit with Encryption

All inter-network data in transit supports TLS 1.2 encryption.

Amazon SageMaker ensures that machine learning (ML) model artifacts and other system artifacts are encrypted in transit and at rest. Requests to the SageMaker API and console are made over a secure (SSL) connection. You pass AWS Identity and Access Management roles to SageMaker to provide permissions to access resources on your behalf for training and deployment. You can use encrypted Amazon S3 buckets for model artifacts and data, as well as pass a AWS KMS key to SageMaker instances to encrypt the attached ML storage volumes.

Some intra-network data in-transit (inside the service platform) is unencrypted. This includes:

- Command and control communications between the service control plane and training job instances (not customer data).
- Communications between nodes in distributed processing jobs (intra-network).
- Communications between nodes in distributed training jobs (intra-network).
There are no inter-node communications for batch processing.

You can choose to encrypt communication between nodes in a training cluster. For information about how to do this, see Protect Communications Between ML Compute Instances in a Distributed Training Job (p. 1616). Enabling inter-container traffic encryption can increase training time, especially if you are using distributed deep learning algorithms. For affected algorithms, adding this additional level of security also increases cost. The training time for most SageMaker built-in algorithms, such as XGBoost, DeepAR, and linear learner, typically aren't affected.

FIPS validated endpoints are available for the SageMaker API and request router for hosted models (runtime). For information about FIPS compliant endpoints, see Federal Information Processing Standard (FIPS) 140-2.

**Protect Communications Between ML Compute Instances in a Distributed Training Job**

By default, Amazon SageMaker runs training jobs in an Amazon Virtual Private Cloud (Amazon VPC) to help keep your data secure. You can add another level of security to protect your training containers and data by configuring a private VPC. Distributed ML frameworks and algorithms usually transmit information that is directly related to the model such as weights, not the training dataset. When performing distributed training, you can further protect data that is transmitted between instances. This can help you to comply with regulatory requirements. To do this, use inter-container traffic encryption.

Enabling inter-container traffic encryption can increase training time, especially if you are using distributed deep learning algorithms. Enabling inter-container traffic encryption doesn't affect training jobs with a single compute instance. However, for training jobs with several compute instances, the effect on training time depends on the amount of communication between compute instances. For affected algorithms, adding this additional level of security also increases cost. The training time for most SageMaker built-in algorithms, such as XGBoost, DeepAR, and linear learner, typically aren't affected.

You can enable inter-container traffic encryption for training jobs or hyperparameter tuning jobs. You can use SageMaker APIs or console to enable inter-container traffic encryption.

For information about running training jobs in a private VPC, see Give SageMaker Training Jobs Access to Resources in Your Amazon VPC (p. 1694).

**Enable Inter-Container Traffic Encryption (API)**

Before enabling inter-container traffic encryption on training or hyperparameter tuning jobs with APIs, you need to add inbound and outbound rules to your private VPC's security group.

**To enable inter-container traffic encryption (API)**

1. Add the following inbound and outbound rules in the security group for your private VPC:

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Port Range</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>UDP</td>
<td>500</td>
<td>Self Security Group ID</td>
</tr>
<tr>
<td>50</td>
<td>N/A</td>
<td>Self Security Group ID</td>
</tr>
</tbody>
</table>

2. When you send a request to the CreateTrainingJob or CreateHyperParameterTuningJob API, specify True for the EnableInterContainerTrafficEncryption parameter.

**Note**

The AWS Security Group Console might show display ports range as "All", however EC2 ignores the specified port range because it is not applicable for the ESP 50 IP protocol.
Enable Inter-Container Traffic Encryption (Console)

Enable Inter-container Traffic Encryption in a Training Job

To enable inter-container traffic encryption in a training job
1. Open the Amazon SageMaker console at https://console.aws.amazon.com/sagemaker/.
2. In the navigation pane, choose Training, then choose Training jobs.
3. Choose Create training job.
4. Under Network, choose a VPC. You can use the default VPC or one that you have created.
5. Choose Enable inter-container traffic encryption.

After you enable inter-container traffic encryption, finish creating the training job. For more information, see Step 5: Train a Model (p. 64).

Enable Inter-container Traffic Encryption in a Hyperparameter Tuning Job

To enable inter-container traffic encryption in a hyperparameter tuning job
1. Open the Amazon SageMaker console at https://console.aws.amazon.com/sagemaker/.
2. In the navigation pane, choose Training, then choose Hyperparameter tuning jobs.
3. Choose Create hyperparameter tuning job.
4. Under Network, choose a VPC. You can use the default VPC or one that you created.
5. Choose Enable inter-container traffic encryption.

After enabling inter-container traffic encryption, finish creating the hyperparameter tuning job. For more information, see Configure and Launch a Hyperparameter Tuning Job (p. 1033).

Key Management

Customers can specify AWS KMS keys, including bring your own keys (BYOK), to use for envelope encryption with Amazon S3 input/output buckets and machine learning (ML) Amazon EBS volumes. ML volumes for notebook instances and for processing, training, and hosted model Docker containers can be optionally encrypted by using AWS KMS customer-owned keys. All instance OS volumes are encrypted with an AWS-managed AWS KMS key.

**Note**
Certain Nitro-based instances include local storage, dependent on the instance type. Local storage volumes are encrypted using a hardware module on the instance. You can't request a VolumeKmsKeyId when using an instance type with local storage.
For a list of instance types that support local instance storage, see Instance Store Volumes.
For more information about local instance storage encryption, see SSD Instance Store Volumes.
For more information about storage volumes on nitro-based instances, see Amazon EBS and NVMe on Linux Instances.

For information about AWS KMS keys see What is AWS Key Management Service? in the AWS Key Management Service Developer Guide.

Internetwork Traffic Privacy

This topic describes how Amazon SageMaker secures connections from the service to other locations.

Internetwork communications support TLS 1.2 encryption between all components and clients.
Instances can be connected to Customer VPC, providing access to S3 VPC endpoints or customer repositories. Internet egress can be managed through this interface by the customer if service platform internet egress is disabled for notebooks. For training and hosting, egress through the service platform is not available when connected to the customer’s VPC.

By default, API calls made to published endpoints traverse the public network to the request router. SageMaker supports Amazon Virtual Private Cloud interface endpoints powered by AWS PrivateLink for private connectivity between the customer’s VPC and the request router to access hosted model endpoints. For information about Amazon VPC, see Connect to SageMaker Through a VPC Interface Endpoint (p. 1683)

Identity and Access Management for Amazon SageMaker

AWS Identity and Access Management (IAM) is an AWS service that helps an administrator securely control access to AWS resources. IAM administrators control who can be authenticated (signed in) and authorized (have permissions) to use SageMaker resources. IAM is an AWS service that you can use with no additional charge.

**Topics**
- Audience (p. 1618)
- Authenticating with Identities (p. 1619)
- Managing Access Using Policies (p. 1620)
- How Amazon SageMaker Works with IAM (p. 1622)
- Amazon SageMaker Identity-Based Policy Examples (p. 1625)
- SageMaker Roles (p. 1647)
- AWS Managed (Predefined) Policies for Amazon SageMaker (p. 1663)
- Amazon SageMaker API Permissions: Actions, Permissions, and Resources Reference (p. 1663)
- Troubleshooting Amazon SageMaker Identity and Access (p. 1675)

**Audience**

How you use AWS Identity and Access Management (IAM) differs, depending on the work that you do in SageMaker.

**Service user** – If you use the SageMaker service to do your job, then your administrator provides you with the credentials and permissions that you need. As you use more SageMaker features to do your work, you might need additional permissions. Understanding how access is managed can help you request the right permissions from your administrator. If you cannot access a feature in SageMaker, see Troubleshooting Amazon SageMaker Identity and Access (p. 1675).

**Service administrator** – If you're in charge of SageMaker resources at your company, you probably have full access to SageMaker. It's your job to determine which SageMaker features and resources your employees should access. You must then submit requests to your IAM administrator to change the permissions of your service users. Review the information on this page to understand the basic concepts of IAM. To learn more about how your company can use IAM with SageMaker, see How Amazon SageMaker Works with IAM (p. 1622).

**IAM administrator** – If you're an IAM administrator, you might want to learn details about how you can write policies to manage access to SageMaker. To view example SageMaker identity-based policies that you can use in IAM, see Amazon SageMaker Identity-Based Policy Examples (p. 1625).
Authenticating with Identities

Authentication is how you sign in to AWS using your identity credentials. For more information about signing in using the AWS Management Console, see Signing in to the AWS Management Console as an IAM user or root user in the IAM User Guide.

You must be authenticated (signed in to AWS) as the AWS account root user, an IAM user, or by assuming an IAM role. You can also use your company’s single sign-on authentication or even sign in using Google or Facebook. In these cases, your administrator previously set up identity federation using IAM roles. When you access AWS using credentials from another company, you are assuming a role indirectly.

To sign in directly to the AWS Management Console, use your password with your root user email address or your IAM user name. You can access AWS programmatically using your root user or IAM users access keys. AWS provides SDK and command line tools to cryptographically sign your request using your credentials. If you don’t use AWS tools, you must sign the request yourself. Do this using Signature Version 4, a protocol for authenticating inbound API requests. For more information about authenticating requests, see Signature Version 4 signing process in the AWS General Reference.

Regardless of the authentication method that you use, you might also be required to provide additional security information. For example, AWS recommends that you use multi-factor authentication (MFA) to increase the security of your account. To learn more, see Using multi-factor authentication (MFA) in AWS in the IAM User Guide.

AWS account root user

When you first create an AWS account, you begin with a single sign-in identity that has complete access to all AWS services and resources in the account. This identity is called the AWS account root user and is accessed by signing in with the email address and password that you used to create the account. We strongly recommend that you do not use the root user for your everyday tasks, even the administrative ones. Instead, adhere to the best practice of using the root user only to create your first IAM user. Then securely lock away the root user credentials and use them to perform only a few account and service management tasks.

IAM Users and Groups

An IAM user is an identity within your AWS account that has specific permissions for a single person or application. An IAM user can have long-term credentials such as a user name and password or a set of access keys. To learn how to generate access keys, see Managing access keys for IAM users in the IAM User Guide. When you generate access keys for an IAM user, make sure you view and securely save the key pair. You cannot recover the secret access key in the future. Instead, you must generate a new access key pair.

An IAM group is an identity that specifies a collection of IAM users. You can’t sign in as a group. You can use groups to specify permissions for multiple users at a time. Groups make permissions easier to manage for large sets of users. For example, you could have a group named IAMAdmins and give that group permissions to administer IAM resources.

Users are different from roles. A user is uniquely associated with one person or application, but a role is intended to be assumable by anyone who needs it. Users have permanent long-term credentials, but roles provide temporary credentials. To learn more, see When to create an IAM user (instead of a role) in the IAM User Guide.

IAM Roles

An IAM role is an identity within your AWS account that has specific permissions. It is similar to an IAM user, but is not associated with a specific person. You can temporarily assume an IAM role in the AWS Management Console by switching roles. You can assume a role by calling an AWS CLI or AWS API
operation or by using a custom URL. For more information about methods for using roles, see Using IAM roles in the IAM User Guide.

IAM roles with temporary credentials are useful in the following situations:

- **Temporary IAM user permissions** – An IAM user can assume an IAM role to temporarily take on different permissions for a specific task.

- **Federated user access** – Instead of creating an IAM user, you can use existing identities from AWS Directory Service, your enterprise user directory, or a web identity provider. These are known as federated users. AWS assigns a role to a federated user when access is requested through an identity provider. For more information about federated users, see Federated users and roles in the IAM User Guide.

- **Cross-account access** – You can use an IAM role to allow someone (a trusted principal) in a different account to access resources in your account. Roles are the primary way to grant cross-account access. However, with some AWS services, you can attach a policy directly to a resource (instead of using a role as a proxy). To learn the difference between roles and resource-based policies for cross-account access, see How IAM roles differ from resource-based policies in the IAM User Guide.

- **Cross-service access** – Some AWS services use features in other AWS services. For example, when you make a call in a service, it's common for that service to run applications in Amazon EC2 or store objects in Amazon S3. A service might do this using the calling principal's permissions, using a service role, or using a service-linked role.

- **Principal permissions** – When you use an IAM user or role to perform actions in AWS, you are considered a principal. Policies grant permissions to a principal. When you use some services, you might perform an action that then triggers another action in a different service. In this case, you must have permissions to perform both actions. To see whether an action requires additional dependent actions in a policy, see Actions, Resources, and Condition Keys for Amazon SageMaker in the Service Authorization Reference.

- **Service role** – A service role is an IAM role that a service assumes to perform actions on your behalf. Service roles provide access only within your account and cannot be used to grant access to services in other accounts. An IAM administrator can create, modify, and delete a service role from within IAM. For more information, see Creating a role to delegate permissions to an AWS service in the IAM User Guide.

- **Service-linked role** – A service-linked role is a type of service role that is linked to an AWS service. The service can assume the role to perform an action on your behalf. Service-linked roles appear in your IAM account and are owned by the service. An IAM administrator can view, but not edit the permissions for service-linked roles.

- **Applications running on Amazon EC2** – You can use an IAM role to manage temporary credentials for applications that are running on an EC2 instance and making AWS CLI or AWS API requests. This is preferable to storing access keys within the EC2 instance. To assign an AWS role to an EC2 instance and make it available to all of its applications, you create an instance profile that is attached to the instance. An instance profile contains the role and enables programs that are running on the EC2 instance to get temporary credentials. For more information, see Using an IAM role to grant permissions to applications running on Amazon EC2 instances in the IAM User Guide.

To learn whether to use IAM roles or IAM users, see When to create an IAM role (instead of a user) in the IAM User Guide.

## Managing Access Using Policies

You control access in AWS by creating policies and attaching them to IAM identities or AWS resources. A policy is an object in AWS that, when associated with an identity or resource, defines its permissions. You can sign in as the root user or an IAM user, or you can assume an IAM role. When you then make a request, AWS evaluates the related identity-based or resource-based policies. Permissions in the policies determine whether the request is allowed or denied. Most policies are stored in AWS as JSON
documents. For more information about the structure and contents of JSON policy documents, see Overview of JSON policies in the IAM User Guide.

Administrators can use AWS JSON policies to specify who has access to what. That is, which principal can perform actions on what resources, and under what conditions.

Every IAM entity (user or role) starts with no permissions. In other words, by default, users can do nothing, not even change their own password. To give a user permission to do something, an administrator must attach a permissions policy to a user. Or the administrator can add the user to a group that has the intended permissions. When an administrator gives permissions to a group, all users in that group are granted those permissions.

IAM policies define permissions for an action regardless of the method that you use to perform the operation. For example, suppose that you have a policy that allows the iam:GetRole action. A user with that policy can get role information from the AWS Management Console, the AWS CLI, or the AWS API.

Identity-Based Policies

Identity-based policies are JSON permissions policy documents that you can attach to an identity, such as an IAM user, group of users, or role. These policies control what actions users and roles can perform, on which resources, and under what conditions. To learn how to create an identity-based policy, see Creating IAM policies in the IAM User Guide.

Identity-based policies can be further categorized as inline policies or managed policies. Inline policies are embedded directly into a single user, group, or role. Managed policies are standalone policies that you can attach to multiple users, groups, and roles in your AWS account. Managed policies include AWS managed policies and customer managed policies. To learn how to choose between a managed policy or an inline policy, see Choosing between managed policies and inline policies in the IAM User Guide.

Resource-Based Policies

Resource-based policies are JSON policy documents that you attach to a resource. Examples of resource-based policies are IAM role trust policies and Amazon S3 bucket policies. In services that support resource-based policies, service administrators can use them to control access to a specific resource. For the resource where the policy is attached, the policy defines what actions a specified principal can perform on that resource and under what conditions. You must specify a principal in a resource-based policy. Principals can include accounts, users, roles, federated users, or AWS services.

Resource-based policies are inline policies that are located in that service. You can’t use AWS managed policies from IAM in a resource-based policy.

Access Control Lists (ACLs)

Access control lists (ACLs) control which principals (account members, users, or roles) have permissions to access a resource. ACLs are similar to resource-based policies, although they do not use the JSON policy document format.

Amazon S3, AWS WAF, and Amazon VPC are examples of services that support ACLs. To learn more about ACLs, see Access control list (ACL) overview in the Amazon Simple Storage Service Developer Guide.

Other Policy Types

AWS supports additional, less-common policy types. These policy types can set the maximum permissions granted to you by the more common policy types.

- Permissions boundaries – A permissions boundary is an advanced feature in which you set the maximum permissions that an identity-based policy can grant to an IAM entity (IAM user or role). You can set a permissions boundary for an entity. The resulting permissions are the intersection of
entity's identity-based policies and its permissions boundaries. Resource-based policies that specify the user or role in the Principal field are not limited by the permissions boundary. An explicit deny in any of these policies overrides the allow. For more information about permissions boundaries, see Permissions boundaries for IAM entities in the IAM User Guide.

- **Service control policies (SCPs)** – SCPs are JSON policies that specify the maximum permissions for an organization or organizational unit (OU) in AWS Organizations. AWS Organizations is a service for grouping and centrally managing multiple AWS accounts that your business owns. If you enable all features in an organization, then you can apply service control policies (SCPs) to any or all of your accounts. The SCP limits permissions for entities in member accounts, including each AWS account root user. For more information about Organizations and SCPs, see How SCPs work in the AWS Organizations User Guide.

- **Session policies** – Session policies are advanced policies that you pass as a parameter when you programmatically create a temporary session for a role or federated user. The resulting session's permissions are the intersection of the user or role's identity-based policies and the session policies. Permissions can also come from a resource-based policy. An explicit deny in any of these policies overrides the allow. For more information, see Session policies in the IAM User Guide.

### Multiple Policy Types

When multiple types of policies apply to a request, the resulting permissions are more complicated to understand. To learn how AWS determines whether to allow a request when multiple policy types are involved, see Policy evaluation logic in the IAM User Guide.

### How Amazon SageMaker Works with IAM

Before you use IAM to manage access to SageMaker, you should understand what IAM features are available to use with SageMaker. To get a high-level view of how SageMaker and other AWS services work with IAM, see AWS Services That Work with IAM in the IAM User Guide.

#### Topics

- SageMaker Identity-Based Policies (p. 1622)

### SageMaker Identity-Based Policies

With IAM identity-based policies, you can specify allowed or denied actions and resources as well as the conditions under which actions are allowed or denied. SageMaker supports specific actions, resources, and condition keys. To learn about all of the elements that you use in a JSON policy, see IAM JSON Policy Elements Reference in the IAM User Guide.

#### Actions

Administrators can use AWS JSON policies to specify who has access to what. That is, which principal can perform actions on what resources, and under what conditions.

The Action element of a JSON policy describes the actions that you can use to allow or deny access in a policy. Policy actions usually have the same name as the associated AWS API operation. There are some exceptions, such as permission-only actions that don't have a matching API operation. There are also some operations that require multiple actions in a policy. These additional actions are called dependent actions.

Include actions in a policy to grant permissions to perform the associated operation.

Policy actions in SageMaker use the following prefix before the action: sagemaker:. For example, to grant someone permission to run an SageMaker training job with the SageMaker CreateTrainingJob
API operation, you include the sagemaker:CreateTrainingJob action in their policy. Policy statements must include either an Action or NotAction element. SageMaker defines its own set of actions that describe tasks that you can perform with this service.

To specify multiple actions in a single statement, separate them with commas as follows:

```
"Action": [
    "sagemaker:action1",
    "sagemaker:action2"
]
```

You can specify multiple actions using wildcards (*). For example, to specify all actions that begin with the word Describe, include the following action:

```
"Action": "sagemaker:Describe*"
```

To see a list of SageMaker actions, see Actions, Resources, and Condition Keys for Amazon SageMaker in the IAM User Guide.

Resources

SageMaker does not support specifying resource ARNs in a policy.

Condition Keys

Administrators can use AWS JSON policies to specify who has access to what. That is, which principal can perform actions on what resources, and under what conditions.

The Condition element (or Condition block) lets you specify conditions in which a statement is in effect. The Condition element is optional. You can create conditional expressions that use condition operators, such as equals or less than, to match the condition in the policy with values in the request.

If you specify multiple Condition elements in a statement, or multiple keys in a single Condition element, AWS evaluates them using a logical AND operation. If you specify multiple values for a single condition key, AWS evaluates the condition using a logical OR operation. All of the conditions must be met before the statement's permissions are granted.

You can also use placeholder variables when you specify conditions. For example, you can grant an IAM user permission to access a resource only if it is tagged with their IAM user name. For more information, see IAM policy elements: variables and tags in the IAM User Guide.

AWS supports global condition keys and service-specific condition keys. To see all AWS global condition keys, see AWS global condition context keys in the IAM User Guide.

SageMaker defines its own set of condition keys and also supports using some global condition keys. To see all AWS global condition keys, see AWS Global Condition Context Keys in the IAM User Guide.

SageMaker supports a number of service-specific condition keys that you can use for fine-grained access control for the following operations:

- CreateProcessingJob
- CreateTrainingJob
- CreateModel
• CreateEndpointConfig
• CreateTransformJob
• CreateHyperParameterTuningJob
• CreateLabelingJob
• CreateNotebookInstance
• UpdateNotebookInstance

To see a list of SageMaker condition keys, see Condition Keys for Amazon SageMaker in the IAM User Guide. To learn with which actions and resources you can use a condition key, see Actions Defined by Amazon SageMaker.

For examples of using SageMaker condition keys, see the following: Control Creation of SageMaker Resources with Condition Keys (p. 1635).

Examples

To view examples of SageMaker identity-based policies, see Amazon SageMaker Identity-Based Policy Examples (p. 1625).

SageMaker Resource-Based Policies

SageMaker does not support resource-based policies.

Authorization Based on SageMaker Tags

You can attach tags to SageMaker resources or pass tags in a request to SageMaker. To control access based on tags, you provide tag information in the condition element of a policy using the sagemaker:ResourceTag/key-name, aws:RequestTag/key-name, or aws:TagKeys condition keys. For more information about tagging SageMaker resources, see Control Access to SageMaker Resources by Using Tags (p. 1644).

To view an example identity-based policy for limiting access to a resource based on the tags on that resource, see Control Access to SageMaker Resources by Using Tags (p. 1644).

SageMaker IAM Roles

An IAM role is an entity within your AWS account that has specific permissions.

Using Temporary Credentials with SageMaker

You can use temporary credentials to sign in with federation, assume an IAM role, or to assume a cross-account role. You obtain temporary security credentials by calling AWS STS API operations such as AssumeRole or GetFederationToken.

SageMaker supports using temporary credentials.

Service-Linked Roles

SageMaker doesn’t support service-linked roles.

Service Roles

This feature allows a service to assume a service role on your behalf. This role allows the service to access resources in other services to complete an action on your behalf. Service roles appear in your
IAM account and are owned by the account. This means that an IAM administrator can change the permissions for this role. However, doing so might break the functionality of the service.

SageMaker supports service roles.

**Choosing an IAM Role in SageMaker**

When you create a notebook instance, processing job, training job, hosted endpoint, or batch transform job resource in SageMaker, you must choose a role to allow SageMaker to access SageMaker on your behalf. If you have previously created a service role or service-linked role, then SageMaker provides you with a list of roles to choose from. It's important to choose a role that allows access to the AWS operations and resources you need. For more information, see SageMaker Roles (p. 1647).

**Amazon SageMaker Identity-Based Policy Examples**

By default, IAM users and roles don't have permission to create or modify SageMaker resources. They also can't perform tasks using the AWS Management Console, AWS CLI, or AWS API. An IAM administrator must create IAM policies that grant users and roles permission to perform specific API operations on the specified resources they need. The administrator must then attach those policies to the IAM users or groups that require those permissions. To learn how to attach policies to an IAM user or group, see Adding and Removing IAM Identity Permissions in the IAM User Guide.

To learn how to create an IAM identity-based policy using these example JSON policy documents, see Creating Policies on the JSON Tab in the IAM User Guide.

**Topics**

- Policy Best Practices (p. 1625)
- Using the SageMaker Console (p. 1626)
- Allow Users to View Their Own Permissions (p. 1634)
- Control Creation of SageMaker Resources with Condition Keys (p. 1635)
- Control Access to the SageMaker API by Using Identity-based Policies (p. 1642)
- Limit Access to SageMaker API and Runtime Calls by IP Address (p. 1643)
- Limit Access to a Notebook Instance by IP Address (p. 1644)
- Control Access to SageMaker Resources by Using Tags (p. 1644)
- Require the Presence or Absence of Tags for API Calls (p. 1646)
- Use Tags with Hyperparameter Tuning Jobs (p. 1647)

**Policy Best Practices**

Identity-based policies are very powerful. They determine whether someone can create, access, or delete SageMaker resources in your account. These actions can incur costs for your AWS account. When you create or edit identity-based policies, follow these guidelines and recommendations:

- **Get started using AWS managed policies** – To start using SageMaker quickly, use AWS managed policies to give your employees the permissions they need. These policies are already available in your account and are maintained and updated by AWS. For more information, see Get started using permissions with AWS managed policies in the IAM User Guide.

- **Grant least privilege** – When you create custom policies, grant only the permissions required to perform a task. Start with a minimum set of permissions and grant additional permissions as necessary. Doing so is more secure than starting with permissions that are too lenient and then trying to tighten them later. For more information, see Grant least privilege in the IAM User Guide.
• **Enable MFA for sensitive operations** – For extra security, require IAM users to use multi-factor authentication (MFA) to access sensitive resources or API operations. For more information, see Using multi-factor authentication (MFA) in AWS in the IAM User Guide.

• **Use policy conditions for extra security** – To the extent that it's practical, define the conditions under which your identity-based policies allow access to a resource. For example, you can write conditions to specify a range of allowable IP addresses that a request must come from. You can also write conditions to allow requests only within a specified date or time range, or to require the use of SSL or MFA. For more information, see IAM JSON policy elements: Condition in the IAM User Guide.

Using the SageMaker Console

To access the Amazon SageMaker console, you must have a minimum set of permissions. These permissions must allow you to list and view details about the SageMaker resources in your AWS account. If you create an identity-based policy that is more restrictive than the minimum required permissions, the console won't function as intended for entities (IAM users or roles) with that policy.

To ensure that those entities can still use the SageMaker console, also attach the following AWS managed policy to the entities. For more information, see Adding Permissions to a User in the IAM User Guide:

You don't need to allow minimum console permissions for users that are making calls only to the AWS CLI or the AWS API. Instead, allow access to only the actions that match the API operation that you're trying to perform.

**Topics**

• Permissions Required to Use the Amazon SageMaker Console (p. 1626)

• Permissions Required to Use the Amazon SageMaker Ground Truth Console (p. 1628)

• Permissions Required to Use the Amazon Augmented AI (Preview) Console (p. 1629)

Permissions Required to Use the Amazon SageMaker Console

The permissions reference table lists the Amazon SageMaker API operations and shows the required permissions for each operation. For more information about Amazon SageMaker API operations, see Amazon SageMaker API Permissions: Actions, Permissions, and Resources Reference (p. 1663).

To use the Amazon SageMaker console, you need to grant permissions for additional actions. Specifically, the console needs permissions that allow the ec2 actions to display subnets, VPCs, and security groups. Optionally, the console needs permission to create execution roles for tasks such as CreateNotebook, CreateTrainingJob, and CreateModel. Grant these permissions with the following permissions policy:

```json
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "SageMakerApis",
            "Effect": "Allow",
            "Action": [
                "sagemaker:*"
            ],
            "Resource": "*"
        },
        {
            "Sid": "VpcConfigurationForCreateForms",
            "Effect": "Allow",
            "Action": [
                "ec2:*"
            ],
            "Resource": "*"
        }
    ]
}
```
"Action": [  "ec2:DescribeVpcs",  "ec2:DescribeSubnets",  "ec2:DescribeSecurityGroups"
],
"Resource": "*"
},
{
  "Sid": "KmsKeysForCreateForms",
  "Effect": "Allow",
  "Action": [  "kms:DescribeKey",  "kms:ListAliases"
],
  "Resource": "*"
},
{
  "Sid": "AccessAwsMarketplaceSubscritions",
  "Effect": "Allow",
  "Action": [  "aws-marketplace:ViewSubscriptions"
],
  "Resource": "*"
},
{
  "Effect": "Allow",
],
  "Resource": "*"
},
{
  "Sid": "ListAndCreateExecutionRoles",
  "Effect": "Allow",
  "Action": [  "iam:ListRoles",  "iam:CreateRole",  "iam:CreatePolicy",  "iam:AttachRolePolicy"
],
  "Resource": "*"
},
{
  "Sid": "DescribeECRMetaData",
  "Effect": "Allow",
  "Action": [  "ecr:Describe*"
],
  "Resource": "*"
},
{
  "Sid": "PassRoleForExecutionRoles",
  "Effect": "Allow",
  "Action": [  "iam:PassRole"
],
  "Resource": "*",
  "Condition": {
    "StringEquals": {

}````
Permissions Required to Use the Amazon SageMaker Ground Truth Console

To use the Amazon SageMaker Ground Truth console, you need to grant permissions for additional resources. Specifically, the console needs permissions for the AWS Marketplace to view subscriptions, Amazon Cognito operations to manage your private workforce, Amazon S3 actions for access to your input and output files, and AWS Lambda actions to list and invoke functions. Grant these permissions with the following permissions policy:

```json
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Sid": "GroundTruthConsole",
         "Effect": "Allow",
         "Action": [
            "aws-marketplace:DescribeListings",
            "aws-marketplace:ViewSubscriptions",
            "cognito-idp:AdminAddUserToGroup",
            "cognito-idp:AdminCreateUser",
            "cognito-idp:AdminDeleteUser",
            "cognito-idp:AdminDisableUser",
            "cognito-idp:AdminEnableUser",
            "cognito-idp:AdminRemoveUserFromGroup",
            "cognito-idp:CreateGroup",
            "cognito-idp:CreateUserPool",
            "cognito-idp:CreateUserPoolClient",
            "cognito-idp:CreateUserPoolDomain",
            "cognito-idp:DescribeUserPool",
            "cognito-idp:DeleteUserPool",
            "cognito-idp:DeleteUserPoolClient",
            "cognito-idp:ListGroups",
            "cognito-idp:ListIdentityProviders",
            "cognito-idp:ListUsers",
            "cognito-idp:ListUsersInGroup",
            "cognito-idp:ListUserPoolClients",
            "cognito-idp:ListUserPools",
            "cognito-idp:ListUsers",
            "cognito-idp:ListUserPoolClient",
            "cognito-idp:ListUserPoolClients",
            "cognito-idp:ListUserPools",
            "cognito-idp:UpdateUserPool",
            "cognito-idp:UpdateUserPoolClient",
            "groundtruthlabeling:DescribeConsoleJob",
            "groundtruthlabeling:ListDatasetObjects",
            "groundtruthlabeling:RunFilterOrSampleManifestJob",
            "groundtruthlabeling:RunGenerateManifestByCrawlingJob",
            "lambda:InvokeFunction",
            "lambda:ListFunctions",
            "s3:GetObject",
            "s3:PutObject",
            "s3:SelectObjectContent"
         ],
         "Resource": "*"
      }
   ]
}
```
Permissions Required to Use the Amazon Augmented AI (Preview) Console

To use the Augmented AI console, you need to grant permissions for additional resources. Grant these permissions with the following permissions policy:

```json
{
  "Version": "2012-10-17",
  "Statement": [ 
    {
      "Effect": "Allow",
      "Action": [ 
        "sagemaker:*Algorithm",
        "sagemaker:*Algorithms",
        "sagemaker:*App",
        "sagemaker:*Apps",
        "sagemaker:*AutoMLJob",
        "sagemaker:*AutoMLJobs",
        "sagemaker:*CodeRepositories",
        "sagemaker:*CodeRepository",
        "sagemaker:*CompilationJob",
        "sagemaker:*CompilationJobs",
        "sagemaker:*Endpoint",
        "sagemaker:*EndpointConfig",
        "sagemaker:*EndpointConfigs",
        "sagemaker:*EndpointWeightsAndCapacities",
        "sagemaker:*Endpoints",
        "sagemaker:*Environment",
        "sagemaker:*EnvironmentVersion",
        "sagemaker:*EnvironmentVersions",
        "sagemaker:*Environments",
        "sagemaker:*Experiment",
        "sagemaker:*Experiments",
        "sagemaker:*FlowDefinitions",
        "sagemaker:*HumanLoop",
        "sagemaker:*HumanLoops",
        "sagemaker:*HumanTaskUi",
        "sagemaker:*HumanTaskUis",
        "sagemaker:*HyperParameterTuningJob",
        "sagemaker:*HyperParameterTuningJobs",
        "sagemaker:*LabelingJob",
        "sagemaker:*LabelingJobs",
        "sagemaker:*Metrics",
        "sagemaker:*Model",
        "sagemaker:*ModelPackage",
        "sagemaker:*ModelPackages",
        "sagemaker:*ModelTemplate",
        "sagemaker:*ModelTemplates",
        "sagemaker:*MonitoringExecutions",
        "sagemaker:*MonitoringSchedule",
        "sagemaker:*MonitoringSchedules",
        "sagemaker:*NotebookInstance",
        "sagemaker:*NotebookInstanceLifecycleConfig",
        "sagemaker:*NotebookInstanceLifecycleConfigs",
        "sagemaker:*NotebookInstanceUrl",
        "sagemaker:*NotebookInstances",
        "sagemaker:*ProcessingJob",
        "sagemaker:*ProcessingJobs",
        "sagemaker:*RenderUiTemplate",
        "sagemaker:*Search",
        "sagemaker:*SearchSuggestions",
        "sagemaker:*Tags",
        "sagemaker:*TrainingJob",
        "sagemaker:*TrainingJobs",
        "sagemaker:*TransformJob",
        "sagemaker:*TransformJobs",
        "sagemaker:*Trial",
```
"sagemaker:*TrialComponent",
  "sagemaker:*TrialComponents",
  "sagemaker:*Trials",
  "sagemaker:*Workteam",
  "sagemaker:*Workteams"
],
  "Resource": "*
},
},
  
  "Effect": "Allow",
  "Action": [
    "sagemaker:*FlowDefinition"
  ],
  "Resource": "*
",  
  "Condition": {
    "StringEqualsIfExists": {
      "sagemaker:WorkteamType": [
        "private-crowd",
        "vendor-crowd"
      ]
    }
  }
},
},

  "Effect": "Allow",
  "Action": [
    "application-autoscaling:DeleteScalingPolicy",
    "application-autoscaling:DeleteScheduledAction",
    "application-autoscaling:DeregisterScalableTarget",
    "application-autoscaling:DescribeScalableTargets",
    "application-autoscaling:DescribeScalingActivities",
    "application-autoscaling:DescribeScalingPolicies",
    "application-autoscaling:DescribeScheduledActions",
    "application-autoscaling:PutScalingPolicy",
    "application-autoscaling:PutScheduledAction",
    "application-autoscaling:RegisterScalableTarget",
    "aws-marketplace:ViewSubscriptions",
    "cloudwatch:DeleteAlarms",
    "cloudwatch:DescribeAlarms",
    "cloudwatch:GetMetricData",
    "cloudwatch:GetMetricStatistics",
    "cloudwatch:ListMetrics",
    "cloudwatch:PutMetricAlarm",
    "cloudwatch:PutMetricData",
    "codecommit:BatchGetRepositories",
    "codecommit:CreateRepository",
    "codecommit:GetRepository",
    "codecommit:GetBranches",
    "codecommit:GetRepositories",
    "cognito-idp:AdminAddUserToGroup",
    "cognito-idp:AdminCreateUser",
    "cognito-idp:AdminDeleteUser",
    "cognito-idp:AdminDisableUser",
    "cognito-idp:AdminEnableUser",
    "cognito-idp:AdminRemoveUserFromGroup",
    "cognito-idp:CreateGroup",
    "cognito-idp:CreateUserPool",
    "cognito-idp:CreateUserPoolClient",
    "cognito-idp:DescribeUserPool",
    "cognito-idp:DescribeUserPoolClient",
    "cognito-idp:ListGroups",
    "cognito-idp:ListIdentityProviders",
    "cognito-idp:ListUserPoolClients",
    "cognito-idp:ListUserPools",
    "cognito-idp:ListUsers",
    "cognito-idp:ListUserPools"}
"cognito-idp:ListUsersInGroup",
"cognito-idp:UpdateUserPool",
"cognito-idp:UpdateUserPoolClient",
"ec2:CreateNetworkInterface",
"ec2:CreateNetworkInterfacePermission",
"ec2:CreateVpcEndpoint",
"ec2:DeleteNetworkInterface",
"ec2:DeleteNetworkInterfacePermission",
"ec2:DescribeDhcpOptions",
"ec2:DescribeNetworkInterfaces",
"ec2:DescribeRouteTables",
"ec2:DescribeSecurityGroups",
"ec2:DescribeSubnets",
"ec2:DescribeVpcEndpoints",
"ec2:DescribeVpcs",
"ecr:BatchCheckLayerAvailability",
"ecr:BatchGetImage",
"ecr:CreateRepository",
"ecr:Describe*",
"ecr:GetAuthorizationToken",
"ecr:GetDownloadUrlForLayer",
"elastic-inference:Connect",
"elasticfilesystem:DescribeFileSystems",
"elasticfilesystem:DescribeMountTargets",
"fsx:DescribeFileSystems",
"glue:CreateJob",
"glue:DeleteJob",
"glue:GetJob",
"glue:GetJobRun",
"glue:GetJobRuns",
"glue:GetJobs",
"glue:ResetJobBookmark",
"glue:StartJobRun",
"glue:UpdateJob",
"groundtruthlabeling:*",
"iam:ListRoles",
"kms:DescribeKey",
"kms:ListAliases",
"lambda:ListFunctions",
"logs:CreateLogGroup",
"logs:CreateLogStream",
"logs:DescribeLogGroups",
"logs:DescribeLogStreams",
"logs:GetLogEvents",
"logs:PutLogEvents",
"sns:ListTopics"
],
"Resource": "*
},
{
"Effect": "Allow",
"Action": [
"logs:CreateLogDelivery",
"logs:DeleteLogDelivery",
"logs:DescribeResourcePolicies",
"logs:GetLogDelivery",
"logs:ListLogDeliveries",
"logs:PutLogResourcePolicy",
"logs:UpdateLogDelivery"
],
"Resource": "*
},
{
"Effect": "Allow",
"Action": [
"ecr:SetRepositoryPolicy",
"ecr:BatchCheckLayerAvailability",
"ecr:BatchGetImage",
"ecr:CreateRepository",
"ecr:Describe*",
"ecr:GetAuthorizationToken",
"ecr:GetDownloadUrlForLayer",
"elastic-inference:Connect",
"elasticfilesystem:DescribeFileSystems",
"elasticfilesystem:DescribeMountTargets",
"fsx:DescribeFileSystems",
"glue:CreateJob",
"glue:DeleteJob",
"glue:GetJob",
"glue:GetJobRun",
"glue:GetJobRuns",
"glue:GetJobs",
"glue:ResetJobBookmark",
"glue:StartJobRun",
"glue:UpdateJob",
"groundtruthlabeling:*",
"iam:ListRoles",
"kms:DescribeKey",
"kms:ListAliases",
"lambda:ListFunctions",
"logs:CreateLogGroup",
"logs:CreateLogStream",
"logs:DescribeLogGroups",
"logs:DescribeLogStreams",
"logs:GetLogEvents",
"logs:PutLogEvents",
"sns:ListTopics"
],
"Resource": "*
},
{
"Effect": "Allow",
"Action": [
"logs:CreateLogDelivery",
"logs:DeleteLogDelivery",
"logs:DescribeResourcePolicies",
"logs:GetLogDelivery",
"logs:ListLogDeliveries",
"logs:PutLogResourcePolicy",
"logs:UpdateLogDelivery"
],
"Resource": "*
},
{
"Effect": "Allow",
"Action": [
"ecr:SetRepositoryPolicy",
"ecr:BatchCheckLayerAvailability",
"ecr:BatchGetImage",
"ecr:CreateRepository",
"ecr:Describe*",
"ecr:GetAuthorizationToken",
"ecr:GetDownloadUrlForLayer",
"elastic-inference:Connect",
"elasticfilesystem:DescribeFileSystems",
"elasticfilesystem:DescribeMountTargets",
"fsx:DescribeFileSystems",
"glue:CreateJob",
"glue:DeleteJob",
"glue:GetJob",
"glue:GetJobRun",
"glue:GetJobRuns",
"glue:GetJobs",
"glue:ResetJobBookmark",
"glue:StartJobRun",
"glue:UpdateJob",
"groundtruthlabeling:*",
"iam:ListRoles",
"kms:DescribeKey",
"kms:ListAliases",
"lambda:ListFunctions",
"logs:CreateLogGroup",
"logs:CreateLogStream",
"logs:DescribeLogGroups",
"logs:DescribeLogStreams",
"logs:GetLogEvents",
"logs:PutLogEvents",
"sns:ListTopics"
],
"Resource": "*
},
{
"Effect": "Allow",
"Action": [
"logs:CreateLogDelivery",
"logs:DeleteLogDelivery",
"logs:DescribeResourcePolicies",
"logs:GetLogDelivery",
"logs:ListLogDeliveries",
"logs:PutLogResourcePolicy",
"logs:UpdateLogDelivery"
],
"Resource": "*
},
{
"Effect": "Allow",
"Action": [
"ecr:SetRepositoryPolicy",
"ecr:BatchCheckLayerAvailability",
"ecr:BatchGetImage",
"ecr:CreateRepository",
"ecr:Describe*",
"ecr:GetAuthorizationToken",
"ecr:GetDownloadUrlForLayer",
"elastic-inference:Connect",
"elasticfilesystem:DescribeFileSystems",
"elasticfilesystem:DescribeMountTargets",
"fsx:DescribeFileSystems",
"glue:CreateJob",
"glue:DeleteJob",
"glue:GetJob",
"glue:GetJobRun",
"glue:GetJobRuns",
"glue:GetJobs",
"glue:ResetJobBookmark",
"glue:StartJobRun",
"glue:UpdateJob",
"groundtruthlabeling:*",
"iam:ListRoles",
"kms:DescribeKey",
"kms:ListAliases",
"lambda:ListFunctions",
"logs:CreateLogGroup",
"logs:CreateLogStream",
"logs:DescribeLogGroups",
"logs:DescribeLogStreams",
"logs:GetLogEvents",
"logs:PutLogEvents",
"sns:ListTopics"
],
"Resource": "*
},
{
"Effect": "Allow",
"Action": [
"logs:CreateLogDelivery",
"logs:DeleteLogDelivery",
"logs:DescribeResourcePolicies",
"logs:GetLogDelivery",
"logs:ListLogDeliveries",
"logs:PutLogResourcePolicy",
"logs:UpdateLogDelivery"
],
"Resource": "*
},
{
"Effect": "Allow",
"Action": [
"ecr:SetRepositoryPolicy",
"ecr:BatchCheckLayerAvailability",
"ecr:BatchGetImage",
"ecr:CreateRepository",
"ecr:Describe*",
"ecr:GetAuthorizationToken",
"ecr:GetDownloadUrlForLayer",
"elastic-inference:Connect",
"elasticfilesystem:DescribeFileSystems",
"elasticfilesystem:DescribeMountTargets",
"fsx:DescribeFileSystems",
"glue:CreateJob",
"glue:DeleteJob",
"glue:GetJob",
"glue:GetJobRun",
"glue:GetJobRuns",
"glue:GetJobs",
"glue:ResetJobBookmark",
"glue:StartJobRun",
"glue:UpdateJob",
"groundtruthlabeling:*",
"iam:ListRoles",
"kms:DescribeKey",
"kms:ListAliases",
"lambda:ListFunctions",
"logs:CreateLogGroup",
"logs:CreateLogStream",
"logs:DescribeLogGroups",
"logs:DescribeLogStreams",
"logs:GetLogEvents",
"logs:PutLogEvents",
"sns:ListTopics"
],
"Resource": "*
}
"ecr:CompleteLayerUpload",
"ecr:BatchDeleteImage",
"ecr:UploadLayerPart",
"ecr:DeleteRepositoryPolicy",
"ecr:InitiateLayerUpload",
"ecr:DeleteRepository",
"ecr:PutImage"
],
"Resource": "arn:aws:ecr:*:*:repository/*sagemaker**",

},
{
"Effect": "Allow",
"Action": [
  "codecommit:GitPull",
  "codecommit:GitPush"
],
"Resource": [
  "arn:aws:codecommit:*:*:*sagemaker*",
  "arn:aws:codecommit:*:*:*SageMaker*",
  "arn:aws:codecommit:*:*:*Sagemaker*"
]
},
{
"Effect": "Allow",
"Action": [
  "secretsmanager:ListSecrets"
],
"Resource": "*"
},
{
"Effect": "Allow",
"Action": [
  "secretsmanager:DescribeSecret",
  "secretsmanager:GetSecretValue",
  "secretsmanager:CreateSecret"
],
"Resource": [
  "arn:aws:secretsmanager:*:*:secret:AmazonSageMaker-*"
]
},
{
"Effect": "Allow",
"Action": [
  "secretsmanager:DescribeSecret",
  "secretsmanager:GetSecretValue"
],
"Resource": "*",
"Condition": {
  "StringEquals": {
    "secretsmanager:ResourceTag/SageMaker": "true"
  }
}
},
{
"Effect": "Allow",
"Action": [
  "robomaker:CreateSimulationApplication",
  "robomaker:DescribeSimulationApplication",
  "robomaker:DeleteSimulationApplication"
],
"Resource": [
  "*
]
},
{
"Effect": "Allow",

}
"Action": [  "robomaker:CreateSimulationJob",  "robomaker:DescribeSimulationJob",  "robomaker:CancelSimulationJob"
],
"Resource": [  "*"
],
"Effect": "Allow",
],
],
"Effect": "Allow",
"Action": [  "s3:CreateBucket",  "s3:GetBucketLocation",  "s3:ListBucket",  "s3:ListAllMyBuckets"
],
"Resource": "*",
"Effect": "Allow",
"Action": [  "s3:GetObject"
],
"Resource": "*",
"Condition": {  "StringEqualsIgnoreCase": {  "s3:ExistingObjectTag/SageMaker": "true"
}}
},
"Effect": "Allow",
"Action": [  "lambda:InvokeFunction"
],
],
"Action": "iam:CreateServiceLinkedRole",
"Effect": "Allow",
"Resource": "arn:aws:iam::*:role/aws-service-role/sagemaker.application-autoscaling.amazonaws.com/AWSServiceRoleForApplicationAutoScaling_SageMakerEndpoint",
"Condition": {

Allow Users to View Their Own Permissions

This example shows how you might create a policy that allows IAM users to view the inline and managed policies that are attached to their user identity. This policy includes permissions to complete this action on the console or programmatically using the AWS CLI or AWS API.

```json
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "ViewOwnUserInfo",
            "Effect": "Allow",
            "Action": [
                "iam:GetUserPolicy",
                "iam:ListGroupsForUser",
                "iam:ListAttachedUserPolicies",
                "iam:ListUserPolicies",
            ],
            "Resource": ["arn:aws:iam::*:user/*"]
        }
    ]
}
```
Identity-Based Policy Examples

```
  "iam:GetUser",
  "Resource": ["arn:aws:iam::*:user/${aws:username}"],
},
{
  "Sid": "NavigateInConsole",
  "Effect": "Allow",
  "Action": [
    "iam:GetGroupPolicy",
    "iam:GetPolicyVersion",
    "iam:GetPolicy",
    "iam:ListAttachedGroupPolicies",
    "iam:ListGroupPolicies",
    "iam:ListPolicyVersions",
    "iam:ListPolicies",
    "iam:ListUsers"
  ],
  "Resource": "*"
}
```

Control Creation of SageMaker Resources with Condition Keys

Control fine-grained access to allow the creation of SageMaker resources by using SageMaker-specific condition keys. For information about using condition keys in IAM policies, see IAM JSON Policy Elements: Condition in the IAM User Guide.

The condition keys, along with related API actions, and links to relevant documentation are listed in Condition Keys for SageMaker in the IAM User Guide.

The following examples show how to use the SageMaker condition keys to control access.

Topics
- Control Access to SageMaker Resources by Using File System Condition Keys (p. 1635)
- Restrict Training to a Specific VPC (p. 1637)
- Restrict Access to Workforce Types for Ground Truth Labeling Jobs and Amazon A2I Human Review Workflows (p. 1638)
- Enforce Encryption of Input Data (p. 1639)
- Enforce Encryption of Notebook Instance Storage Volume (p. 1639)
- Enforce Network Isolation for Training Jobs (p. 1640)
- Enforce a Specific Instance Type for Training Jobs (p. 1640)
- Enforce a Specific EI Accelerator for Training Jobs (p. 1641)
- Enforce Disabling Internet Access and Root Access for Creating Notebook Instances (p. 1641)

Control Access to SageMaker Resources by Using File System Condition Keys

SageMaker training provides a secure infrastructure for the training algorithm to run in, but for some cases you may want increased defense in depth. For example, you minimize the risk of running untrusted code in your algorithm, or you have specific security mandates in your organization. For these scenarios, you can use the service-specific condition keys in the Condition element of an IAM policy to scope down the user to specific file systems, directories, access modes (read-write, read-only) and security groups.

Topics
- Restrict an IAM User to Specific Directories and Access Modes (p. 1636)
- Restrict an IAM User to a Specific File System (p. 1636)
Restrict an IAM User to Specific Directories and Access Modes

The policy below restricts an IAM user to the /sagemaker/xgboost-dm/train and /sagemaker/xgboost-dm/validation directories of an EFS file system to ro (read-only) AccessMode:

**Note**
When a directory is allowed, all of its subdirectories are also accessible by the training algorithm. POSIX permissions are ignored.

```json
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "AccessToElasticFileSystem",
      "Effect": "Allow",
      "Action": [
        "sagemaker:CreateTrainingJob",
        "sagemaker:CreateHyperParameterTuningJob"
      ],
      "Resource": "*",
      "Condition": {
        "StringEquals": {
          "sagemaker:FileSystemId": "fs-12345678",
          "sagemaker:FileSystemAccessMode": "ro",
          "sagemaker:FileSystemType": "EFS",
          "sagemaker:FileSystemDirectoryPath": "/sagemaker/xgboost-dm/train"
        }
      }
    },
    {
      "Sid": "AccessToElasticFileSystemValidation",
      "Effect": "Allow",
      "Action": [
        "sagemaker:CreateTrainingJob",
        "sagemaker:CreateHyperParameterTuningJob"
      ],
      "Resource": "*",
      "Condition": {
        "StringEquals": {
          "sagemaker:FileSystemId": "fs-12345678",
          "sagemaker:FileSystemAccessMode": "ro",
          "sagemaker:FileSystemType": "EFS",
          "sagemaker:FileSystemDirectoryPath": "/sagemaker/xgboost-dm/validation"
        }
      }
    }
  ]
}
```

Restrict an IAM User to a Specific File System

To prevent a malicious algorithm using a user space client from accessing any file system directly in your account, you can restrict networking traffic by allowing ingress from a specific security group. In the following example, the IAM user can only use the specified security group to access the file system:

```json
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "AccessToLustreFileSystem",
      "Effect": "Allow",
      "Action": [
        "sagemaker:CreateTrainingJob",
        "sagemaker:CreateHyperParameterTuningJob"
      ],
      "Resource": "*",
      "Condition": {
        "StringEquals": {
          "sagemaker:FileSystemId": "fs-12345678",
          "sagemaker:FileSystemAccessMode": "ro",
          "sagemaker:FileSystemType": "Lustre",
          "sagemaker:FileSystemDirectoryPath": "/sagemaker/xgboost-dm/train"
        }
      }
    }
  ]
}
```
Although the above example can restrict an algorithm to a specific file system, it does not prevent an algorithm from accessing any directory within that file system using the user space client. To mitigate this, you can:

- Ensure that the file system only contains data that you trust your IAM users to access
- Create an IAM role that restricts your IAM users to launching training jobs with algorithms from approved ECR repositories

For more information on how to use roles with SageMaker, see SageMaker Roles.

**Restrict Training to a Specific VPC**

Restrict an AWS user to creating training jobs from within a Amazon VPC. When a training job is created within a VPC, you can use VPC flow logs to monitor all traffic to and from the training cluster. For information about using VPC flow logs, see VPC Flow Logs in the Amazon Virtual Private Cloud User Guide.

The following policy enforces that a training job is created by an IAM user calling CreateTrainingJob from within a VPC:

```json
{
   "Version": "2012-10-17",
   "Statement": [ 
      { 
         "Sid": "AllowFromVpc",
         "Effect": "Allow",
         "Action": [ 
            "sagemaker:CreateTrainingJob",
            "sagemaker:CreateHyperParameterTuningJob"
         ],
         "Resource": "*",
         "Condition": { 
            "ForAllValues:StringEquals": { 
               "sagemaker:VpcSubnets": ["subnet-a1234"],
               "sagemaker:VpcSecurityGroupIds": ["sg12345", "sg-67890"]
            },
            "Null": { 
               "sagemaker:VpcSubnets": "false",
               "sagemaker:VpcSecurityGroupIds": "false"
            }
         }
      }
   ]
}
```
Identity-Based Policy Examples

Restrict Access to Workforce Types for Ground Truth Labeling Jobs and Amazon A2I Human Review Workflows

Amazon SageMaker Ground Truth and Amazon Augmented AI work teams fall into one of three workforce types: public (with Amazon Mechanical Turk), private, and vendor. To restrict IAM user access to a specific work team using one of these types or the work team ARN, use the `sagemaker:WorkteamType` and/or the `sagemaker:WorkteamArn` condition keys.

For the `sagemaker:WorkteamType` condition key, use string condition operators. For the `sagemaker:WorkteamArn` condition key, use Amazon Resource Name (ARN) condition operators. If the user attempts to create a labeling job with a restricted work team, SageMaker returns an access denied error.

The policies below demonstrate different ways to use the `sagemaker:WorkteamType` and `sagemaker:WorkteamArn` condition keys with appropriate condition operators and valid condition values.

The following example uses the `sagemaker:WorkteamType` condition key with the `StringEquals` condition operator to restrict access to a public work team. It accepts condition values in the following format: `workforcetype-crowd`, where `workforcetype` can equal public, private, or vendor.

```json
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "RestrictWorkteamType",
            "Effect": "Deny",
            "Action": "sagemaker:CreateLabelingJob",
            "Resource": "*",
            "Condition": {
                "StringEquals": {
                    "sagemaker:WorkteamType": "public-crowd"
                }
            }
        }
    ]
}
```

The following policies show how to restrict access to a public work team using the `sagemaker:WorkteamArn` condition key. The first shows how to use it with a valid IAM regex-variant of the work team ARN and the `ArnLike` condition operator. The second shows how to use it with the `ArnEquals` condition operator and the work team ARN.

```json
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "RestrictWorkteamType",
            "Effect": "Deny",
            "Action": "sagemaker:CreateLabelingJob",
            "Resource": "*",
            "Condition": {
                "ArnLike": {
                    "sagemaker:WorkteamArn": "arn:aws:sagemaker:*:*:workteam/public-crowd/*"
                }
            }
        }
    ]
}
```
Identity-Based Policy Examples

Enforce Encryption of Input Data

The following policy restricts an IAM user to specify an AWS KMS key to encrypt input data when creating training, hyperparameter tuning, and labeling jobs by using the `sagemaker:VolumeKmsKey` condition key:

```
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Sid": "EnforceEncryption",
         "Effect": "Allow",
         "Action": ["sagemaker:CreateTrainingJob",
                     "sagemaker:CreateHyperParameterTuningJob",
                     "sagemaker:CreateLabelingJob",
                     "sagemaker:CreateFlowDefinition"],
         "Resource": "*",
         "Condition": {
            "ArnEquals": {
               "sagemaker:VolumeKmsKey": "arn:aws:kms:us-west-2:111122233333:key/1234abcd-12ab-34cd-56ef-1234567890ab"
            }
         }
      }
   ]
}
```

Enforce Encryption of Notebook Instance Storage Volume

The following policy restricts an IAM user to specify an AWS KMS key to encrypt the attached storage volume when creating or updating a notebook instance by using the `sagemaker:VolumeKmsKey` condition key:

```
{
   "Version": "2012-10-17",
   "Statement": [
      {
```
Enforce Network Isolation for Training Jobs

The following policy restricts an IAM user to enable network isolation when creating training jobs by using the `sagemaker:NetworkIsolation` condition key:

```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "EnforceIsolation",
      "Effect": "Allow",
      "Action": [
        "sagemaker:CreateTrainingJob",
        "sagemaker:CreateHyperParameterTuningJob"
      ],
      "Resource": "*",
      "Condition": {
        "Bool": {
          "sagemaker:NetworkIsolation": "true"
        }
      }
    }
  ]
}
```

Enforce a Specific Instance Type for Training Jobs

The following policy restricts an IAM user to use a specific instance type when creating training jobs by using the `sagemaker:InstanceTypes` condition key:

```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "EnforceInstanceType",
      "Effect": "Allow",
      "Action": [
        "sagemaker:CreateTrainingJob",
        "sagemaker:CreateHyperParameterTuningJob"
      ],
      "Resource": "*",
      "Condition": {
        "ForAllValues:StringLike": {
          "sagemaker:InstanceTypes": ["ml.c5.*"]
        }
      }
    }
  ]
}
```
Enforce a Specific EI Accelerator for Training Jobs

The following policy restricts an IAM user to use a specific elastic inference (EI) accelerator, if an accelerator is provided, when creating or updating notebook instances and when creating endpoint configurations by using the `sagemaker:AcceleratorTypes` condition key:

```json
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Sid": "EnforceAcceleratorType",
         "Effect": "Allow",
         "Action": [
            "sagemaker:CreateNotebookInstance",
            "sagemaker:UpdateNotebookInstance",
            "sagemaker:CreateEndpointConfig"
         ],
         "Resource": "*",
         "Condition": {
            "ForAllValues:StringEquals": {
               "sagemaker:AcceleratorTypes": ["ml.eia1.medium"],
            }
         }
      }
   ]
}
```

Enforce Disabling Internet Access and Root Access for Creating Notebook Instances

You can disable both internet access and root access to notebook instances to help make them more secure. For information about controlling root access to a notebook instance, see Control root access to a SageMaker notebook instance (p. 1613). For information about disabling internet access for a notebook instance, see Connect a Notebook Instance to Resources in a VPC (p. 1681).

The following policy requires an IAM user to disable network access when creating instance, and disable root access when creating or updating a notebook instance.

```json
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Sid": "LockDownCreateNotebookInstance",
         "Effect": "Allow",
         "Action": [
            "sagemaker:CreateNotebookInstance"
         ],
         "Resource": "*",
         "Condition": {
            "StringEquals": {
               "sagemaker:DirectInternetAccess": "Disabled",
               "sagemaker:RootAccess": "Disabled"
            },
            "Null": {
               "sagemaker:VpcSubnets": "false",
               "sagemaker:VpcSecurityGroupIds": "false"
            }
         }
      }
   ]
}
```
Control Access to the SageMaker API by Using Identity-based Policies

To control access to SageMaker API calls and calls to SageMaker hosted endpoints, use identity-based IAM policies.

Topics
- Restrict Access to SageMaker API and Runtime to Calls from Within Your VPC (p. 1642)

Restrict Access to SageMaker API and Runtime to Calls from Within Your VPC

If you set up an interface endpoint in your VPC, individuals outside the VPC can still connect to the SageMaker API and runtime over the internet unless you attach an IAM policy that restricts access to calls coming from within the VPC to all users and groups that have access to your SageMaker resources. For information about creating a VPC interface endpoint for the SageMaker API and runtime, see Connect to SageMaker Through a VPC Interface Endpoint (p. 1683).

Important
If you apply an IAM policy similar to one of the following, users can't access the specified SageMaker APIs through the console.

To restrict access to only connections made from within your VPC, create an AWS Identity and Access Management policy that restricts access to only calls that come from within your VPC. Then add that policy to every AWS Identity and Access Management user, group, or role used to access the SageMaker API or runtime.

Note
This policy allows connections only to callers within a subnet where you created an interface endpoint.
"StringEquals": {  
  "aws:SourceVpc": "vpc-111bbaaa"
}
}
]

If you want to restrict access to the API to only calls made using the interface endpoint, use the `aws:SourceVpce` condition key instead of `aws:SourceVpc`:

```json
{
  "Id": "api-example-1",
  "Version": "2012-10-17",
  "Statement": [
    {  
      "Sid": "Enable API Access",
      "Effect": "Allow",
      "Action": [  
        "sagemaker:CreatePresignedNotebookInstanceUrl"
      ],
      "Resource": "*",
      "Condition": {
        "ForAllValues:StringEquals": {  
          "aws:sourceVpce": [  
            "vpce-111bbccc",
            "vpce-111bbddd"
          ]
        }
      }
    }
  ]
}
```

**Limit Access to SageMaker API and Runtime Calls by IP Address**

To allow access to SageMaker API calls and runtime invocations only from IP addresses in a list that you specify, attach an IAM policy that denies access to the API unless the call comes from an IP address in the list to every AWS Identity and Access Management user, group, or role used to access the API or runtime. For information about creating IAM policies, see [Creating IAM Policies in the AWS Identity and Access Management User Guide](https://docs.aws.amazon.com/IAM/latest/UserGuide/id_Policy-creating.html). To specify the list of IP addresses that you want to have access to the API call, use the `IpAddress` condition operator and the `aws:SourceIp` condition context key. For information about IAM condition operators, see [IAM JSON Policy Elements: Condition Operators in the AWS Identity and Access Management User Guide](https://docs.aws.amazon.com/IAM/latest/UserGuide/id_Policy-creating.html). For information about IAM condition context keys, see [AWS Global Condition Context Keys](https://docs.aws.amazon.com/IAM/latest/UserGuide/id_Policy-creating.html).

For example, the following policy allows access to the `CreateTrainingJob` only from IP addresses in the ranges 192.0.2.0-192.0.2.255 and 203.0.113.0-203.0.113.255:

```json
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": "sagemaker:CreateTrainingJob",
      "Resource": "*",
      "Condition": {
        "IpAddress": {
          "aws:SourceIp": [
            "192.0.2.0/24",
            "203.0.113.0/24"
          ]
        }
      }
    }
  ]
}
```
Limit Access to a Notebook Instance by IP Address

To allow access to a notebook instance only from IP addresses in a list that you specify, attach an IAM policy that denies access to `CreatePresignedNotebookInstanceUrl` unless the call comes from an IP address in the list to every AWS Identity and Access Management user, group, or role used to access the notebook instance. For information about creating IAM policies, see Creating IAM Policies in the AWS Identity and Access Management User Guide. To specify the list of IP addresses that you want to have access to the notebook instance, use the `IpAddress` condition operator and the `aws:SourceIP` condition context key. For information about IAM condition operators, see IAM JSON Policy Elements: Condition Operators in the AWS Identity and Access Management User Guide. For information about IAM condition context keys, see AWS Global Condition Context Keys.

For example, the following policy allows access to a notebook instance only from IP addresses in the ranges `192.0.2.0-192.0.2.255` and `203.0.113.0-203.0.113.255`:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": "sagemaker:CreatePresignedNotebookInstanceUrl",
      "Resource": "*",
      "Condition": {
        "IpAddress": {"aws:SourceIp": ["192.0.2.0/24", "203.0.113.0/24"]
      }
    }
  ]
}
```

The policy restricts access to both the call to `CreatePresignedNotebookInstanceUrl` and to the URL that the call returns. The policy also restricts access to opening a notebook instance in the console and is enforced for every HTTP request and WebSocket frame that attempts to connect to the notebook instance.

**Note**
Using this method to filter by IP address is incompatible when connecting to SageMaker through a VPC interface endpoint. For information about restricting access to a notebook instance when connecting through a VPC interface endpoint, see Connect to a Notebook Instance Through a VPC Interface Endpoint (p. 1684).

Control Access to SageMaker Resources by Using Tags

Control access to groups of SageMaker resources by attaching tags to the resources and specifying `ResourceTag` conditions in IAM policies.

**Note**
Tag-based policies don't work to restrict the following API calls:
Identity-Based Policy Examples

- ListAlgorithms
- ListCodeRepositories
- ListCompilationJobs
- ListEndpointConfigs
- ListEndpoints
- ListFlowDefinitions
- ListHumanTaskUis
- ListHyperparameterTuningJobs
- ListLabelingJobs
- ListLabelingJobsForWorkteam
- ListModelPackages
- ListModels
- ListNotebookInstanceLifecycleConfigs
- ListNotebookInstances
- ListSubscribedWorkteams
- ListTags
- ListProcessingJobs
- ListTrainingJobs
- ListTrainingJobsForHyperParameterTuningJob
- ListTransformJobs
- ListWorkteams
- Search

For example, suppose you’ve defined two different IAM groups, named DevTeam1 and DevTeam2, in your AWS account. Suppose also that you’ve created 10 notebook instances, 5 of which are used for one project, and 5 of which are used for a second project. You want to allow members of DevTeam1 to make API calls on notebook instances used for the first project, and members of DevTeam2 to make API calls on notebook instances used for the second project.

To control access to API calls (example)

1. Add a tag with the key Project and value A to the notebook instances used for the first project. For information about adding tags to SageMaker resources, see AddTags.
2. Add a tag with the key Project and value B to the notebook instances used for the second project.
3. Create an IAM policy with a ResourceTag condition that denies access to the notebook instances used for the second project, and attach that policy to DevTeam1. The following is an example of a policy that denies all API calls on any notebook instance that has a tag with a key of Project and a value of B:

   ```json
   {
     "Version": "2012-10-17",
     "Statement": [
       {
         "Effect": "Allow",
         "Action": "sagemaker:*",
         "Resource": "*"
       },
       {
         "Effect": "Deny",
         "Action": "sagemaker:*",
         "Resource": "*",
         "Condition": {
           "ResourceTag": {
             "Project": {"Values": ["B"]}
           }
         }
       }
     ]
   }
   ```
For information about creating IAM policies and attaching them to identities, see Controlling Access Using Policies in the AWS Identity and Access Management User Guide.

4. Create an IAM policy with a ResourceTag condition that denies access to the notebook instances used for the first project, and attach that policy to DevTeam2. The following is an example of a policy that denies all API calls on any notebook instance that has a tag with a key of Project and a value of A:

```json
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": "*",
      "Resource": "*"
    },
    {
      "Effect": "Deny",
      "Action": "sagemaker:*",
      "Resource": "*",
      "Condition": {
        "StringEquals": {
          "sagemaker:ResourceTag/Project": "A"
        }
      }
    },
    {
      "Effect": "Deny",
      "Action": ["sagemaker:AddTags", "sagemaker:DeleteTags"],
      "Resource": "*"
    }
  ]
}
```

Require the Presence or Absence of Tags for API Calls

Require the presence or absence of specific tags or specific tag values by using RequestTag condition keys in an IAM policy. For example, if you want to require that every endpoint created by any member of an IAM group to be created with a tag with the key environment and value dev, create a policy as follows:

```json
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": "*",
      "Resource": "*"
    },
    {
      "Effect": "Deny",
      "Action": "sagemaker:*",
      "Resource": "*",
      "Condition": {
        "StringEquals": {
          "sagemaker:ResourceTag/Project": "A"
        }
      }
    },
    {
      "Effect": "Deny",
      "Action": ["sagemaker:AddTags", "sagemaker:DeleteTags"],
      "Resource": "*"
    }
  ]
}
```
SageMaker Roles

As a managed service, SageMaker performs operations on your behalf on the AWS hardware that is managed by SageMaker. SageMaker can perform only operations that the user permits.

A SageMaker user can grant these permissions with an IAM role (referred to as an execution role).

To create and use a locally available execution role, you can use the following procedures.
Get execution role

When you run a notebook within SageMaker you can access the execution role with the following code:

```python
sagemaker_session = sagemaker.Session()
role = sagemaker.get_execution_role()
```

**Note**
The execution role is intended to be available only when running a notebook within SageMaker.
If you run `get_execution_role` in a notebook not on SageMaker, expect a "region" error.

To find the IAM role ARN created when you created the notebook instance or Studio application, go to the **Notebook instances** page in the console and select the relevant notebook from the list of **Names**.
in the configuration detail page the IAM role ARN is given in the **Permissions and encryption** section.

**To create a new role**

1. Log onto the console -> IAM -> Roles -> Create Role
2. Create a service-linked role with `sagemaker.amazonaws.com`
3. Give the role `AmazonSageMakerFullAccess`
4. Give the role `AmazonS3FullAccess` (limit the permissions to specific buckets if possible)
5. Make note of the ARN once it is created

With a known ARN for your role, you can programmatically check the role when running the notebook locally or on SageMaker. Replace `RoleName` with your known ARN:

```python
try:
    role = sagemaker.get_execution_role()
except ValueError:
    iam = boto3.client('iam')
    role = iam.get_role(RoleName='AmazonSageMaker-ExecutionRole-20201200T100000')['Role']['Arn']
```

**Passing Roles**

Actions like passing a role between services are a common function within SageMaker. You can find more details on **Actions**, **Resources**, and **Condition Keys for SageMaker** in the [IAM User Guide](#).

You pass the role (`iam:PassRole`) when making these API calls: `CreateAutoMLJob`, `CreateCompilationJob`, `CreateDomain`, `CreateFlowDefinition`, `CreateHyperParameterTuningJob`, `CreateImage`, `CreateLabelingJob`, `CreateModel`, `CreateMonitoringSchedule`, `CreateNotebookInstance`, `CreateProcessingJob`, `CreateTrainingJob`, `CreateUserProfile`, `RenderUiTemplate`, and `UpdateImage`.

You attach the following trust policy to the IAM role which grants SageMaker principal permissions to assume the role, and is the same for all of the execution roles:

```json
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {
                "Service": "sagemaker.amazonaws.com"
            },
            "Action": "sts:AssumeRole"
        }
    ]
}
```
The permissions that you need to grant to the role vary depending on the API that you call. The following sections explain these permissions.

**Note**
Instead of managing permissions by crafting a permission policy, you can use the AWS-managed AmazonSageMakerFullAccess permission policy. The permissions in this policy are fairly broad, to allow for any actions you might want to perform in SageMaker. For a listing of the policy including information about the reasons for adding many of the permissions, see AmazonSageMakerFullAccess Policy (p. 1662). If you prefer to create custom policies and manage permissions to scope the permissions only to the actions you need to perform with the execution role, see the following topics.

For more information about IAM roles, see IAM Roles in the IAM User Guide.

**Topics**
- CreateDomain API: Execution Role Permissions (p. 1649)
- CreateImage and UpdateImage APIs: Execution Role Permissions (p. 1650)
- CreateNotebookInstance API: Execution Role Permissions (p. 1650)
- CreateHyperParameterTuningJob API: Execution Role Permissions (p. 1653)
- CreateProcessingJob API: Execution Role Permissions (p. 1655)
- CreateTrainingJob API: Execution Role Permissions (p. 1658)
- CreateModel API: Execution Role Permissions (p. 1660)
- AmazonSageMakerFullAccess Policy (p. 1662)

### CreateDomain API: Execution Role Permissions

The execution role for AWS SSO domains and the user/execution role for IAM domains need the following permissions when you pass an AWS KMS customer managed key (CMK) as the `KmsKeyId` in the CreateDomain API request. The permissions are enforced during the CreateApp API call.

For an execution role that you can pass in the CreateDomain API request, you can attach the following permission policy to the role:

```json
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "kms:CreateGrant",
        "kms:DescribeKey"
      ],
      "Resource": "arn:aws:kms:region:account-id:key/kms-key-id"
    }
  ]
}
```

Alternatively, if the permissions are specified in a KMS policy, you can attach the following policy to the role:

```json
{
  "Sid": "Allow use of the key",
  "Effect": "Allow",
  "Principal": {
    "AWS": "*"
  }
}
```
CreateImage and UpdateImage APIs: Execution Role Permissions

For an execution role that you can pass in a CreateImage or UpdateImage API request, you can attach the following permission policy to the role:

```json
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "ecr:BatchGetImage",
        "ecr:GetDownloadUrlForLayer"
      ],
      "Resource": "*"
    },
    {
      "Effect": "Allow",
      "Action": [
        "iam:PassRole"
      ],
      "Resource": "*",
      "Condition": {
        "StringEquals": {
          "iam:PassedToService": "sagemaker.amazonaws.com"
        }
      }
    }
  ]
}
```

CreateNotebookInstance API: Execution Role Permissions

The permissions that you grant to the execution role for calling the CreateNotebookInstance API depend on what you plan to do with the notebook instance. If you plan to use it to invoke SageMaker APIs and pass the same role when calling the CreateTrainingJob and CreateModel APIs, attach the following permissions policy to the role:

```json
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "sagemaker:*",
        "ecr:GetAuthorizationToken",
        "ecr:GetDownloadUrlForLayer",
        "ecr:BatchGetImage",
        "ecr:BatchCheckLayerAvailability",
        "ecr:SetRepositoryPolicy",
```
To tighten the permissions, limit them to specific Amazon S3 and Amazon ECR resources, by restricting "Resource": "*", as follows:
```json
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "sagemaker:*",
                "ecr:GetAuthorizationToken",
                "cloudwatch:PutMetricData",
                "logs:CreateLogGroup",
                "logs:CreateLogStream",
                "logs:DescribeLogStreams",
                "logs:PutLogEvents",
                "logs:GetLogEvents"
            ],
            "Resource": "*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "iam:PassRole"
            ],
            "Resource": "*",
            "Condition": {
                "StringEquals": {
                    "iam:PassedToService": "sagemaker.amazonaws.com"
                }
            }
        },
        {
            "Effect": "Allow",
            "Action": [
                "s3:ListBucket"
            ],
            "Resource": [
                "arn:aws:s3:::inputbucket"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "s3:GetObject",
                "s3:PutObject",
                "s3:DeleteObject"
            ],
            "Resource": [
                "arn:aws:s3:::inputbucket/object1",
                "arn:aws:s3:::outputbucket/path",
                "arn:aws:s3:::inputbucket/object2",
                "arn:aws:s3:::inputbucket/object3"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "ecr:BatchCheckLayerAvailability",
                "ecr:GetDownloadUrlForLayer",
                "ecr:BatchGetImage"
            ],
            "Resource": [
                "arn:aws:ecr:::repository/my-repo1",
                "arn:aws:ecr:::repository/my-repo2",
                "arn:aws:ecr:::repository/my-repo3"
            ]
        }
    ]
}
```
If you plan to access other resources, such as Amazon DynamoDB or Amazon Relational Database Service, add the relevant permissions to this policy.

In the preceding policy, you scope the policy as follows:

- **Scope the `s3:ListBucket` permission to the specific bucket that you specify as `InputDataConfig.DataSource.S3DataSource.S3Uri` in a `CreateTrainingJob` request.**
- **Scope `s3:GetObject`, `s3:PutObject`, and `s3:DeleteObject` permissions as follows:**
  - Scope to the following values that you specify in a `CreateTrainingJob` request:
    - `OutputDataConfig.S3OutputPath`
    - `SuplementalContainers.ModelDataUrl`
- **Scope ecr permissions as follows:**
  - Scope to the `AlgorithmSpecification.TrainingImage` value that you specify in a `CreateTrainingJob` request.
  - Scope to the `PrimaryContainer.Image` value that you specify in a `CreateModel` request.

The `cloudwatch` and `logs` actions are applicable for "*" resources. For more information, see CloudWatch Resources and Operations in the Amazon CloudWatch User Guide.

### CreateHyperParameterTuningJob API: Execution Role Permissions

For an execution role that you can pass in a `CreateHyperParameterTuningJob` API request, you can attach the following permission policy to the role:

```json
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "cloudwatch:PutMetricData",
        "logs:CreateLogStream",
        "logs:PutLogEvents",
        "logs:CreateLogGroup",
        "logs:DescribeLogStreams",
        "s3:GetObject",
        "s3:PutObject",
        "s3:ListBucket",
        "ecr:GetAuthorizationToken",
        "ecr:BatchCheckLayerAvailability",
        "ecr:GetDownloadUrlForLayer",
        "ecr:BatchGetImage"
      ],
      "Resource": "*"
    }
  ]
}
```
Instead of the specifying "Resource": "*", you could scope these permissions to specific Amazon S3 and Amazon ECR resources:

```json
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "cloudwatch:PutMetricData",
                "logs:CreateLogStream",
                "logs:PutLogEvents",
                "logs:CreateLogGroup",
                "logs:DescribeLogStreams",
                "ecr:GetAuthorizationToken"
            ],
            "Resource": "*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "s3:ListBucket"
            ],
            "Resource": [
                "arn:aws:s3:::inputbucket"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "s3:GetObject",
                "s3:PutObject"
            ],
            "Resource": [
                "arn:aws:s3:::inputbucket/object",
                "arn:aws:s3:::outputbucket/path"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "ecr:BatchCheckLayerAvailability",
                "ecr:GetDownloadUrlForLayer",
                "ecr:BatchGetImage"
            ],
            "Resource": "arn:aws:ecr:::repository/my-repo"
        }
    ]
}
```

If the training container associated with the hyperparameter tuning job needs to access other data sources, such as DynamoDB or Amazon RDS resources, add relevant permissions to this policy.

In the preceding policy, you scope the policy as follows:

- Scope the `s3:ListBucket` permission to a specific bucket that you specify as the `InputDataConfig.DataSource.S3DataSource.S3Uri` in a `CreateTrainingJob` request.
- Scope the `s3:GetObject` and `s3:PutObject` permissions to the following objects that you specify in the input and output data configuration in a `CreateHyperParameterTuningJob` request:

    `InputDataConfig.DataSource.S3DataSource.S3Uri`
OutputDataConfig.S3OutputPath

- Scope Amazon ECR permissions to the registry path (AlgorithmSpecification.TrainingImage) that you specify in a CreateHyperParameterTuningJob request.

The `cloudwatch` and `logs` actions are applicable for "*" resources. For more information, see CloudWatch Resources and Operations in the Amazon CloudWatch User Guide.

If you specify a private VPC for your hyperparameter tuning job, add the following permissions:

```json
{
    "Effect": "Allow",
    "Action": [
        "ec2:CreateNetworkInterface",
        "ec2:CreateNetworkInterfacePermission",
        "ec2:DeleteNetworkInterface",
        "ec2:DeleteNetworkInterfacePermission",
        "ec2:DescribeNetworkInterfaces",
        "ec2:DescribeVpcs",
        "ec2:DescribeDhcpOptions",
        "ec2:DescribeSubnets",
        "ec2:DescribeSecurityGroups"
    ]
}
```

If your input is encrypted using server-side encryption with an AWS KMS–managed key (SSE-KMS), add the following permissions:

```json
{
    "Effect": "Allow",
    "Action": ["kms:Decrypt"
}
```

If you specify a KMS key in the output configuration of your hyperparameter tuning job, add the following permissions:

```json
{
    "Effect": "Allow",
    "Action": ["kms:Encrypt"
}
```

If you specify a volume KMS key in the resource configuration of your hyperparameter tuning job, add the following permissions:

```json
{
    "Effect": "Allow",
    "Action": ["kms:CreateGrant"
}
```

CreateProcessingJob API: Execution Role Permissions

For an execution role that you can pass in a CreateProcessingJob API request, you can attach the following permission policy to the role:
Instead of the specifying "Resource": "*", you could scope these permissions to specific Amazon S3 and Amazon ECR resources:

```json
{
"Version": "2012-10-17",
"Statement": [
{
"Effect": "Allow",
"Action": [
"cloudwatch:PutMetricData",
"logs:CreateLogStream",
"logs:PutLogEvents",
"logs:CreateLogGroup",
"logs:DescribeLogStreams",
"s3:GetObject",
"s3:PutObject",
"s3:ListBucket",
"ecr:GetAuthorizationToken",
"ecr:BatchCheckLayerAvailability",
"ecr:GetDownloadUrlForLayer",
"ecr:BatchGetImage"
],
"Resource": "*"
}
]
}

```
If `CreateProcessingJob.AppSpecification.ImageUri` needs to access other data sources, such as DynamoDB or Amazon RDS resources, add relevant permissions to this policy.

In the preceding policy, you scope the policy as follows:

- Scope the `s3:ListBucket` permission to a specific bucket that you specify as the `ProcessingInputs` in a `CreateProcessingJob` request.
- Scope the `s3:GetObject` and `s3:PutObject` permissions to the objects that will be downloaded or uploaded in the `ProcessingInputs` and `ProcessingOutputConfig` in a `CreateProcessingJob` request.
- Scope Amazon ECR permissions to the registry path (`AppSpecification.ImageUri`) that you specify in a `CreateProcessingJob` request.

The `cloudwatch` and `logs` actions are applicable for "*" resources. For more information, see CloudWatch Resources and Operations in the Amazon CloudWatch User Guide.

If you specify a private VPC for your processing job, add the following permissions:

```json
{
  "Effect": "Allow",
  "Action": [
    "ec2:CreateNetworkInterface",
    "ec2:CreateNetworkInterfacePermission",
    "ec2:DeleteNetworkInterface",
    "ec2:DeleteNetworkInterfacePermission",
    "ec2:DescribeNetworkInterfaces",
    "ec2:DescribeDhcpOptions",
    "ec2:DescribeSubnets",
    "ec2:DescribeSecurityGroups"
  ]
}
```

If your input is encrypted using server-side encryption with an AWS KMS–managed key (SSE-KMS), add the following permissions:

```json
{
  "Effect": "Allow",
  "Action": [
    "kms:Decrypt"
  ]
}
```

If you specify a KMS key in the output configuration of your processing job, add the following permissions:

```json
{
  "Effect": "Allow",
  "Action": [
    "kms:Encrypt"
  ]
}
```
If you specify a volume KMS key in the resource configuration of your processing job, add the following permissions:

```json
{
  "Effect": "Allow",
  "Action": [
    "kms:CreateGrant"
  ]
}
```

**CreateTrainingJob API: Execution Role Permissions**

For an execution role that you can pass in a `CreateTrainingJob` API request, you can attach the following permission policy to the role:

```json
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "cloudwatch:PutMetricData",
        "logs:CreateLogStream",
        "logs:PutLogEvents",
        "logs:CreateLogGroup",
        "logs:DescribeLogStreams",
        "s3:GetObject",
        "s3:PutObject",
        "s3:ListBucket",
        "ecr:GetAuthorizationToken",
        "ecr:BatchCheckLayerAvailability",
        "ecr:GetDownloadUrlForLayer",
        "ecr:BatchGetImage"
      ],
      "Resource": "*"
    },
    {
      "Effect": "Allow",
      "Action": [
        "cloudwatch:PutMetricData",
        "logs:CreateLogStream",
        "logs:PutLogEvents",
        "logs:CreateLogGroup",
        "logs:DescribeLogStreams",
        "ecr:GetAuthorizationToken"
      ],
      "Resource": "*"
    }
  ]
}
```

Instead of the specifying `"Resource": "*"`, you could scope these permissions to specific Amazon S3 and Amazon ECR resources:

```json
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "cloudwatch:PutMetricData",
        "logs:CreateLogStream",
        "logs:PutLogEvents",
        "logs:CreateLogGroup",
        "logs:DescribeLogStreams",
        "ecr:GetAuthorizationToken"
      ],
      "Resource": "*"
    },
    {
      "Effect": "Allow",
      "Action": [
        "cloudwatch:PutMetricData",
        "logs:CreateLogStream",
        "logs:PutLogEvents",
        "logs:CreateLogGroup",
        "logs:DescribeLogStreams",
        "ecr:GetAuthorizationToken"
      ],
      "Resource": "*"
    },
    {
      "Effect": "Allow",
      "Action": [
        "cloudwatch:PutMetricData",
        "logs:CreateLogStream",
        "logs:PutLogEvents",
        "logs:CreateLogGroup",
        "logs:DescribeLogStreams",
        "ecr:GetAuthorizationToken"
      ],
      "Resource": "*"
    }
  ]
}
```
If `CreateTrainingJob.AlgorithmSpecifications.TrainingImage` needs to access other data sources, such as DynamoDB or Amazon RDS resources, add relevant permissions to this policy.

In the preceding policy, you scope the policy as follows:

- Scope the `s3:ListBucket` permission to a specific bucket that you specify as the `InputDataConfig.DataSource.S3DataSource.S3Uri` in a `CreateTrainingJob` request.
- Scope the `s3:GetObject` and `s3:PutObject` permissions to the following objects that you specify in the input and output data configuration in a `CreateTrainingJob` request:
  - `InputDataConfig.DataSource.S3DataSource.S3Uri`
  - `OutputDataConfig.S3OutputPath`
- Scope Amazon ECR permissions to the registry path (`AlgorithmSpecification.TrainingImage`) that you specify in a `CreateTrainingJob` request.

The `cloudwatch` and `logs` actions are applicable for `*` resources. For more information, see CloudWatch Resources and Operations in the Amazon CloudWatch User Guide.

If you specify a private VPC for your training job, add the following permissions:

```json
{
    "Effect": "Allow",
    "Action": [
        "ec2:CreateNetworkInterface",
        "ec2:CreateNetworkInterfacePermission",
        "ec2:DeleteNetworkInterface",
        "ec2:DeleteNetworkInterfacePermission",
        "ec2:DescribeNetworkInterfaces",
        "ec2:DescribeVpcs",
        "ec2:DescribeDhcpOptions",
        "ec2:DescribeSubnets",
    ],
    "Resource": "*"
}
```
If your input is encrypted using server-side encryption with an AWS KMS–managed key (SSE-KMS), add the following permissions:

```
{
  "Effect": "Allow",
  "Action": [
    "kms:Decrypt"
  ]
}
```

If you specify a KMS key in the output configuration of your training job, add the following permissions:

```
{
  "Effect": "Allow",
  "Action": [
    "kms:Encrypt"
  ]
}
```

If you specify a volume KMS key in the resource configuration of your training job, add the following permissions:

```
{
  "Effect": "Allow",
  "Action": [
    "kms:CreateGrant"
  ]
}
```

### CreateModel API: Execution Role Permissions

For an execution role that you can pass in a CreateModel API request, you can attach the following permission policy to the role:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "cloudwatch:PutMetricData",
        "logs:CreateLogStream",
        "logs:PutLogEvents",
        "logs:CreateLogGroup",
        "logs:DescribeLogStreams",
        "s3:GetObject",
        "ecr:GetAuthorizationToken",
        "ecr:BatchCheckLayerAvailability",
        "ecr:GetDownloadUrlForLayer",
        "ecr:BatchGetImage"
      ],
      "Resource": "*
    }
  ]
}
```

Instead of specifying "Resource": "*", you can scope these permissions to specific Amazon S3 and Amazon ECR resources:
If `CreateModel.PrimaryContainer.Image` need to access other data sources, such as Amazon DynamoDB or Amazon RDS resources, add relevant permissions to this policy.

In the preceding policy, you scope the policy as follows:

- **Scope S3 permissions to objects that you specify in the `PrimaryContainer.ModelDataUrl` in a `CreateModel` request.**
- **Scope Amazon ECR permissions to a specific registry path that you specify as the `PrimaryContainer.Image` and `SecondaryContainer.Image` in a `CreateModel` request.**

The `cloudwatch` and `logs` actions are applicable for "*" resources. For more information, see [CloudWatch Resources and Operations](https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/cloudwatch_resources.html) in the Amazon CloudWatch User Guide.

If you specify a private VPC for your model, add the following permissions:

```json
{
   "Effect": "Allow",
   "Action": [
      "ec2:CreateNetworkInterface",
      "ec2:CreateNetworkInterfacePermission",
      "ec2:DeleteNetworkInterface",
      "ec2:DeleteNetworkInterfacePermission"
   ],
   "Resource": "*"
}
```
AmazonSageMakerFullAccess Policy

The AmazonSageMakerFullAccess managed policy includes all of the necessary permissions to perform most actions in SageMaker. You can use attach this policy to any role that you pass to a SageMaker execution role. You can also create more narrowly-scoped policies if you want more granular control of the permissions that you grant to your execution role.

The following list explains why some of the categories of permissions in the AmazonSageMakerFullAccess policy are needed.

- application-autoscaling
  Needed for automatically scaling a SageMaker real-time inference endpoint.
- aws-marketplace
  Needed to view AWS AI Marketplace subscriptions.
- cloudwatch
  Needed to post CloudWatch metrics, interact with alarms, and upload CloudWatch Logs logs in your account.
- codecommit
  Needed for AWS CodeCommit integration with SageMaker notebook instances.
- cognito
  Needed for SageMaker Ground Truth to define your private workforce and work teams.
- ec2
  Needed to manage elastic network interfaces when you specify a Amazon VPC for your SageMaker jobs and notebook instances.
- ec2:DescribeVpcs
  All SageMaker services launch Amazon EC2 instances and require this permission set.
- ecr
  Needed to pull and store Docker artifacts for training and inference. This is required only if you use your own container in SageMaker.
- elastic-inference
  Needed to integrate Amazon Elastic Inference with SageMaker.
- glue
  Needed for inference pipeline pre-processing from within SageMaker notebook instances.
- groundtruthlabeling
  Needed for SageMaker Ground Truth.
- iam:ListRoles
  Needed to give the SageMaker console access to list available roles.
- kms
  Needed to give the SageMaker console access to list the available AWS KMS keys.
AWS Managed (Predefined) Policies for Amazon SageMaker

AWS addresses many common use cases by providing standalone IAM policies that are created and administered by AWS. These AWS managed policies grant necessary permissions for common use cases so that you can avoid having to investigate which permissions are needed. For more information, see AWS Managed Policies in the IAM User Guide.

The following AWS managed policies, which you can attach to users in your account, are specific to Amazon SageMaker:

- **AmazonSageMakerReadOnly** – Grants read-only access to Amazon SageMaker resources.
- **AmazonSageMakerFullAccess** – Grants full access to Amazon SageMaker resources and the supported operations. (This does not provide unrestricted S3 access, but supports buckets/objects with specific sagemaker tags.)

The following AWS managed policies can also be attached to users in your account:

- **AdministratorAccess** – Grants all actions for all AWS services and for all resources in the account.
- **DataScientist** – Grants a wide range of permissions to cover most of the use cases (primarily for analytics and business intelligence) encountered by data scientists.

You can review these permissions policies by signing in to the IAM console and searching for them.

You can also create your own custom IAM policies to allow permissions for Amazon SageMaker actions and resources as you need them. You can attach these custom policies to the IAM users or groups that require them.

### Amazon SageMaker API Permissions: Actions, Permissions, and Resources Reference

When you are setting up access control and writing a permissions policy that you can attach to an IAM identity (an identity-based policy), use the following as a reference. The each Amazon SageMaker API operation, the corresponding actions for which you can grant permissions to perform the action, and the AWS resource for which you can grant the permissions. You specify the actions in the policy's `Action` field, and you specify the resource value in the policy's `Resource` field.

**Note**

Except for the `ListTags` API, resource-level restrictions are not available on `List-` calls. Any user calling a `List-` API will see all resources of that type in the account.

To express conditions in your Amazon SageMaker policies, you can use AWS-wide condition keys. For a complete list of AWS-wide keys, see Available Keys in the IAM User Guide.
### Amazon SageMaker API Operations and Required Permissions for Actions

<table>
<thead>
<tr>
<th>Amazon SageMaker API Operations</th>
<th>Required Permissions (API Actions)</th>
<th>Resources</th>
</tr>
</thead>
<tbody>
<tr>
<td>AddTags</td>
<td>sagemaker:AddTags</td>
<td>arn:aws:sagemaker:region:account-id:*</td>
</tr>
<tr>
<td>CreateDomain</td>
<td>sagemaker:CreateDomain</td>
<td>arn:aws:sagemaker:region:account-id:domain/domain-id</td>
</tr>
<tr>
<td></td>
<td>iam:CreateServiceLinkedRole</td>
<td></td>
</tr>
<tr>
<td></td>
<td>iam:PassRole</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Required if a KMS customer managed CMK is specified for KmsKeyId:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>efs:CreateFileSystem</td>
<td></td>
</tr>
<tr>
<td></td>
<td>kms:CreateGrant</td>
<td></td>
</tr>
<tr>
<td></td>
<td>kms:Decrypt</td>
<td></td>
</tr>
<tr>
<td></td>
<td>kms:DescribeKey</td>
<td></td>
</tr>
<tr>
<td></td>
<td>kms:GenerateDataKeyWithoutPlainText</td>
<td></td>
</tr>
<tr>
<td>CreateEndpoint</td>
<td>sagemaker:CreateEndpoint</td>
<td>arn:aws:sagemaker:region:account-id:endpoint/endpointName</td>
</tr>
<tr>
<td></td>
<td>kms:CreateGrant (required only if the associated EndpointConfig has a KmsKeyId specified)</td>
<td>arn:aws:sagemaker:region:account-id:endpoint-config/endpointConfigName</td>
</tr>
<tr>
<td>CreateEndpointConfig</td>
<td>sagemaker:CreateEndpointConfig</td>
<td>arn:aws:sagemaker:region:account-id:endpoint-config/endpointConfigName</td>
</tr>
<tr>
<td>CreateFlowDefinition</td>
<td>sagemaker:CreateFlowDefinition</td>
<td>arn:aws:sagemaker:region:account-id:flow-definition/flowDefinitionName</td>
</tr>
<tr>
<td></td>
<td>iam:PassRole</td>
<td></td>
</tr>
<tr>
<td>CreateHumanTaskUi</td>
<td>sagemaker:CreateHumanTaskUi</td>
<td>arn:aws:sagemaker:region:account-id:human-task-ui/humanTaskUiName</td>
</tr>
<tr>
<td>CreateHyperParameterTuningJob</td>
<td>sagemaker:CreateHyperParameterTuningJob</td>
<td>arn:aws:sagemaker:region:account-id:hyper-parameter-tuning-job/hyperParameterTuningJobName</td>
</tr>
<tr>
<td></td>
<td>iam:PassRole</td>
<td></td>
</tr>
<tr>
<td>CreateImage</td>
<td>sagemaker:CreateImage</td>
<td>arn:aws:sagemaker:region:account-id:image/*</td>
</tr>
<tr>
<td></td>
<td>iam:PassRole</td>
<td></td>
</tr>
<tr>
<td>Amazon SageMaker API Operations</td>
<td>Required Permissions (API Actions)</td>
<td>Resources</td>
</tr>
<tr>
<td>---------------------------------</td>
<td>-----------------------------------</td>
<td>-----------</td>
</tr>
<tr>
<td></td>
<td>iam:PassRole</td>
<td></td>
</tr>
<tr>
<td>CreateModel</td>
<td>sagemaker:CreateModel</td>
<td>arn:aws:sagemaker:region:account-id:model/modelName</td>
</tr>
<tr>
<td></td>
<td>iam:PassRole</td>
<td></td>
</tr>
<tr>
<td>CreateNotebookInstance</td>
<td>sagemaker:CreateNotebookInstance</td>
<td>arn:aws:sagemaker:region:account-id:notebook-instance/notebookInstanceName</td>
</tr>
<tr>
<td></td>
<td>iam:PassRole</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ec2:CreateNetworkInterface</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ec2:DescribeSecurityGroups</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ec2:DescribeSubnets</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ec2:DescribeVpcs</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ec2:DescribeVpcEndpoints</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>kms:DescribeKey</td>
<td></td>
</tr>
<tr>
<td></td>
<td>kms:CreateGrant</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CreatePresignedNotebookInstance</td>
<td>sagemaker:CreatePresignedNotebookInstance</td>
<td>arn:aws:sagemaker:region:account-id:notebook-instance/notebookInstanceName</td>
</tr>
<tr>
<td>Amazon SageMaker API Operations</td>
<td>Required Permissions (API Actions)</td>
<td>Resources</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>--------------------------------------------------------------------------------------------------</td>
<td>---------------------------------------------------------------------------</td>
</tr>
<tr>
<td>CreateProcessingJob</td>
<td>sagemaker:CreateProcessingJob</td>
<td>arn:aws:sagemaker:region:accountId:processing-job/processingJobName</td>
</tr>
<tr>
<td></td>
<td>iam:PassRole</td>
<td></td>
</tr>
<tr>
<td></td>
<td>kms:CreateGrant (required only if the associated ProcessingResources has a specified VolumeKmsKeyId and the associated role does not have a policy that permits this action)</td>
<td></td>
</tr>
<tr>
<td>CreateTrainingJob</td>
<td>sagemaker:CreateTrainingJob</td>
<td>arn:aws:sagemaker:region:accountId:training-job/trainingJobName</td>
</tr>
<tr>
<td></td>
<td>iam:PassRole</td>
<td></td>
</tr>
<tr>
<td></td>
<td>kms:CreateGrant (required only if the associated ResourceConfig has a specified VolumeKmsKeyId and the associated role does not have a policy that permits this action)</td>
<td></td>
</tr>
<tr>
<td>CreateTransformJob</td>
<td>sagemaker:CreateTransformJob</td>
<td>arn:aws:sagemaker:region:accountId:transform-job/transformJobName</td>
</tr>
<tr>
<td></td>
<td>kms:CreateGrant (required only if the associated TransformResources has a specified VolumeKmsKeyId and the associated role does not have a policy that permits this action)</td>
<td></td>
</tr>
<tr>
<td>CreateWorkteam</td>
<td>sagemaker:CreateWorkteam</td>
<td>arn:aws:sagemaker:region:accountId:workteam/private-crowd/work team name</td>
</tr>
<tr>
<td></td>
<td>cognito-idp:DescribeUserPoolClient</td>
<td></td>
</tr>
<tr>
<td></td>
<td>cognito-idp:UpdateUserPool</td>
<td></td>
</tr>
<tr>
<td></td>
<td>cognito-idp:DescribeUserPool</td>
<td></td>
</tr>
<tr>
<td></td>
<td>cognito-idp:UpdateUserPoolClient</td>
<td></td>
</tr>
<tr>
<td>DeleteEndpoint</td>
<td>sagemaker:DeleteEndpoint</td>
<td>arn:aws:sagemaker:region:accountId:endpoint/endpointName</td>
</tr>
<tr>
<td>DeleteEndpointConfig</td>
<td>sagemaker:DeleteEndpointConfig</td>
<td>arn:aws:sagemaker:region:accountId:endpoint-config/endpointConfigName</td>
</tr>
<tr>
<td>DeleteFlowDefinition</td>
<td>sagemaker:DeleteFlowDefinition</td>
<td>arn:aws:sagemaker:region:accountId:flow-definition/flowDefinitionName</td>
</tr>
<tr>
<td>Amazon SageMaker API Operations</td>
<td>Required Permissions (API Actions)</td>
<td>Resources</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>----------------------------------</td>
<td>-----------</td>
</tr>
<tr>
<td>DeleteHumanLoop</td>
<td>sagemaker:DeleteHumanLoop</td>
<td>arn:aws:sagemaker:region:accountId:human-loop/humanLoopName</td>
</tr>
<tr>
<td>DeleteImage</td>
<td>sagemaker:DeleteImage</td>
<td>arn:aws:sagemaker:region:accountId:image/imageName</td>
</tr>
<tr>
<td>DeleteModel</td>
<td>sagemaker:DeleteModel</td>
<td>arn:aws:sagemaker:region:accountId:model/modelName</td>
</tr>
<tr>
<td>DeleteNotebookInstance</td>
<td>sagemaker:DeleteNotebookInstance</td>
<td>arn:aws:sagemaker:region:accountId:notebook-instance/notebookInstanceName</td>
</tr>
<tr>
<td>DeleteTags</td>
<td>sagemaker:DeleteTags</td>
<td>arn:aws:sagemaker:region:accountId:*</td>
</tr>
<tr>
<td>DeleteWorkteam</td>
<td>sagemaker:DeleteWorkteam</td>
<td>arn:aws:sagemaker:region:accountId:workteam/*</td>
</tr>
<tr>
<td>DescribeEndpoint</td>
<td>sagemaker:DescribeEndpoint</td>
<td>arn:aws:sagemaker:region:accountId:endpoint/endpointName</td>
</tr>
<tr>
<td>DescribeEndpointConfig</td>
<td>sagemaker:DescribeEndpointConfig</td>
<td>arn:aws:sagemaker:region:accountId:endpoint-config/endpointConfigName</td>
</tr>
<tr>
<td>DescribeFlowDefinition</td>
<td>sagemaker:DescribeFlowDefinition</td>
<td>arn:aws:sagemaker:region:accountId:flow-definition/flowDefinitionName</td>
</tr>
<tr>
<td>DescribeHumanLoop</td>
<td>sagemaker:DescribeHumanLoop</td>
<td>arn:aws:sagemaker:region:accountId:human-loop/humanLoopName</td>
</tr>
<tr>
<td>DescribeHumanTaskUi</td>
<td>sagemaker:DescribeHumanTaskUi</td>
<td>arn:aws:sagemaker:region:accountId:human-task-ui/humanTaskUiName</td>
</tr>
<tr>
<td>DescribeHyperParameterTuningJob</td>
<td>sagemaker:DescribeHyperParameterTuningJob</td>
<td>arn:aws:sagemaker:region:accountId:hyper-parameter-tuning-job/hyperParameterTuningJob</td>
</tr>
<tr>
<td>Amazon SageMaker API Operations</td>
<td>Required Permissions (API Actions)</td>
<td>Resources</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>-----------------------------------</td>
<td>-----------</td>
</tr>
<tr>
<td>DescribeImage</td>
<td>sagemaker:DescribeImage</td>
<td>arn:aws:sagemaker:region:account-id:imageName</td>
</tr>
<tr>
<td>DescribeModel</td>
<td>sagemaker:DescribeModel</td>
<td>arn:aws:sagemaker:region:account-id:modelName</td>
</tr>
<tr>
<td>DescribeNotebookInstance</td>
<td>sagemaker:DescribeNotebookInstance</td>
<td>arn:aws:sagemaker:region:account-id:notebook-instance/notebookInstanceName</td>
</tr>
<tr>
<td>DescribeSubscribedWorkteam</td>
<td>sagemaker:DescribeSubscribedWorkteam</td>
<td>arn:aws:marketplace:ViewSubscriptions</td>
</tr>
<tr>
<td>DescribeWorkteam</td>
<td>sagemaker:DescribeWorkteam</td>
<td>arn:aws:sagemaker:region:account-id:workteam/*</td>
</tr>
<tr>
<td>runtime_InvokeEndpoint</td>
<td>sagemaker:InvokeEndpoint</td>
<td>arn:aws:sagemaker:region:account-id:endpointName</td>
</tr>
<tr>
<td>ListEndpointConfigs</td>
<td>sagemaker:ListEndpointConfigs</td>
<td>*</td>
</tr>
<tr>
<td>ListEndpoints</td>
<td>sagemaker:ListEndpoints</td>
<td>*</td>
</tr>
<tr>
<td>ListFlowDefinitions</td>
<td>sagemaker:ListFlowDefinitions</td>
<td>*</td>
</tr>
<tr>
<td>ListHumanLoops</td>
<td>sagemaker:ListHumanLoops</td>
<td>*</td>
</tr>
<tr>
<td>ListHumanTaskUis</td>
<td>sagemaker:ListHumanTaskUis</td>
<td>*</td>
</tr>
<tr>
<td>ListImages</td>
<td>sagemaker:ListImages</td>
<td>*</td>
</tr>
<tr>
<td>Amazon SageMaker API Operations</td>
<td>Required Permissions (API Actions)</td>
<td>Resources</td>
</tr>
<tr>
<td>---------------------------------</td>
<td>------------------------------------</td>
<td>-----------</td>
</tr>
<tr>
<td>ListLabelingJobs</td>
<td>sagemaker:ListLabelingJobs</td>
<td>*</td>
</tr>
<tr>
<td>ListLabelingJobsForWorkteam</td>
<td>sagemaker:ListLabelingJobForWorkteam</td>
<td></td>
</tr>
<tr>
<td>ListModels</td>
<td>sagemaker:ListModels</td>
<td>*</td>
</tr>
<tr>
<td>ListNotebookInstances</td>
<td>sagemaker:ListNotebookInstances</td>
<td></td>
</tr>
<tr>
<td>ListProcessingJobs</td>
<td>sagemaker:ListProcessingJobs</td>
<td>*</td>
</tr>
<tr>
<td></td>
<td>aws-marketplace:ViewSubscriptions</td>
<td></td>
</tr>
<tr>
<td>ListTags</td>
<td>sagemaker:ListTags</td>
<td>arn:aws:sagemaker:region:account-id:*</td>
</tr>
<tr>
<td>ListTrainingJobs</td>
<td>sagemaker:ListTrainingJobs</td>
<td>*</td>
</tr>
<tr>
<td>ListTransformJobs</td>
<td>sagemaker:ListTransformJobs</td>
<td>*</td>
</tr>
<tr>
<td>StartHumanLoop</td>
<td>sagemaker:StartHumanLoop</td>
<td>arn:aws:sagemaker:region:account-id:human-loop/humanLoopName</td>
</tr>
<tr>
<td>Amazon SageMaker API Operations</td>
<td>Required Permissions (API Actions)</td>
<td>Resources</td>
</tr>
<tr>
<td>---------------------------------</td>
<td>-----------------------------------</td>
<td>-----------</td>
</tr>
</tbody>
</table>
| StartNotebookInstance          | sagemaker:StartNotebookInstance
  iam:PassRole                   | arn:aws:sagemaker:region:account-id:notebook-instance/notebookInstanceName |
|                                 | ec2:CreateNetworkInterface       |           |
|                                 | ec2:DescribeNetworkInterfaces    |           |
|                                 | ec2:DescribeSecurityGroups       |           |
|                                 | ec2:DescribeSubnets              |           |
|                                 | ec2:DescribeVpcs                 |           |
|                                 | The following permissions are     |           |
|                                 | required only if you specified a |           |
|                                 | VPC when you created your notebook instance: |           |
|                                 | ec2:DescribeVpcEndpoints         |           |
|                                 | The following permission is       |           |
|                                 | required only if you specify a VPC |           |
|                                 | and an elastic inference          |           |
|                                 | accelerator for your notebook     |           |
|                                 | instance:                        |           |
|                                 | ec2:DescribeVpcEndpoints         |           |
|                                 | The following permissions are     |           |
|                                 | required only if you specified an |           |
|                                 | encryption key when you created   |           |
|                                 | the notebook instance:            |           |
|                                 | kms:DescribeKey                  |           |
|                                 | kms:CreateGrant                  |           |
|                                 | The following permission is       |           |
|                                 | required only if you specified an |           |
|                                 | AWS Secrets Manager secret to     |           |
|                                 | access a private Git repository   |           |
|                                 | when you created the notebook     |           |
|                                 | instance:                        |           |
|                                 | secretsmanager:GetSecretValue     |           |
| StopHumanLoop                   | sagemaker:StopHumanLoop          | arn:aws:sagemaker:region:account-id:human-loop/humanLoopName |
| StopHyperParameterTuningJob     | sagemaker:StopHyperParameterTuningJob | arn:aws:sagemaker:region:account-id:hyper-parameter-tuning-job/hyperParameterTuningJob |
| StopNotebookInstance            | sagemaker:StopNotebookInstance   | arn:aws:sagemaker:region:account-id:notebook-instance/notebookInstanceName |
### Amazon SageMaker API and Required Permissions for Actions

**API Operation:** **AddTags**

Required Permissions (API Action): `sagemaker:AddTags`

Resources: *

**API Operation:** **CreateEndpoint**

Required Permissions (API Action): `sagemaker:CreateEndpoint`

Resources: `arn:aws:sagemaker:region:account-id:endpoint/endpointName`

**API Operation:** **CreateEndpointConfig**

Required Permissions (API Action): `sagemaker:CreateEndpointConfig`

Resources: `arn:aws:sagemaker:region:account-id:endpoint-config/endpointConfigName`

**API Operation:** **CreateModel**

Required Permissions (API Action): `sagemaker:CreateModel`, `iam:PassRole`

Resources: `arn:aws:sagemaker:region:account-id:model/modelName`

**API Operation:** **CreateLabelingJob**

Required Permissions (API Action): `sagemaker:CreateLabelingJob`, `iam:PassRole`

Resources: `arn:aws:sagemaker:region:account-id:labelingJob/labelingJobName`
Resources: arn:aws:sagemaker:region:account-id:labeling-job/labelingJobName

API Operation: CreateNotebookInstance


Resources: arn:aws:sagemaker:region:account-id:notebook-instance/notebookInstanceName

API Operation: CreateTrainingJob

Required Permissions (API Action): sagemaker:CreateTrainingJob, iam:PassRole

Resources: arn:aws:sagemaker:region:account-id:training-job/trainingJobName

API Operation: CreateWorkteam


API Operation: DeleteEndpoint

Required Permissions (API Action): sagemaker:DeleteEndpoint

Resources: arn:aws:sagemaker:region:account-id:endpoint/endpointName

API Operation: DeleteEndpointConfig

Required Permissions (API Action): sagemaker:DeleteEndpointConfig

Resources: arn:aws:sagemaker:region:account-id:endpoint-config/endpointConfigName

API Operation: DeleteModel

Required Permissions (API Action): sagemaker:DeleteModel

Resources: arn:aws:sagemaker:region:account-id:model/modelName

API Operation: DeleteNotebookInstance


Resources: arn:aws:sagemaker:region:account-id:notebook-instance/notebookInstanceName

API Operation: DeleteTags

Required Permissions (API Action): sagemaker:DeleteTags

Resources: *

1672
API Operation: **DeleteWorkteam**

Required Permissions (API Action): sagemaker:DeleteWorkteam


API Operation: **DescribeEndpoint**

Required Permissions (API Action): sagemaker:DescribeEndpoint

Resources: arn:aws:sagemaker:region:account-id:endpoint/endpointName

API Operation: **DescribeEndpointConfig**

Required Permissions (API Action): sagemaker:DescribeEndpointConfig

Resources: arn:aws:sagemaker:region:account-id:endpoint-config/endpointConfigName

API Operation: **DescribeLabelingJob**

Required Permissions (API Action): sagemaker:DescribeLabelingJob

Resources: arn:aws:sagemaker:region:account-id:labeling-job/labelingJobName

API Operation: **DescribeModel**

Required Permissions (API Action): sagemaker:DescribeModel

Resources: arn:aws:sagemaker:region:account-id:model/modelName

API Operation: **DescribeNotebookInstance**

Required Permissions (API Action): sagemaker:DescribeNotebookInstance

Resources: arn:aws:sagemaker:region:account-id:notebook-instance/notebookInstanceName

API Operation: **DescribeSubscribedWorkteam**

Required Permissions (API Action): sagemaker:DescribeSubscribedWorkteam, aws-marketplace:ViewSubscriptions


API Operation: **DescribeTrainingJob**

Required Permissions (API Action): sagemaker:DescribeTrainingJob

Resources: arn:aws:sagemaker:region:account-id:training-job/trainingJobName

API Operation: **DescribeWorkteam**

Required Permissions (API Action): sagemaker:DescribeWorkteam


API Operation: **CreatePresignedNotebookInstanceUrl**

Required Permissions (API Action): sagemaker:CreatePresignedNotebookInstanceUrl

Resources: arn:aws:sagemaker:region:account-id:notebook-instance/notebookInstanceName

API Operation: **runtime_InvokeEndpoint**

Required Permissions (API Action): sagemaker:InvokeEndpoint

Resources: arn:aws:sagemaker:region:account-id:endpoint/endpointName
API Operation: `ListEndpointConfigs`

Required Permissions (API Action): `sagemaker:ListEndpointConfigs`

Resources: *

API Operation: `ListEndpoints`

Required Permissions (API Action): `sagemaker:ListEndpoints`

Resources: *

API Operation: `ListLabelingJobs`

Required Permissions (API Action): `sagemaker:ListLabelingJobs`

Resources: *

API Operation: `ListLabelingJobsForWorkteam`

Required Permissions (API Action): `sagemaker:ListLabelingJobsForWorkteam`

Resources: *

API Operation: `ListModels`

Required Permissions (API Action): `sagemaker:ListModels`

Resources: *

API Operation: `ListNotebookInstances`

Required Permissions (API Action): `sagemaker:ListNotebookInstances`

Resources: *

API Operation: `ListSubscribedWorkteams`


API Operation: `ListTags`

Required Permissions (API Action): `sagemaker:ListTags`

Resources: *

API Operation: `ListTrainingJobs`

Required Permissions (API Action): `sagemaker:ListTrainingJobs`

Resources: *

API Operation: `ListWorkteams`

Required Permissions (API Action): `sagemaker:ListWorkteams`


API Operation: `StartNotebookInstance`

Troubleshooting Amazon SageMaker Identity and Access

Use the following information to help you diagnose and fix common issues that you might encounter when working with SageMaker and IAM.

Topics
- I Am Not Authorized to Perform an Action in SageMaker (p. 1675)
- I Am Not Authorized to Perform iam:PassRole (p. 1676)
- I Want to View My Access Keys (p. 1676)
- I'm an Administrator and Want to Allow Others to Access SageMaker (p. 1676)
- I Want to Allow People Outside of My AWS Account to Access My SageMaker Resources (p. 1677)

I Am Not Authorized to Perform an Action in SageMaker

If the AWS Management Console tells you that you're not authorized to perform an action, then you must contact your administrator for assistance. Your administrator is the person that provided you with your user name and password.
The following example error occurs when the mateojackson IAM user tries to use the console to view details about a training job but does not have sagemaker:sagemaker:DescribeTrainingJob permissions.

```
User: arn:aws:iam::123456789012:user/mateojackson is not authorized to perform: sagemaker:DescribeTrainingJob on resource: my-example-widget
```

In this case, Mateo asks his administrator to update his policies to allow him to access the TrainingJob resource using the sagemaker:DescribeTrainingJob action.

### I Am Not Authorized to Perform iam:PassRole

If you receive an error that you're not authorized to perform the iam:PassRole action, then you must contact your administrator for assistance. Your administrator is the person that provided you with your user name and password. Ask that person to update your policies to allow you to pass a role to SageMaker.

Some AWS services allow you to pass an existing role to that service, instead of creating a new service role or service-linked role. To do this, you must have permissions to pass the role to the service.

The following example error occurs when an IAM user named marymajor tries to use the console to perform an action in SageMaker. However, the action requires the service to have permissions granted by a service role. Mary does not have permissions to pass the role to the service.

```
User: arn:aws:iam::123456789012:user/marymajor is not authorized to perform: iam:PassRole
```

In this case, Mary asks her administrator to update her policies to allow her to perform the iam:PassRole action.

### I Want to View My Access Keys

After you create your IAM user access keys, you can view your access key ID at any time. However, you can't view your secret access key again. If you lose your secret key, you must create a new access key pair.

Access keys consist of two parts: an access key ID (for example, AKIAIOSFODNN7EXAMPLE) and a secret access key (for example, wJalrXUtnFEMI/K7MDEng/hPsxrYtECi6YXIn/4k养example). Like a user name and password, you must use both the access key ID and secret access key together to authenticate your requests. Manage your access keys as securely as you do your user name and password.

**Important**

Do not provide your access keys to a third party, even to help find your canonical user ID. By doing this, you might give someone permanent access to your account.

When you create an access key pair, you are prompted to save the access key ID and secret access key in a secure location. The secret access key is available only at the time you create it. If you lose your secret access key, you must add new access keys to your IAM user. You can have a maximum of two access keys. If you already have two, you must delete one key pair before creating a new one. To view instructions, see Managing access keys in the IAM User Guide.

### I'm an Administrator and Want to Allow Others to Access SageMaker

To allow others to access SageMaker, you must create an IAM entity (user or role) for the person or application that needs access. They will use the credentials for that entity to access AWS. You must then attach a policy to the entity that grants them the correct permissions in SageMaker.

To get started right away, see Creating your first IAM delegated user and group in the IAM User Guide.
I Want to Allow People Outside of My AWS Account to Access My SageMaker Resources

You can create a role that users in other accounts or people outside of your organization can use to access your resources. You can specify who is trusted to assume the role. For services that support resource-based policies or access control lists (ACLs), you can use those policies to grant people access to your resources.

To learn more, consult the following:

- To learn whether SageMaker supports these features, see How Amazon SageMaker Works with IAM (p. 1622).
- To learn how to provide access to your resources across AWS accounts that you own, see Providing access to an IAM user in another AWS account that you own in the IAM User Guide.
- To learn how to provide access to your resources to third-party AWS accounts, see Providing access to AWS accounts owned by third parties in the IAM User Guide.
- To learn how to provide access through identity federation, see Providing access to externally authenticated users (identity federation) in the IAM User Guide.
- To learn the difference between using roles and resource-based policies for cross-account access, see How IAM roles differ from resource-based policies in the IAM User Guide.

Logging and Monitoring

You can monitor Amazon SageMaker using Amazon CloudWatch, which collects raw data and processes it into readable, near real-time metrics. These statistics are kept for 15 months, so that you can access historical information and gain a better perspective on how your web application or service is performing. You can also set alarms that watch for certain thresholds and send notifications or take actions when those thresholds are met. For more information, see Monitor Amazon SageMaker with Amazon CloudWatch (p. 1704).

Amazon CloudWatch Logs enables you to monitor, store, and access your log files from Amazon EC2 instances, AWS CloudTrail, and other sources. You can collect and track metrics, create customized dashboards, and set alarms that notify you or take actions when a specified metric reaches a threshold that you specify. CloudWatch Logs can monitor information in the log files and notify you when certain thresholds are met. You can also archive your log data in highly durable storage. For more information, see Log Amazon SageMaker Events with Amazon CloudWatch (p. 1712).

AWS CloudTrail provides a record of actions taken by a user, role, or an AWS service in SageMaker. Using the information collected by CloudTrail, you can determine the request that was made to SageMaker, the IP address from which the request was made, who made the request, when it was made, and additional details. For more information, Log Amazon SageMaker API Calls with AWS CloudTrail (p. 1713).

Note
CloudTrail does not monitor calls to runtime_InvokeEndpoint.

You can create rules in Amazon CloudWatch Events to react to status changes in status in a SageMaker training, hyperparameter tuning, or batch transform job. For more information, see Automating Amazon SageMaker with Amazon EventBridge (p. 1715).

Compliance Validation for Amazon SageMaker

Third-party auditors assess the security and compliance of Amazon SageMaker as part of multiple AWS compliance programs. These include SOC, PCI, FedRAMP, HIPAA, and others.
Resilience in Amazon SageMaker

The AWS global infrastructure is built around AWS Regions and Availability Zones. AWS Regions provide multiple physically separated and isolated Availability Zones, which are connected with low-latency, high-throughput, and highly redundant networking. With Availability Zones, you can design and operate applications and databases that automatically fail over between Availability Zones without interruption. Availability Zones are more highly available, fault tolerant, and scalable than traditional single or multiple data center infrastructures.

For more information about AWS Regions and Availability Zones, see AWS Global Infrastructure.

In addition to the AWS global infrastructure, Amazon SageMaker offers several features to help support your data resiliency and backup needs.

Infrastructure Security in Amazon SageMaker

As a managed service, Amazon SageMaker is protected by the AWS global network security procedures that are described in the Amazon Web Services: Overview of Security Processes whitepaper.

You use AWS published API calls to access Amazon SageMaker through the network. Clients must support Transport Layer Security (TLS) 1.0 or later. We recommend TLS 1.2 or later. Clients must also support cipher suites with perfect forward secrecy (PFS) such as Ephemeral Diffie-Hellman (DHE) or Elliptic Curve Ephemeral Diffie-Hellman (ECDHE). Most modern systems such as Java 7 and later support these modes.

Additionally, requests must be signed by using an access key ID and a secret access key that is associated with an IAM principal. Or you can use the AWS Security Token Service (AWS STS) to generate temporary security credentials to sign requests.

Topics
Connect SageMaker Studio Notebooks to Resources in a VPC

Amazon SageMaker Studio allows direct internet access by default. This could provide an avenue for unauthorized access to your data. For example, if you install malicious code on your computer (in the form of a publicly available notebook or a publicly available source code library), it could access your data. You can choose to restrict which traffic can access the internet by launching Studio in a Virtual Private Cloud (VPC) of your choosing. This allows you fine-grained control of the network access and internet connectivity of your SageMaker Studio notebooks. You can disable direct internet access to add an additional layer of security.

By default, SageMaker Studio provides a network interface that allows communication with the internet through a VPC managed by SageMaker. Traffic to AWS services like Amazon S3 and CloudWatch goes through an internet gateway as does traffic that accesses the SageMaker API and SageMaker runtime. Traffic between the domain and your Amazon EFS volume goes through the VPC that you specified when you onboarded to Studio or called the CreateDomain API. The following diagram shows the default configuration.

To disable direct internet access, you can specify the VPC only network access type when you onboard to Studio or call the CreateDomain API. Doing so prevents SageMaker from providing internet access to your Studio notebooks. As a result, you won't be able to run a Studio notebook unless your VPC has an interface endpoint to the SageMaker API and runtime, or a NAT gateway, and your security groups allow outbound connections. The following diagram shows a configuration for using VPC-only mode.

---

To disable direct internet access, you can specify the VPC only network access type when you onboard to Studio or call the CreateDomain API. Doing so prevents SageMaker from providing internet access to your Studio notebooks. As a result, you won't be able to run a Studio notebook unless your VPC has an interface endpoint to the SageMaker API and runtime, or a NAT gateway, and your security groups allow outbound connections. The following diagram shows a configuration for using VPC-only mode.
VPC requirements in VpcOnly

When you choose VpcOnly, your VPC must contain the following:

- Subnets with one IP address for each instance. For more information, see VPC and subnet sizing for IPv4.

  **Note**
  
  You can configure only subnets with a default tenancy VPC in which your instance runs on shared hardware. For more information on the tenancy attribute for VPCs, see Dedicated Instances.

- One or more security groups with inbound and outbound rules that together allow the following traffic:
  - NFS traffic over TCP on port 2049 between the domain and the Amazon EFS volume.
  - TCP traffic within the security group. This is required for connectivity between the JupyterServer app and the KernelGateway apps.

- If you want to allow internet access, you must attach an internet gateway or NAT gateway.

- If you don't want to allow internet access, you must create interface VPC endpoints (AWS PrivateLink) to access the following:
  - The SageMaker API and SageMaker runtime. This is required to run Studio notebooks and to train and host models.
  - Amazon S3 and other AWS services you require.

  You must associate the security groups for your VPC with these endpoints.

For more information

- Securing Amazon SageMaker Studio connectivity using a private VPC.
- Security groups for your VPC
- Connect to SageMaker Through a VPC Interface Endpoint (p. 1683)
- VPC with public and private subnets (NAT)
Connect a Notebook Instance to Resources in a VPC

SageMaker notebook instances are internet-enabled by default. This allows you to download popular packages and notebooks, customize your development environment, and work efficiently. However, this could provide an additional avenue for unauthorized access to your data. For example, a malicious user or code that you accidentally install on the computer (in the form of a publicly available notebook or a publicly available source code library) could access your data. You can choose to launch your notebook instance in your Virtual Private Cloud (VPC) to restrict which traffic can go through the public Internet. When launched with your VPC attached, the notebook instance can either be configured with or without direct internet access.

When your notebook allows **direct internet access**, SageMaker provides a network interface that allows the notebook to communicate with the internet through a VPC managed by SageMaker. Traffic within your VPC's CIDR goes through elastic network interface created in your VPC. All the other traffic goes through the network interface created by SageMaker, which is essentially through the public internet. Traffic to gateway VPC endpoints like Amazon S3 and DynamoDB goes through the public internet, while traffic to interface VPC endpoints (interface endpoints) still goes through your VPC. If you want to use gateway VPC endpoints, you might want to disable direct internet access.

To disable direct internet access, you can specify a VPC for your notebook instance. By doing so, you prevent SageMaker from providing internet access to your notebook instance. As a result, the notebook instance won't be able to train or host models unless your VPC has an interface endpoint (PrivateLink) or a NAT gateway, and your security groups allow outbound connections.

For information about creating a VPC interface endpoint to use AWS PrivateLink for your notebook instance, see [Connect to a Notebook Instance Through a VPC Interface Endpoint](p. 1684). For information about setting up a NAT gateway for your VPC, see [VPC with Public and Private Subnets (NAT)](p. 1684) in the Amazon Virtual Private Cloud User Guide. For information about security groups, see [Security Groups for Your VPC](p. 1684). For more information about networking configurations in each networking mode and configuring network on premise, see [Understanding Amazon SageMaker notebook instance networking configurations and advanced routing options](p. 1684).

**Security and Shared Notebook Instances**

A SageMaker notebook instance is designed to work best for an individual user. It is designed to give data scientists and other users the most power for managing their development environment.

A notebook instance user has root access for installing packages and other pertinent software. We recommend that you exercise judgement when granting individuals access to notebook instances that are attached to a VPC that contains sensitive information. For example, you might grant a user access to a notebook instance with an IAM policy, as shown in the following example:

```json
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": "sagemaker:CreatePresignedNotebookInstanceUrl",
    }
  ]
}
```
Training and Inference Containers Run in Internet-Free Mode

SageMaker training and deployed inference containers are internet-enabled by default. This allows containers to access external services and resources on the public internet as part of your training and inference workloads. However, this could provide an avenue for unauthorized access to your data. For example, a malicious user or code that you accidentally install on the container (in the form of a publicly available source code library) could access your data and transfer it to a remote host.

If you use an Amazon VPC by specifying a value for the `VpcConfig` parameter when you call `CreateTrainingJob`, `CreateHyperParameterTuningJob`, or `CreateModel`, you can protect your data and resources by managing security groups and restricting internet access from your VPC. However, this comes at the cost of additional network configuration, and has the risk of configuring your network incorrectly. If you do not want SageMaker to provide external network access to your training or inference containers, you can enable network isolation when you create your training job or model by setting the value of the `EnableNetworkIsolation` parameter to `True` when you call `CreateTrainingJob`, `CreateHyperParameterTuningJob`, or `CreateModel`.

If you enable network isolation, the containers can't make any outbound network calls, even to other AWS services such as Amazon S3. Additionally, no AWS credentials are made available to the container runtime environment. In the case of a training job with multiple instances, network inbound and outbound traffic is limited to the peers of each training container. SageMaker still performs download and upload operations against Amazon S3 using your SageMaker execution role in isolation from the training or inference container.

Network isolation is required for training jobs and models run using resources from AWS Marketplace. Network isolation can be used in conjunction with a VPC. In this scenario, download and upload of customer data and model artifacts are routed via your VPC subnet. However, the training and inference containers themselves continue to be isolated from the network, and do not have access to any resource within your VPC or on the internet.

Network isolation is not supported by the following managed SageMaker containers as they require access to Amazon S3:

- Chainer
- PyTorch
- Scikit-learn
- SageMaker Reinforcement Learning

SageMaker Scans AWS Marketplace Training and Inference Containers for Security Vulnerabilities

To meet our security requirements, algorithms and model packages listed in AWS Marketplace are scanned for Common Vulnerabilities and Exposures (CVE). CVE is a list of publicly known information about security vulnerability and exposure. The National Vulnerability Database (NVD) provides CVE details such as severity, impact rating, and fix information. Both CVE and NVD are available for public consumption and free for security tools and services to use. For more information, see http://cve.mitre.org/about/faqs.html#what_is_cve.
Connect to SageMaker Through a VPC Interface Endpoint

You can connect directly to the SageMaker API or to the SageMaker Runtime through an interface endpoint in your Virtual Private Cloud (VPC) instead of connecting over the internet. When you use a VPC interface endpoint, communication between your VPC and the SageMaker API or Runtime is conducted entirely and securely within the AWS network.

The SageMaker API and Runtime support Amazon Virtual Private Cloud (Amazon VPC) interface endpoints that are powered by AWS PrivateLink. Each VPC endpoint is represented by one or more Elastic Network Interfaces with private IP addresses in your VPC subnets.

The VPC interface endpoint connects your VPC directly to the SageMaker API or Runtime without an internet gateway, NAT device, VPN connection, or AWS Direct Connect connection. The instances in your VPC don’t need public IP addresses to communicate with the SageMaker API or Runtime.

You can create an interface endpoint to connect to SageMaker or to SageMaker Runtime with either the AWS console or AWS Command Line Interface (AWS CLI) commands. For instructions, see Creating an Interface Endpoint.

After you have created a VPC endpoint, you can use the following example CLI commands that use the endpoint-url parameter to specify interface endpoints to the SageMaker API or Runtime:

```sh
aws sagemaker list-notebook-instances --endpoint-url VPC_Endpoint_ID.api.sagemaker.Region.vpce.amazonaws.com
aws sagemaker list-training-jobs --endpoint-url VPC_Endpoint_ID.api.sagemaker.Region.vpce.amazonaws.com
aws sagemaker-runtime invoke-endpoint --endpoint-url VPC_Endpoint_ID.runtime.sagemaker.Region.vpce.amazonaws.com --endpoint-name Endpoint_Name --body "Endpoint_Body" --content-type "Content_Type" Output_File
```

If you enable private DNS hostnames for your VPC endpoint, you don’t need to specify the endpoint URL. The SageMaker API DNS hostname that the CLI and SageMaker SDK use by default (https://api.sagemaker.Region.amazonaws.com) resolves to your VPC endpoint. Similarly, the SageMaker Runtime DNS hostname that the CLI and SageMaker Runtime SDK use by default (https://runtime.sagemaker.Region.amazonaws.com) resolves to your VPC endpoint.

The SageMaker API and Runtime support VPC endpoints in all AWS Regions where both Amazon VPC and SageMaker are available. SageMaker supports making calls to all of its Operations inside your VPC. The result AuthorizedUrl from the CreatePresignedNotebookInstanceUrl is not supported by Private Link. For information about how to enable PrivateLink for the authorized URL that users use to connect to a notebook instance, see Connect to a Notebook Instance Through a VPC Interface Endpoint (p. 1684).

To learn more about AWS PrivateLink, see the AWS PrivateLink documentation. Refer to VPC Pricing for the price of VPC Endpoints. To learn more about VPC and Endpoints, see Amazon VPC. For information about how to use identity-based AWS Identity and Access Management policies to restrict access to the SageMaker API and runtime, see Control Access to the SageMaker API by Using Identity-based Policies (p. 1642).

Create a VPC Endpoint Policy for SageMaker

You can create a policy for Amazon VPC endpoints for SageMaker to specify the following:
Connect to SageMaker Through a VPC Interface Endpoint

- The principal that can perform actions.
- The actions that can be performed.
- The resources on which actions can be performed.

For more information, see Controlling Access to Services with VPC Endpoints in the Amazon VPC User Guide.

Note
VPC endpoint policies aren't supported for Federal Information Processing Standard (FIPS) SageMaker runtime endpoints for `runtime_InvokeEndpoint`.

The following example VPC endpoint policy specifies that all users who have access to the VPC interface endpoint are allowed to invoke the SageMaker hosted endpoint named `myEndpoint`.

```
{
  "Statement": [
    {
      "Action": "sagemaker:InvokeEndpoint",
      "Effect": "Allow",
      "Principal": "*"
    }
  ]
}
```

In this example, the following are denied:
- Other SageMaker API actions, such as `sagemaker:CreateEndpoint` and `sagemaker:CreateTrainingJob`.
- Invoking SageMaker hosted endpoints other than `myEndpoint`.

Note
In this example, users can still take other SageMaker API actions from outside the VPC. For information about how to restrict API calls to those from within the VPC, see Control Access to the SageMaker API by Using Identity-based Policies (p. 1642).

Connect to a Notebook Instance Through a VPC Interface Endpoint

You can connect to your notebook instance from your VPC through an interface endpoint in your Virtual Private Cloud (VPC) instead of connecting over the internet. When you use a VPC interface endpoint, communication between your VPC and the notebook instance is conducted entirely and securely within the AWS network.

SageMaker notebook instances support Amazon Virtual Private Cloud (Amazon VPC) interface endpoints that are powered by AWS PrivateLink. Each VPC endpoint is represented by one or more Elastic Network Interfaces with private IP addresses in your VPC subnets.

Note
Before you create an interface VPC endpoint to connect to a notebook instance, create an interface VPC endpoint to connect to the SageMaker API. That way, when users call `CreatePresignedNotebookInstanceUrl` to get the URL to connect to the notebook instance, that call also goes through the interface VPC endpoint. For information, see Connect to SageMaker Through a VPC Interface Endpoint (p. 1683).

You can create an interface endpoint to connect to your notebook instance with either the AWS console or AWS Command Line Interface (AWS CLI) commands. For instructions, see Creating an Interface
Endpoint. Make sure that you create an interface endpoint for all of the subnets in your VPC from which you want to connect to the notebook instance.

When you create the interface endpoint, specify `aws.sagemaker.region.notebook` as the service name. After you create a VPC endpoint, enable private DNS for your VPC endpoint. Anyone using the SageMaker API, the AWS CLI, or the console to connect to the notebook instance from within the VPC will connect to the notebook instance through the VPC endpoint instead of the public internet.

SageMaker notebook instances support VPC endpoints in all AWS Regions where both Amazon VPC and SageMaker are available.

Topics
- Connect Your Private Network to Your VPC (p. 1685)
- Create a VPC Endpoint Policy for SageMaker Notebook Instances (p. 1685)
- Restrict Access to Connections from Within Your VPC (p. 1685)

Connect Your Private Network to Your VPC

To connect to your notebook instance through your VPC, you either have to connect from an instance that is inside the VPC, or connect your private network to your VPC by using an Amazon Virtual Private Network (VPN) or AWS Direct Connect. For information about Amazon VPN, see VPN Connections in the Amazon Virtual Private Cloud User Guide. For information about AWS Direct Connect, see Creating a Connection in the AWS Direct Connect User Guide.

Create a VPC Endpoint Policy for SageMaker Notebook Instances

You can create a policy for Amazon VPC endpoints for SageMaker notebook instances to specify the following:

- The principal that can perform actions.
- The actions that can be performed.
- The resources on which actions can be performed.

For more information, see Controlling Access to Services with VPC Endpoints in the Amazon VPC User Guide.

The following example of a VPC endpoint policy specifies that all users that have access to the endpoint are allowed to access the notebook instance named `myNotebookInstance`.

```json
{
    "Statement": [
        {
            "Action": "sagemaker:CreatePresignedNotebookInstanceUrl",
            "Effect": "Allow",
            "Principal": "*"
        }
    ]
}
```

Access to other notebook instances is denied.

Restrict Access to Connections from Within Your VPC

Even if you set up an interface endpoint in your VPC, individuals outside the VPC can connect to the notebook instance over the internet.
Important
If you apply an IAM policy similar to one of the following, users can’t access the specified SageMaker APIs or the notebook instance through the console.

To restrict access to only connections made from within your VPC, create an AWS Identity and Access Management policy that restricts access to only calls that come from within your VPC. Then add that policy to every AWS Identity and Access Management user, group, or role used to access the notebook instance.

Note
This policy allows connections only to callers within a subnet where you created an interface endpoint.

```
{
  "Id": "notebook-example-1",
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "Enable Notebook Access",
      "Effect": "Allow",
      "Action": [
        "sagemaker:CreatePresignedNotebookInstanceUrl",
        "sagemaker:DescribeNotebookInstance"
      ],
      "Resource": "*",
      "Condition": {
        "StringEquals": {
          "aws:SourceVpc": "vpc-111bbaaa"
        }
      }
    }
  ]
}
```

If you want to restrict access to the notebook instance to only connections made using the interface endpoint, use the `aws:SourceVpce` condition key instead of `aws:SourceVpc`:

```
{
  "Id": "notebook-example-1",
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "Enable Notebook Access",
      "Effect": "Allow",
      "Action": [
        "sagemaker:CreatePresignedNotebookInstanceUrl",
        "sagemaker:DescribeNotebookInstance"
      ],
      "Resource": "*",
      "Condition": {
        "ForAnyValue:StringEquals": {
          "aws:sourceVpce": [
            "vpce-111bbccc",
            "vpce-111bbddd"
          ]
        }
      }
    }
  ]
}
```

Both of these policy examples assume that you have also created an interface endpoint for the SageMaker API. For more information, see Connect to SageMaker Through a VPC Interface Endpoint.
Connect to SageMaker Through a VPC Interface Endpoint

In the second example, one of the values for `aws:SourceVpc` is the ID of the interface endpoint for the notebook instance. The other is the ID of the interface endpoint for the SageMaker API.

The policy examples here include `DescribeNotebookInstance` because typically you would call `DescribeNotebookInstance` to make sure that the `NotebookInstanceStatus` is `InService` before you try to connect to it. For example:

```
aws sagemaker describe-notebook-instance \
   --notebook-instance-name myNotebookInstance
```

```
{
   "NotebookInstanceName": "myNotebookInstance",
   "NotebookInstanceStatus": "InService",
   "Url": "mynotebookinstance.notebook.us-west-2.sagemaker.aws",
   "InstanceType": "ml.m4.xlarge",
   "RoleArn": "arn:aws:iam::1234567890ab:role/service-role/AmazonSageMaker-ExecutionRole-12345678T123456",
   "LastModifiedTime": 1540334777.501,
   "CreationTime": 1523050674.078,
   "DirectInternetAccess": "Disabled"
}
```

For both of these calls, if you did not enable private DNS hostnames for your VPC endpoint, or if you are using a version of the AWS SDK that was released before August 13, 2018, you must specify the endpoint URL in the call. For example, the call to `create-presigned-notebook-instance-url` would be:

```
aws sagemaker create-presigned-notebook-instance-url \
   --notebook-instance-name myNotebookInstance \
   --endpoint-url VPC_Endpoint_ID.api.sagemaker.Region.vpce.amazonaws.com
```

For the `create-presigned-notebook-instance-url` call, replace `VPC_Endpoint_ID` with the VPC endpoint ID for your notebook instance.

The `--endpoint-url` parameter specifies the VPC endpoint URL that you are connecting to.

For example:

```
aws sagemaker create-presigned-notebook-instance-url \
   --notebook-instance-name myNotebookInstance \
   --endpoint-url VPC_Endpoint_ID.api.sagemaker.Region.vpce.amazonaws.com
```

Connect Your Private Network to Your VPC

To call the SageMaker API and runtime through your VPC, you have to connect from an instance that is inside the VPC or connect your private network to your VPC by using an Amazon Virtual Private Network (VPN) or AWS Direct Connect. For information about Amazon VPN, see [VPN Connections](https://docs.aws.amazon.com/vpc/userguide/vpc-connections.html) in the Amazon Virtual Private Cloud User Guide. For information about AWS Direct Connect, see [Creating a Connection](https://docs.aws.amazon.com/directconnect/latest/userguide/creating-a-connection.html) in the AWS Direct Connect User Guide.

Connect to SageMaker Studio Through an Interface VPC Endpoint

You can connect to Amazon SageMaker Studio from your Amazon Virtual Private Cloud (Amazon VPC) through an interface endpoint in your VPC instead of connecting over the internet. When you use an
interface VPC endpoint (interface endpoint), communication between your VPC and Studio is conducted entirely and securely within the AWS network.

SageMaker Studio supports interface endpoints that are powered by AWS PrivateLink. Each interface endpoint is represented by one or more Elastic network interfaces with private IP addresses in your VPC subnets.

You can create an interface endpoint to connect to Studio with either the AWS console or the AWS Command Line Interface (AWS CLI). For instructions, see Creating an interface endpoint. Make sure that you create interface endpoints for all of the subnets in your VPC from which you want to connect to Studio.

**Note**
In addition to creating an interface endpoint to connect to SageMaker Studio, create an interface endpoint to connect to the Amazon SageMaker API. When users call `CreatePresignedDomainUrl` to get the URL to connect to Studio, that call goes through the interface endpoint.

When you create the interface endpoint, specify `aws.sagemaker.Region.studio` as the service name. After you create the interface endpoint, enable private DNS for your endpoint. Anyone who uses the SageMaker API, the AWS CLI, or the console to connect to SageMaker Studio from within the VPC will connect to Studio through the interface endpoint instead of the public internet.

Studio supports interface endpoints in all AWS Regions where both Amazon SageMaker and Amazon VPC are available.

**Topics**
- Create a VPC Endpoint Policy for SageMaker Studio (p. 1688)
- Allow Access Only from Within Your VPC (p. 1689)

**Create a VPC Endpoint Policy for SageMaker Studio**

You can attach an Amazon VPC endpoint policy to the interface VPC endpoints that you use to connect to SageMaker Studio. The endpoint policy controls access to Studio. You can specify the following:

- The principal that can perform actions
- The actions that can be performed
- The resources on which actions can be performed

For more information, see Controlling access to services with VPC endpoints.

The following example of a VPC endpoint policy specifies that all users that have access to the endpoint are allowed to access the user profiles in the SageMaker Studio domain with the specified domain ID. Access to other domains is denied.

```json
{
  "Statement": [
    {
      "Action": "sagemaker:CreatePresignedDomainUrl",
      "Effect": "Allow",
      "Principal": "*
    }
  ]
}
```
Allow Access Only from Within Your VPC

Users outside your VPC can connect to SageMaker Studio over the internet even if you set up an interface endpoint in your VPC.

To allow access to only connections made from within your VPC, create an AWS Identity and Access Management (IAM) policy to that effect. Add that policy to every IAM user, group, or role used to access Studio. The following examples demonstrate how to create such policies.

Important
If you apply an IAM policy similar to one of the following examples, users can't access SageMaker Studio or the specified SageMaker APIs through the SageMaker console. To access Studio, users must use a presigned URL or call the SageMaker APIs directly.

Example 1
The following policy allows connections only to callers within the subnet where you created the interface endpoint.

```json
{
  "Id": "sagemaker-studio-example-1",
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "Enable SageMaker Studio Access",
      "Effect": "Allow",
      "Action": [
        "sagemaker:CreatePresignedDomainUrl",
        "sagemaker:DescribeUserProfile"
      ],
      "Resource": "*",
      "Condition": {
        "StringEquals": {
          "aws:SourceVpc": "vpc-111bbaaa"
        }
      }
    }
  ]
}
```

Example 2
The following policy allows connections only to those made through the interface endpoints specified by the `aws:sourceVpce` condition key. For example, the first interface endpoint could allow access through the SageMaker Studio Control Panel. The second interface endpoint could allow access through the SageMaker API.

```json
{
  "Id": "sagemaker-studio-example-2",
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "Enable SageMaker Studio Access",
      "Effect": "Allow",
      "Action": [
        "sagemaker:CreatePresignedDomainUrl",
        "sagemaker:DescribeUserProfile"
      ],
      "Resource": "*",
      "Condition": {
        "ForAnyValue:StringEquals": {
          "aws:sourceVpce": [1689]
        }
      }
    }
  ]
}
```
This policy includes the `DescribeUserProfile` action. Typically you call `DescribeUserProfile` to make sure that the status of the user profile is `InService` before you try to connect to the domain. For example:

```
aws sagemaker describe-user-profile \
  --domain-id domain-id \
  --user-profile-name profile-name
```

Response:

```
{
  "DomainId": "domain-id",
  "UserProfileName": "profile-name",
  "HomeEfsFileSystemUid": "200001",
  "Status": "InService",
  "LastModifiedTime": 1605418785.555,
  "CreationTime": 1605418477.297
}
```

```
aws sagemaker create-presigned-domain-url \
  --domain-id domain-id \
  --user-profile-name profile-name
```

Response:

```
{
  "AuthorizedUrl": "https://domain-id.studio.us-west-2.sagemaker.aws/auth?token=AuthToken"
}
```

For both of these calls, if you are using a version of the AWS SDK that was released before August 13, 2018, you must specify the endpoint URL in the call. For example, the call to `create-presigned-domain-url` would be:

```
aws sagemaker create-presigned-domain-url \
  --domain-id domain-id \
  --user-profile-name profile-name \
  --endpoint-url vpc-endpoint-id.api.sagemaker.Region.vpce.amazonaws.com
```

**Example 3**

The following policy allows connections only from the specified range of IP addresses using the `aws:SourceIp` condition key.

```
{
  "Id": "sagemaker-studio-example-1",
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "example-1",
      "Effect": "Allow",
      "Action": "sagemaker:DescribeUserProfile",
      "Condition": {
        "aws:SourceIp": "10.0.0.0/16"
      }
    }
  ]
}
```
Give SageMaker Processing Jobs Access to Resources in Your Amazon VPC

SageMaker runs processing jobs in an Amazon Virtual Private Cloud by default. However, processing containers access AWS resources—such as the Amazon S3 buckets where you store data—over the internet.

To control access to your data and processing containers, we recommend that you create a private VPC and configure it so that they aren't accessible over the internet. For information about creating and configuring a VPC, see Getting Started With Amazon VPC in the Amazon VPC User Guide. Using a VPC helps to protect your processing containers and data because you can configure your VPC so that it is not connected to the internet. Using a VPC also allows you to monitor all network traffic in and out of your processing containers by using VPC flow logs. For more information, see VPC Flow Logs in the Amazon VPC User Guide.

You specify your private VPC configuration when you create processing jobs by specifying subnets and security groups. When you specify the subnets and security groups, SageMaker creates elastic network interfaces that are associated with your security groups in one of the subnets. Network interfaces allow your processing containers to connect to resources in your VPC. For information about network interfaces, see Elastic Network Interfaces in the Amazon VPC User Guide.

Configure a Processing Job for Amazon VPC Access

To specify subnets and security groups in your private VPC, use the `NetworkConfig.VpcConfig` request parameter of the CreateProcessingJob API, or provide this information when you create a processing job in the SageMaker console. SageMaker uses this information to create network interfaces and attach them to your processing containers. The network interfaces provide your processing containers with a network connection within your VPC that is not connected to the internet. They also enable your processing job to connect to resources in your private VPC.

The following is an example of the `VpcConfig` parameter that you include in your call to CreateProcessingJob:

```
VpcConfig: {
    "Subnets": [
        "subnet-0123456789abcdef0",
        "subnet-0123456789abcdef1",
    ],
    "SecurityGroups": [
        "sg-0123456789abcdef0",
        "sg-0123456789abcdef1"
    ],
}
```
Configure Your Private VPC for SageMaker Processing

When configuring the private VPC for your SageMaker processing jobs, use the following guidelines. For information about setting up a VPC, see Working with VPCs and Subnets in the Amazon VPC User Guide.

Topics
- Ensure That Subnets Have Enough IP Addresses (p. 1692)
- Create an Amazon S3 VPC Endpoint (p. 1692)
- Use a Custom Endpoint Policy to Restrict Access to S3 (p. 1692)
- Configure Route Tables (p. 1693)
- Configure the VPC Security Group (p. 1693)
- Connect to Resources Outside Your VPC (p. 1693)

Ensure That Subnets Have Enough IP Addresses

Your VPC subnets should have at least two private IP addresses for each instance in a processing job. For more information, see VPC and Subnet Sizing for IPv4 in the Amazon VPC User Guide.

Create an Amazon S3 VPC Endpoint

If you configure your VPC so that processing containers don’t have access to the internet, they can’t connect to the Amazon S3 buckets that contain your data unless you create a VPC endpoint that allows access. By creating a VPC endpoint, you allow your processing containers to access the buckets where you store your data. We recommend that you also create a custom policy that allows only requests from your private VPC to access to your S3 buckets. For more information, see Endpoints for Amazon S3.

To create an S3 VPC endpoint:

1. Open the Amazon VPC console at https://console.aws.amazon.com/vpc/.
2. In the navigation pane, choose Endpoints, then choose Create Endpoint.
3. For Service Name, choose com.amazonaws.region.s3, where region is the name of the region where your VPC resides.
4. For VPC, choose the VPC you want to use for this endpoint.
5. For Configure route tables, select the route tables to be used by the endpoint. The VPC service automatically adds a route to each route table you select that points any S3 traffic to the new endpoint.
6. For Policy, choose Full Access to allow full access to the S3 service by any user or service within the VPC. Choose Custom to restrict access further. For information, see Use a Custom Endpoint Policy to Restrict Access to S3 (p. 1692).

Use a Custom Endpoint Policy to Restrict Access to S3

The default endpoint policy allows full access to S3 for any user or service in your VPC. To further restrict access to S3, create a custom endpoint policy. For more information, see Using Endpoint Policies for Amazon S3. You can also use a bucket policy to restrict access to your S3 buckets to only traffic that comes from your Amazon VPC. For information, see Using Amazon S3 Bucket Policies.
Restrict Package Installation on the Processing Container

The default endpoint policy allows users to install packages from the Amazon Linux and Amazon Linux 2 repositories on the processing container. If you don't want users to install packages from that repository, create a custom endpoint policy that explicitly denies access to the Amazon Linux and Amazon Linux 2 repositories. The following is an example of a policy that denies access to these repositories:

```
{
    "Statement": [
        {
            "Sid": "AmazonLinuxAMIRepositoryAccess",
            "Principal": "*",
            "Action": ["s3:GetObject"],
            "Effect": "Deny",
            "Resource": [
                "arn:aws:s3:::packages.*.amazonaws.com/*",
                "arn:aws:s3:::repo.*.amazonaws.com/*"
            ]
        }
    ]
}
{
    "Statement": [
        {
            "Sid": "AmazonLinux2AMIRepositoryAccess",
            "Principal": "*",
            "Action": ["s3:GetObject"],
            "Effect": "Deny",
            "Resource": [
                "arn:aws:s3:::amazonlinux.*.amazonaws.com/*"
            ]
        }
    ]
}
```

Configure Route Tables

Use default DNS settings for your endpoint route table, so that standard Amazon S3 URLs (for example, http://s3-aws-region.amazonaws.com/MyBucket) resolve. If you don't use default DNS settings, ensure that the URLs that you use to specify the locations of the data in your processing jobs resolve by configuring the endpoint route tables. For information about VPC endpoint route tables, see Routing for Gateway Endpoints in the Amazon VPC User Guide.

Configure the VPC Security Group

In distributed processing, you must allow communication between the different containers in the same processing job. To do that, configure a rule for your security group that allows inbound connections between members of the same security group. For information, see Security Group Rules.

Connect to Resources Outside Your VPC

If you configure your VPC so that it doesn't have internet access, processing jobs that use that VPC do not have access to resources outside your VPC. If your processing job needs access to resources outside your VPC, provide access with one of the following options:

- If your processing job needs access to an AWS service that supports interface VPC endpoints, create an endpoint to connect to that service. For a list of services that support interface endpoints, see VPC
Endpoints in the Amazon VPC User Guide. For information about creating an interface VPC endpoint, see Interface VPC Endpoints (AWS PrivateLink) in the Amazon VPC User Guide.

- If your processing job needs access to an AWS service that doesn't support interface VPC endpoints or to a resource outside of AWS, create a NAT gateway and configure your security groups to allow outbound connections. For information about setting up a NAT gateway for your VPC, see Scenario 2: VPC with Public and Private Subnets (NAT) in the Amazon Virtual Private Cloud User Guide.

Give SageMaker Training Jobs Access to Resources in Your Amazon VPC

SageMaker runs training jobs in an Amazon Virtual Private Cloud by default. However, training containers access AWS resources—such as the Amazon S3 buckets where you store training data and model artifacts—over the internet.

To control access to your data and training containers, we recommend that you create a private VPC and configure it so that they aren't accessible over the internet. For information about creating and configuring a VPC, see Getting Started With Amazon VPC in the Amazon VPC User Guide. Using a VPC helps to protect your training containers and data because you can configure your VPC so that it is not connected to the internet. Using a VPC also allows you to monitor all network traffic in and out of your training containers by using VPC flow logs. For more information, see VPC Flow Logs in the Amazon VPC User Guide.

You specify your private VPC configuration when you create training jobs by specifying subnets and security groups. When you specify the subnets and security groups, SageMaker creates elastic network interfaces that are associated with your security groups in one of the subnets. Network interfaces allow your training containers to connect to resources in your VPC. For information about network interfaces, see Elastic Network Interfaces in the Amazon VPC User Guide.

Note
For training jobs, you can configure only subnets with a default tenancy VPC in which your instance runs on shared hardware. For more information on the tenancy attribute for VPCs, see Dedicated Instances.

Configure a Training Job for Amazon VPC Access

To specify subnets and security groups in your private VPC, use the VpcConfig request parameter of the CreateTrainingJob API, or provide this information when you create a training job in the SageMaker console. SageMaker uses this information to create network interfaces and attach them to your training containers. The network interfaces provide your training containers with a network connection within your VPC that is not connected to the internet. They also enable your training job to connect to resources in your private VPC.

The following is an example of the VpcConfig parameter that you include in your call to CreateTrainingJob:

```json

VpcConfig: {
  "Subnets": [
    "subnet-0123456789abcdef0",
    "subnet-0123456789abcdef1",
    "subnet-0123456789abcdef2"
  ],
  "SecurityGroupIds": ["sg-0123456789abcdef0"]
}

```
Configure Your Private VPC for SageMaker Training

When configuring the private VPC for your SageMaker training jobs, use the following guidelines. For information about setting up a VPC, see Working with VPCs and Subnets in the Amazon VPC User Guide.

Topics
- Ensure That Subnets Have Enough IP Addresses (p. 1695)
- Create an Amazon S3 VPC Endpoint (p. 1695)
- Use a Custom Endpoint Policy to Restrict Access to S3 (p. 1695)
- Configure Route Tables (p. 1696)
- Configure the VPC Security Group (p. 1696)
- Connect to Resources Outside Your VPC (p. 1696)

Ensure That Subnets Have Enough IP Addresses

Your VPC subnets should have at least two private IP addresses for each instance in a training job. For more information, see VPC and Subnet Sizing for IPv4 in the Amazon VPC User Guide.

Create an Amazon S3 VPC Endpoint

If you configure your VPC so that training containers don't have access to the internet, they can't connect to the Amazon S3 buckets that contain your training data unless you create a VPC endpoint that allows access. By creating a VPC endpoint, you allow your training containers to access the buckets where you store your data and model artifacts. We recommend that you also create a custom policy that allows only requests from your private VPC to access to your S3 buckets. For more information, see Endpoints for Amazon S3.

To create an S3 VPC endpoint:

1. Open the Amazon VPC console at https://console.aws.amazon.com/vpc/.
2. In the navigation pane, choose Endpoints, then choose Create Endpoint
3. For Service Name, choose com.amazonaws.region.s3, where region is the name of the region where your VPC resides.
4. For VPC, choose the VPC you want to use for this endpoint.
5. For Configure route tables, select the route tables to be used by the endpoint. The VPC service automatically adds a route to each route table you select that points any S3 traffic to the new endpoint.
6. For Policy, choose Full Access to allow full access to the S3 service by any user or service within the VPC. Choose Custom to restrict access further. For information, see Use a Custom Endpoint Policy to Restrict Access to S3 (p. 1695).

Use a Custom Endpoint Policy to Restrict Access to S3

The default endpoint policy allows full access to S3 for any user or service in your VPC. To further restrict access to S3, create a custom endpoint policy. For more information, see Using Endpoint Policies for Amazon S3. You can also use a bucket policy to restrict access to your S3 buckets to only traffic that comes from your Amazon VPC. For information, see Using Amazon S3 Bucket Policies.

Restrict Package Installation on the Training Container

The default endpoint policy allows users to install packages from the Amazon Linux and Amazon Linux 2 repositories on the training container. If you don't want users to install packages from that repository, create a custom endpoint policy that explicitly denies access to the Amazon Linux and Amazon Linux 2 repositories. The following is an example of a policy that denies access to these repositories:
Configure Route Tables

Use default DNS settings for your endpoint route table, so that standard Amazon S3 URLs (for example, http://s3-aws-region.amazonaws.com/MyBucket) resolve. If you don't use default DNS settings, ensure that the URLs that you use to specify the locations of the data in your training jobs resolve by configuring the endpoint route tables. For information about VPC endpoint route tables, see Routing for Gateway Endpoints in the Amazon VPC User Guide.

Configure the VPC Security Group

In distributed training, you must allow communication between the different containers in the same training job. To do that, configure a rule for your security group that allows inbound connections between members of the same security group. For information, see Security Group Rules.

Connect to Resources Outside Your VPC

If you configure your VPC so that it doesn't have internet access, training jobs that use that VPC do not have access to resources outside your VPC. If your training job needs access to resources outside your VPC, provide access with one of the following options:

- If your training job needs access to an AWS service that supports interface VPC endpoints, create an endpoint to connect to that service. For a list of services that support interface endpoints, see VPC Endpoints in the Amazon VPC User Guide. For information about creating an interface VPC endpoint, see Interface VPC Endpoints (AWS PrivateLink) in the Amazon VPC User Guide.
- If your training job needs access to an AWS service that doesn't support interface VPC endpoints or to a resource outside of AWS, create a NAT gateway and configure your security groups to allow outbound connections. For information about setting up a NAT gateway for your VPC, see Scenario 2: VPC with Public and Private Subnets (NAT) in the Amazon Virtual Private Cloud User Guide.
Give SageMaker Hosted Endpoints Access to Resources in Your Amazon VPC

SageMaker hosts models in an Amazon Virtual Private Cloud by default. However, models access AWS resources—such as the Amazon S3 buckets where you store training data and model artifacts—over the internet.

To avoid making your data and model containers accessible over the internet, we recommend that you create a private VPC and configure it to control access to them. For information about creating and configuring a VPC, see Getting Started With Amazon VPC in the Amazon VPC User Guide. Using a VPC helps to protect your training containers and data because you can configure your VPC so that it is not connected to the internet. Using a VPC also allows you to monitor all network traffic in and out of your training containers by using VPC flow logs. For more information, see VPC Flow Logs in the Amazon VPC User Guide.

You specify your private VPC configuration when you create a model by specifying subnets and security groups. When you specify the subnets and security groups, SageMaker creates elastic network interfaces that are associated with your security groups in one of the subnets. Network interfaces allow your model containers to connect to resources in your VPC. For information about network interfaces, see Elastic Network Interfaces in the Amazon VPC User Guide.

Configure a Model for Amazon VPC Access

To specify subnets and security groups in your private VPC, use the VpcConfig request parameter of the CreateModel API, or provide this information when you create a model in the SageMaker console. SageMaker uses this information to create network interfaces and attach them to your model containers. The network interfaces provide your model containers with a network connection within your VPC that is not connected to the internet. They also enable your model to connect to resources in your private VPC.

Note
You must create at least two subnets in different availability zones in your private VPC, even if you have only one hosting instance.

The following is an example of the VpcConfig parameter that you include in your call to CreateModel:

```
VpcConfig: {
    "Subnets": [
        "subnet-0123456789abcdef0",
        "subnet-0123456789abcdef1",
        "subnet-0123456789abcdef2"
    ],
    "SecurityGroupIds": [
        "sg-0123456789abcdef0"
    ]
}
```

Configure Your Private VPC for SageMaker Hosting

When configuring the private VPC for your SageMaker models, use the following guidelines. For information about setting up a VPC, see Working with VPCs and Subnets in the Amazon VPC User Guide.

Topics
- Ensure That Subnets Have Enough IP Addresses (p. 1698)
- Create an Amazon S3 VPC Endpoint (p. 1698)
- Use a Custom Endpoint Policy to Restrict Access to Amazon S3 (p. 1698)
Ensure That Subnets Have Enough IP Addresses

Your VPC subnets should have at least two private IP addresses for each model instance. For more information, see VPC and Subnet Sizing for IPv4 in the Amazon VPC User Guide.

Create an Amazon S3 VPC Endpoint

If you configure your VPC so that model containers don’t have access to the internet, they can’t connect to the Amazon S3 buckets that contain your data unless you create a VPC endpoint that allows access. By creating a VPC endpoint, you allow your model containers to access the buckets where you store your data and model artifacts. We recommend that you also create a custom policy that allows only requests from your private VPC to access to your S3 buckets. For more information, see Endpoints for Amazon S3.

To create an Amazon S3 VPC endpoint:

1. Open the Amazon VPC console at https://console.aws.amazon.com/vpc/.
2. In the navigation pane, choose Endpoints, then choose Create Endpoint.
3. For Service Name, choose `com.amazonaws.region.s3`, where `region` is the name of the AWS Region where your VPC resides.
4. For VPC, choose the VPC that you want to use for this endpoint.
5. For Configure route tables, choose the route tables that the endpoint will use. The VPC service automatically adds a route to each route table that you choose that points Amazon S3 traffic to the new endpoint.
6. For Policy, choose Full Access to allow full access to the Amazon S3 service by any user or service within the VPC. To restrict access further, choose Custom. For more information, see Use a Custom Endpoint Policy to Restrict Access to Amazon S3.

Use a Custom Endpoint Policy to Restrict Access to Amazon S3

The default endpoint policy allows full access to Amazon Simple Storage Service (Amazon S3) for any user or service in your VPC. To further restrict access to Amazon S3, create a custom endpoint policy. For more information, see Using Endpoint Policies for Amazon S3.

You can also use a bucket policy to restrict access to your S3 buckets to only traffic that comes from your Amazon VPC. For information, see Using Amazon S3 Bucket Policies.

Restrict Package Installation on the Model Container with a Custom Endpoint Policy

The default endpoint policy allows users to install packages from the Amazon Linux and Amazon Linux 2 repositories on the model container. If you don’t want users to install packages from those repositories, create a custom endpoint policy that explicitly denies access to the Amazon Linux and Amazon Linux 2 repositories. The following is an example of a policy that denies access to these repositories:

```json
{
   "Statement": [
      {
         "Sid": "AmazonLinuxAMImRepositoryAccess",
         "Principal": "*",
         "Action": ["s3:GetObject"
```

1698
Give SageMaker Hosted Endpoints Access to Resources in Your Amazon VPC

```
{
   "Effect": "Deny",
   "Resource": [
      "arn:aws:s3:::packages.*.amazonaws.com/*",
      "arn:aws:s3:::repo.*.amazonaws.com/*"
   ]
}
```

Add Permissions for Endpoint Access for Containers Running in a VPC to Custom IAM Policies

The `SageMakerFullAccess` managed policy includes the permissions that you need to use models configured for Amazon VPC access with an endpoint. These permissions allow SageMaker to create an elastic network interface and attach it to model containers running in a VPC. If you use your own IAM policy, you must add the following permissions to that policy to use models configured for VPC access.

```
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Effect": "Allow",
         "Action": [
            "ec2:DescribeVpcEndpoints",
            "ec2:DescribeDhcpOptions",
            "ec2:DescribeVpcs",
            "ec2:DescribeSubnets",
            "ec2:DescribeSecurityGroups",
            "ec2:DescribeNetworkInterfaces",
            "ec2:DeleteNetworkInterfacePermission",
            "ec2:DeleteNetworkInterface",
            "ec2:CreateNetworkInterfacePermission",
            "ec2:CreateNetworkInterface"
         ],
         "Resource": "*"
      }
   ]
}
```

For more information about the `SageMakerFullAccess` managed policy, see `AmazonSageMakerFullAccess Policy (p. 1662)`.

Configure Route Tables

Use default DNS settings for your endpoint route table, so that standard Amazon S3 URLs (for example, `http://s3-aws-region.amazonaws.com/MyBucket`) resolve. If you don't use default DNS settings,
ensure that the URLs that you use to specify the locations of the data in your models resolve by configuring the endpoint route tables. For information about VPC endpoint route tables, see Routing for Gateway Endpoints in the Amazon VPC User Guide.

Connect to Resources Outside Your VPC

If you configure your VPC so that it doesn't have internet access, models that use that VPC do not have access to resources outside your VPC. If your model needs access to resources outside your VPC, provide access with one of the following options:

- If your model needs access to an AWS service that supports interface VPC endpoints, create an endpoint to connect to that service. For a list of services that support interface endpoints, see VPC Endpoints in the Amazon VPC User Guide. For information about creating an interface VPC endpoint, see Interface VPC Endpoints (AWS PrivateLink) in the Amazon VPC User Guide.
- If your model needs access to an AWS service that doesn't support interface VPC endpoints or to a resource outside of AWS, create a NAT gateway and configure your security groups to allow outbound connections. For information about setting up a NAT gateway for your VPC, see Scenario 2: VPC with Public and Private Subnets (NAT) in the Amazon Virtual Private Cloud User Guide.

Give Batch Transform Jobs Access to Resources in Your Amazon VPC

SageMaker runs batch transform jobs in an Amazon Virtual Private Cloud by default. However, model containers access AWS resources—such as the Amazon S3 buckets where you store your data and model artifacts—over the internet.

To control access to your model containers and data, we recommend that you create a private VPC and configure it so that they aren't accessible over the internet. For information about creating and configuring a VPC, see Getting Started With Amazon VPC in the Amazon VPC User Guide. Using a VPC helps to protect your model containers and data because you can configure your VPC so that it is not connected to the internet. Using a VPC also allows you to monitor all network traffic in and out of your model containers by using VPC flow logs. For more information, see VPC Flow Logs in the Amazon VPC User Guide.

You specify your private VPC configuration when you create a model by specifying subnets and security groups. You then specify the same model when you create a batch transform job. When you specify the subnets and security groups, SageMaker creates elastic network interfaces that are associated with your security groups in one of the subnets. Network interfaces allow your model containers to connect to resources in your VPC. For information about network interfaces, see Elastic Network Interfaces in the Amazon VPC User Guide.

Configure a Batch Transform Job for Amazon VPC Access

To specify subnets and security groups in your private VPC, use the VpcConfig request parameter of the CreateModel API, or provide this information when you create a model in the SageMaker console. Then specify the same model in the ModelName request parameter of the CreateTransformJob API, or in the Model name field when you create a transform job in the SageMaker console. SageMaker uses this information to create network interfaces and attach them to your model containers. The network interfaces provide your model containers with a network connection within your VPC that is not connected to the internet. They also enable your transform job to connect to resources in your private VPC.

The following is an example of the VpcConfig parameter that you include in your call to CreateModel:

```json
VpcConfig: {
}
```
Give Batch Transform Jobs Access to Resources in Your Amazon VPC

If you are creating a model using the CreateModel API operation, the IAM execution role that you use to create your model must include the permissions described in CreateModel API: Execution Role Permissions (p. 1660), including the following permissions required for a private VPC.

When creating a model in the console, if you select Create a new role in the Model Settings section, the AmazonSageMakerFullAccess policy used to create the role will already contain these permissions. If you select Enter a custom IAM role ARN or Use existing role, the role ARN that you specify must have an execution policy attached with the following permissions.

```
{
  "Effect": "Allow",
  "Action": [
    "ec2:CreateNetworkInterface",
    "ec2:CreateNetworkInterfacePermission",
    "ec2:DeleteNetworkInterface",
    "ec2:DeleteNetworkInterfacePermission",
    "ec2:DescribeNetworkInterfaces",
    "ec2:DescribeVpcs",
    "ec2:DescribeSubnets",
    "ec2:DescribeSecurityGroups"
  ]
}
```

Configure Your Private VPC for SageMaker Batch Transform

When configuring the private VPC for your SageMaker batch transform jobs, use the following guidelines. For information about setting up a VPC, see Working with VPCs and Subnets in the Amazon VPC User Guide.

**Topics**
- Ensure That Subnets Have Enough IP Addresses (p. 1701)
- Create an Amazon S3 VPC Endpoint (p. 1701)
- Use a Custom Endpoint Policy to Restrict Access to S3 (p. 1702)
- Configure Route Tables (p. 1703)
- Configure the VPC Security Group (p. 1703)
- Connect to Resources Outside Your VPC (p. 1703)

**Ensure That Subnets Have Enough IP Addresses**

Your VPC subnets should have at least two private IP addresses for each instance in a transform job. For more information, see VPC and Subnet Sizing for IPv4 in the Amazon VPC User Guide.

**Create an Amazon S3 VPC Endpoint**

If you configure your VPC so that model containers don't have access to the internet, they can't connect to the Amazon S3 buckets that contain your data unless you create a VPC endpoint that allows access. By creating a VPC endpoint, you allow your model containers to access the buckets where you store your data.
data and model artifacts. We recommend that you also create a custom policy that allows only requests from your private VPC to access to your S3 buckets. For more information, see Endpoints for Amazon S3.

To create an S3 VPC endpoint:

1. Open the Amazon VPC console at https://console.aws.amazon.com/vpc/.
2. In the navigation pane, choose Endpoints, then choose Create Endpoint.
3. For Service Name, choose com.amazonaws.
   region.s3, where region is the name of the region where your VPC resides.
4. For VPC, choose the VPC you want to use for this endpoint.
5. For Configure route tables, select the route tables to be used by the endpoint. The VPC service automatically adds a route to each route table you select that points any S3 traffic to the new endpoint.
6. For Policy, choose Full Access to allow full access to the S3 service by any user or service within the VPC. Choose Custom to restrict access further. For information, see Use a Custom Endpoint Policy to Restrict Access to S3 (p. 1702).

Use a Custom Endpoint Policy to Restrict Access to S3

The default endpoint policy allows full access to S3 for any user or service in your VPC. To further restrict access to S3, create a custom endpoint policy. For more information, see Using Endpoint Policies for Amazon S3. You can also use a bucket policy to restrict access to your S3 buckets to only traffic that comes from your Amazon VPC. For information, see Using Amazon S3 Bucket Policies.

Restrict Package Installation on the Model Container

The default endpoint policy allows users to install packages from the Amazon Linux and Amazon Linux 2 repositories on the training container. If you don't want users to install packages from that repository, create a custom endpoint policy that explicitly denies access to the Amazon Linux and Amazon Linux 2 repositories. The following is an example of a policy that denies access to these repositories:

```
{
  "Statement": [
    {
      "Sid": "AmazonLinuxAMIRepositoryAccess",
      "Principal": "*",
      "Action": [
        "s3:GetObject"
      ],
      "Effect": "Deny",
      "Resource": [
        "arn:aws:s3:::packages.*.amazonaws.com/**",
        "arn:aws:s3:::repo.*.amazonaws.com/**"
      ]
    }
  ]
}
{
  "Statement": [
    { "Sid": "AmazonLinux2AMIRepositoryAccess",
      "Principal": "*",
      "Action": [
        "s3:GetObject"
      ],
      "Effect": "Deny",
      "Resource": [
        "arn:aws:s3:::amazonlinux.*.amazonaws.com/**"
      ]
    }
  ]
}
```
Configure Route Tables

Use default DNS settings for your endpoint route table, so that standard Amazon S3 URLs (for example, http://s3-aws-region.amazonaws.com/MyBucket) resolve. If you don't use default DNS settings, ensure that the URLs that you use to specify the locations of the data in your batch transform jobs resolve by configuring the endpoint route tables. For information about VPC endpoint route tables, see Routing for Gateway Endpoints in the Amazon VPC User Guide.

Configure the VPC Security Group

In distributed batch transform, you must allow communication between the different containers in the same batch transform job. To do that, configure a rule for your security group that allows inbound connections between members of the same security group. For information, see Security Group Rules.

Connect to Resources Outside Your VPC

If you configure your VPC so that it doesn't have internet access, batch transform jobs that use that VPC do not have access to resources outside your VPC. If your batch transform job needs access to resources outside your VPC, provide access with one of the following options:

- If your batch transform job needs access to an AWS service that supports interface VPC endpoints, create an endpoint to connect to that service. For a list of services that support interface endpoints, see VPC Endpoints in the Amazon VPC User Guide. For information about creating an interface VPC endpoint, see Interface VPC Endpoints (AWS PrivateLink) in the Amazon VPC User Guide.

- If your batch transform job needs access to an AWS service that doesn't support interface VPC endpoints or to a resource outside of AWS, create a NAT gateway and configure your security groups to allow outbound connections. For information about setting up a NAT gateway for your VPC, see Scenario 2: VPC with Public and Private Subnets (NAT) in the Amazon Virtual Private Cloud User Guide.
Monitor Amazon SageMaker

Monitoring is an important part of maintaining the reliability, availability, and performance of SageMaker and your other AWS solutions. AWS provides the following monitoring tools to watch SageMaker, report when something is wrong, and take automatic actions when appropriate:

- **Amazon CloudWatch** monitors your AWS resources and the applications that you run on AWS in real time. You can collect and track metrics, create customized dashboards, and set alarms that notify you or take actions when a specified metric reaches a threshold that you specify. For example, you can have CloudWatch track CPU usage or other metrics of your Amazon EC2 instances and automatically launch new instances when needed. For more information, see the [Amazon CloudWatch User Guide](#).

- **Amazon CloudWatch Logs** enables you to monitor, store, and access your log files from EC2 instances, AWS CloudTrail, and other sources. CloudWatch Logs can monitor information in the log files and notify you when certain thresholds are met. You can also archive your log data in highly durable storage. For more information, see the [Amazon CloudWatch Logs User Guide](#).

- **AWS CloudTrail** captures API calls and related events made by or on behalf of your AWS account and delivers the log files to an Amazon S3 bucket that you specify. You can identify which users and accounts called AWS, the source IP address from which the calls were made, and when the calls occurred. For more information, see the [AWS CloudTrail User Guide](#).

- **CloudWatch Events** delivers a near real-time stream of system events that describe changes in AWS resources. Create CloudWatch Events rules react to a status change in a SageMaker training, hyperparameter tuning, or batch transform job.

**Topics**

- Monitor Amazon SageMaker with Amazon CloudWatch (p. 1704)
- Log Amazon SageMaker Events with Amazon CloudWatch (p. 1712)
- Log Amazon SageMaker API Calls with AWS CloudTrail (p. 1713)
- Automating Amazon SageMaker with Amazon EventBridge (p. 1715)

Monitor Amazon SageMaker with Amazon CloudWatch

You can monitor Amazon SageMaker using Amazon CloudWatch, which collects raw data and processes it into readable, near real-time metrics. These statistics are kept for 15 months, so that you can access historical information and gain a better perspective on how your web application or service is performing. However, the Amazon CloudWatch console limits the search to metrics that were updated in the last 2 weeks. This limitation ensures that the most current jobs are shown in your namespace. To graph metrics without using a search, specify its exact name in the source view. You can also set alarms that watch for certain thresholds, and send notifications or take actions when those thresholds are met. For more information, see the [Amazon CloudWatch User Guide](#).

SageMaker model training jobs and endpoints write CloudWatch metrics and logs. The following tables list the metrics and dimensions for SageMaker.
## Endpoint Invocation Metrics

The `aws/sagemaker` namespace includes the following request metrics from calls to `InvokeEndpoint`.

Metrics are available at a 1-minute frequency.

For information about how long CloudWatch metrics are retained for, see `GetMetricStatistics` in the Amazon CloudWatch API Reference.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invocation4XXErrors</td>
<td>The number of <code>InvokeEndpoint</code> requests where the model returned a 4xx HTTP response code. For each 4xx response, 1 is sent; otherwise, 0 is sent. Units: None Valid statistics: Average, Sum</td>
</tr>
<tr>
<td>Invocation5XXErrors</td>
<td>The number of <code>InvokeEndpoint</code> requests where the model returned a 5xx HTTP response code. For each 5xx response, 1 is sent; otherwise, 0 is sent. Units: None Valid statistics: Average, Sum</td>
</tr>
<tr>
<td>Invocations</td>
<td>The number of <code>InvokeEndpoint</code> requests sent to a model endpoint. To get the total number of requests sent to a model endpoint, use the Sum statistic. Units: None Valid statistics: Sum, Sample Count</td>
</tr>
<tr>
<td>InvocationsPerInstance</td>
<td>The number of invocations sent to a model, normalized by <code>InstanceCount</code> in each ProductionVariant. $1/\text{numberOfInstances}$ is sent as the value on each request, where <code>numberOfInstances</code> is the number of active instances for the ProductionVariant behind the endpoint at the time of the request. Units: None Valid statistics: Sum</td>
</tr>
<tr>
<td>ModelLatency</td>
<td>The interval of time taken by a model to respond as viewed from SageMaker. This interval includes the local communication times taken to send the request and to fetch the response from the container of a model and the time taken to complete the inference in the container. Units: Microseconds Valid statistics: Average, Sum, Min, Max, Sample Count</td>
</tr>
<tr>
<td>OverheadLatency</td>
<td>The interval of time added to the time taken to respond to a client request by SageMaker overheads. This interval is measured from the time SageMaker receives the request until it returns a response to the client, minus the <code>ModelLatency</code>. Overhead latency can vary depending on multiple factors, including request and response payload sizes, request frequency, and authentication/authorization of the request. Units: Microseconds</td>
</tr>
</tbody>
</table>
### Metric Description

<table>
<thead>
<tr>
<th>Metric</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Valid statistics: Average, Sum, Min, Max, Sample Count</td>
</tr>
</tbody>
</table>

#### Dimensions for Endpoint Invocation Metrics

<table>
<thead>
<tr>
<th>Dimension</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EndpointName,</td>
<td>Filters endpoint invocation metrics for a ProductionVariant of the specified</td>
</tr>
<tr>
<td>VariantName</td>
<td>specified endpoint and variant.</td>
</tr>
</tbody>
</table>

#### Multi-Model Endpoint Model Loading Metrics

The *aws/SageMaker* namespace includes the following model loading metrics from calls to *InvokeEndpoint*.

Metrics are available at a 1-minute frequency.

For information about how long CloudWatch metrics are retained for, see [GetMetricStatistics](https://docs.aws.amazon.com/AmazonCloudWatch/latest/APIReference/API_GetMetricStatistics.html) in the *Amazon CloudWatch API Reference*.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ModelLoadingWaitTime</td>
<td>The interval of time that an invocation request has waited for the target model to be downloaded, or loaded, or both in order to perform inference.</td>
</tr>
<tr>
<td>ModelUnloadingTime</td>
<td>The interval of time that it took to unload the model through the container's UnloadModel API call.</td>
</tr>
<tr>
<td>ModelDownloadingTime</td>
<td>The interval of time that it took to download the model from Amazon Simple Storage Service (Amazon S3).</td>
</tr>
<tr>
<td>ModelLoadingTime</td>
<td>The interval of time that it took to load the model through the container's LoadModel API call.</td>
</tr>
<tr>
<td>ModelCacheHit</td>
<td>The number of InvokeEndpoint requests sent to the multi-model endpoint for which the model was already loaded. The Average statistic shows the ratio of requests for which the model was already loaded.</td>
</tr>
<tr>
<td></td>
<td>Units: None</td>
</tr>
</tbody>
</table>

| Units: Microseconds |
| Valid statistics: Average, Sum, Min, Max, Sample Count |

| Units: Microseconds |
| Valid statistics: Average, Sum, Min, Max, Sample Count |

| Units: Microseconds |
| Valid statistics: Average, Sum, Min, Max, Sample Count |

| Units: Microseconds |
| Valid statistics: Average, Sum, Min, Max, Sample Count |

| Units: None |

1706
<table>
<thead>
<tr>
<th>Metric</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Valid statistics: Average, Sum, Sample Count</td>
</tr>
</tbody>
</table>

**Dimensions for Multi-Model Endpoint Model Loading Metrics**

<table>
<thead>
<tr>
<th>Dimension</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EndpointName, VariantName</td>
<td>Filters endpoint invocation metrics for a ProductionVariant of the specified endpoint and variant.</td>
</tr>
</tbody>
</table>

**Multi-Model Endpoint Model Instance Metrics**

The `/aws/sagemaker/Endpoints` namespaces include the following instance metrics from calls to `InvokeEndpoint`.

Metrics are available at a 1-minute frequency.

For information about how long CloudWatch metrics are retained for, see `GetMetricStatistics` in the *Amazon CloudWatch API Reference*.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LoadedModelCount</td>
<td>The number of models loaded in the containers of the multi-model endpoint. This metric is emitted per instance.</td>
</tr>
<tr>
<td></td>
<td>The Average statistic with a period of 1 minute tells you the average number of models loaded per instance.</td>
</tr>
<tr>
<td></td>
<td>The Sum statistic tells you the total number of models loaded across all instances in the endpoint.</td>
</tr>
<tr>
<td></td>
<td>The models that this metric tracks are not necessarily unique because a model might be loaded in multiple containers at the endpoint.</td>
</tr>
<tr>
<td></td>
<td>Units: None</td>
</tr>
<tr>
<td></td>
<td>Valid statistics: Average, Sum, Min, Max, Sample Count</td>
</tr>
</tbody>
</table>

**Dimensions for Multi-Model Endpoint Model Loading Metrics**

<table>
<thead>
<tr>
<th>Dimension</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EndpointName, VariantName</td>
<td>Filters endpoint invocation metrics for a ProductionVariant of the specified endpoint and variant.</td>
</tr>
</tbody>
</table>

**Processing Job, Training Job, Batch Transform Job, and Endpoint Instance Metrics**

The `/aws/sagemaker/ProcessingJobs, /aws/sagemaker/TrainingJobs, /aws/sagemaker/TransformJobs` and `/aws/sagemaker/Endpoints` namespaces include the following metrics for the training jobs and endpoint instances.

Metrics are available at a 1-minute frequency.
<table>
<thead>
<tr>
<th>Metric</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPUUtilization</td>
<td>The percentage of CPU units that are used by the containers on an instance. The value can range between 0 and 100, and is multiplied by the number of CPUs. For example, if there are four CPUs, CPUUtilization can range from 0% to 400%. For processing jobs, the value is the CPU utilization of the processing container on the instance. For training jobs, the value is the CPU utilization of the algorithm container on the instance. For batch transform jobs, the value is the CPU utilization of the transform container on the instance. For endpoint variants, the value is the sum of the CPU utilization of the primary and supplementary containers on the instance. <strong>Note</strong> For multi-instance, each instance reports CPU utilization metrics. However, the default view in CloudWatch shows the average CPU utilization across all instances. <strong>Units:</strong> Percent</td>
</tr>
<tr>
<td>MemoryUtilization</td>
<td>The percentage of memory that is used by the containers on an instance. This value can range between 0% and 100%. For processing jobs, the value is the memory utilization of the processing container on the instance. For training jobs, the value is the memory utilization of the algorithm container on the instance. For batch transform jobs, the value is the memory utilization of the transform container on the instance. For endpoint variants, the value is the sum of the memory utilization of the primary and supplementary containers on the instance. <strong>Note</strong> For multi-instance, each instance reports memory utilization metrics. However, the default view in CloudWatch shows the average memory utilization across all instances. <strong>Units:</strong> Percent</td>
</tr>
<tr>
<td>GPUUtilization</td>
<td>The percentage of GPU units that are used by the containers on an instance. The value can range between 0 and 100 and is multiplied by the number of GPUs. For example, if there are four GPUs, GPUUtilization can range from 0% to 400%. For processing jobs, the value is the GPU utilization of the processing container on the instance. For training jobs, the value is the GPU utilization of the algorithm container on the instance. For batch transform jobs, the value is the GPU utilization of the transform container on the instance.</td>
</tr>
<tr>
<td>Metric</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| GPUMemoryUtilization | For endpoint variants, the value is the sum of the GPU utilization of the primary and supplementary containers on the instance.  
  **Note**  
  For multi-instance, each instance reports GPU utilization metrics. However, the default view in CloudWatch shows the average GPU utilization across all instances.  
  Units: Percent                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| DiskUtilization  | The percentage of disk space used by the containers on an instance uses. This value can range between 0% and 100%. This metric is not supported for batch transform jobs. For processing jobs, the value is the disk space utilization of the processing container on the instance. For training jobs, the value is the disk space utilization of the algorithm container on the instance. For endpoint variants, the value is the sum of the disk space utilization of the primary and supplementary containers on the instance.  
  **Note**  
  For multi-instance, each instance reports disk utilization metrics. However, the default view in CloudWatch shows the average disk utilization across all instances.  
  Units: Percent                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |

**Dimensions for Processing Job, Training Job and Batch Transform Job Instance Metrics**
<table>
<thead>
<tr>
<th>Dimension</th>
<th>Description</th>
</tr>
</thead>
</table>
| Host      | For processing jobs, the value for this dimension has the format `[processing-job-name]/algo-[instance-number-in-cluster]`. Use this dimension to filter instance metrics for the specified processing job and instance. This dimension format is present only in the `/aws/sagemaker/ProcessingJobs` namespace.  

For training jobs, the value for this dimension has the format `[training-job-name]/algo-[instance-number-in-cluster]`. Use this dimension to filter instance metrics for the specified training job and instance. This dimension format is present only in the `/aws/sagemaker/TrainingJobs` namespace.  

For batch transform jobs, the value for this dimension has the format `[transform-job-name]/[instance-id]`. Use this dimension to filter instance metrics for the specified batch transform job and instance. This dimension format is present only in the `/aws/sagemaker/TransformJobs` namespace. |

---

**Amazon SageMaker Ground Truth Metrics**

<table>
<thead>
<tr>
<th>Metric</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ActiveWorkers</td>
<td>The number of workers on a private work team performing a labeling job.</td>
</tr>
<tr>
<td></td>
<td>Units: None</td>
</tr>
<tr>
<td></td>
<td>Valid statistics: Max</td>
</tr>
<tr>
<td>DatasetObjectsAutoAnnotated</td>
<td>The number of dataset objects auto-annotated in a labeling job. This metric is only emitted when automated labeling is enabled. To view the labeling job progress, use the Max metric.</td>
</tr>
<tr>
<td></td>
<td>Units: None</td>
</tr>
<tr>
<td></td>
<td>Valid statistics: Max</td>
</tr>
<tr>
<td>DatasetObjectsHumanAnnotated</td>
<td>The number of dataset objects annotated by a human in a labeling job. To view the labeling job progress, use the Max metric.</td>
</tr>
<tr>
<td></td>
<td>Units: None</td>
</tr>
<tr>
<td></td>
<td>Valid statistics: Max</td>
</tr>
<tr>
<td>DatasetObjectsLabelingFailed</td>
<td>The number of dataset objects that failed labeling in a labeling job. To view the labeling job progress, use the Max metric.</td>
</tr>
<tr>
<td></td>
<td>Units: None</td>
</tr>
<tr>
<td></td>
<td>Valid statistics: Max</td>
</tr>
<tr>
<td>JobsFailed</td>
<td>The number of labeling jobs that failed. To get the total number of labeling jobs that failed, use the Sum statistic.</td>
</tr>
<tr>
<td></td>
<td>Units: None</td>
</tr>
<tr>
<td></td>
<td>Valid statistics: Sum, Sample Count</td>
</tr>
</tbody>
</table>
### Amazon SageMaker Developer Guide

**Monitoring with CloudWatch**

<table>
<thead>
<tr>
<th>Metric</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>JobsSucceeded</td>
<td>The number of labeling jobs that succeeded. To get the total number of labeling jobs that succeeded, use the Sum statistic.</td>
</tr>
<tr>
<td></td>
<td>Units: None</td>
</tr>
<tr>
<td></td>
<td>Valid statistics: Sum, Sample Count</td>
</tr>
<tr>
<td>JobsStopped</td>
<td>The number of labeling jobs that were stopped. To get the total number of labeling jobs that were stopped, use the Sum statistic.</td>
</tr>
<tr>
<td></td>
<td>Units: None</td>
</tr>
<tr>
<td></td>
<td>Valid statistics: Sum, Sample Count</td>
</tr>
<tr>
<td>TasksAccepted</td>
<td>The total number of tasks accepted by workers.</td>
</tr>
<tr>
<td></td>
<td>Units: None</td>
</tr>
<tr>
<td></td>
<td>Valid statistics: Max</td>
</tr>
<tr>
<td>TasksReturned</td>
<td>The total number of tasks returned by workers.</td>
</tr>
<tr>
<td></td>
<td>Units: None</td>
</tr>
<tr>
<td></td>
<td>Valid statistics: Max</td>
</tr>
<tr>
<td>TasksSubmitted</td>
<td>The number of tasks submitted/completed by a private work team.</td>
</tr>
<tr>
<td></td>
<td>Units: None</td>
</tr>
<tr>
<td></td>
<td>Valid statistics: Max</td>
</tr>
<tr>
<td>TimeSpent</td>
<td>Time spent on a task completed by a private work team.</td>
</tr>
<tr>
<td></td>
<td>Units: Seconds</td>
</tr>
<tr>
<td></td>
<td>Valid statistics: Max</td>
</tr>
<tr>
<td>TotalDatasetObjectsLabeled</td>
<td>The number of dataset objects labeled successfully in a labeling job. To view the labeling job progress, use the Max metric.</td>
</tr>
<tr>
<td></td>
<td>Units: None</td>
</tr>
<tr>
<td></td>
<td>Valid statistics: Max</td>
</tr>
</tbody>
</table>

**Dimensions for Dataset Object Metrics**

<table>
<thead>
<tr>
<th>Dimension</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LabelingJobName</td>
<td>Filters dataset object count metrics for a labeling job.</td>
</tr>
</tbody>
</table>
Log Amazon SageMaker Events with Amazon CloudWatch

To help you debug your processing jobs, training jobs, endpoints, transform jobs, notebook instances, and notebook instance lifecycle configurations, anything an algorithm container, a model container, or a notebook instance lifecycle configuration sends to `stdout` or `stderr` is also sent to Amazon CloudWatch Logs. In addition to debugging, you can use these for progress analysis.

**Logs**

The following table lists all of the logs provided by Amazon SageMaker.

<table>
<thead>
<tr>
<th>Log Group Name</th>
<th>Log Stream Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>/aws/sagemaker/Endpoints/ [EndpointName]</td>
<td>[production-variant-name]/[instance-id]</td>
</tr>
<tr>
<td></td>
<td>[production-variant-name]/[instance-id]/[container-name provided in SageMaker model] (For Inference Pipelines)</td>
</tr>
<tr>
<td>/aws/sagemaker/groundtruth/WorkerActivity</td>
<td>aws/sagemaker/groundtruth/worker-activity/[requester-AWS-Id]-[region]/[timestamp]</td>
</tr>
<tr>
<td>/aws/sagemaker/LabelingJobs</td>
<td>[labeling-job-name]</td>
</tr>
<tr>
<td>/aws/sagemaker/NotebookInstances</td>
<td>[notebook-instance-name]/[LifecycleConfigHook]</td>
</tr>
<tr>
<td></td>
<td>[notebook-instance-name]/jupyter.log</td>
</tr>
<tr>
<td>/aws/sagemaker/ProcessingJobs</td>
<td>[processing-job-name]/[hostname]-[epoch_timestamp]</td>
</tr>
<tr>
<td>/aws/sagemaker/Studio</td>
<td>[domain-id]/[user-profile-name]/[app-type]/[app-name]</td>
</tr>
<tr>
<td>/aws/sagemaker/TrainingJobs</td>
<td>[training-job-name]/algo-[instance-number-in-cluster]-[epoch_timestamp]</td>
</tr>
<tr>
<td>/aws/sagemaker/TransformJobs</td>
<td>[transform-job-name]/[instance-id]-[epoch_timestamp]</td>
</tr>
<tr>
<td></td>
<td>[transform-job-name]/[instance-id]-[epoch_timestamp]/data-log</td>
</tr>
<tr>
<td></td>
<td>[transform-job-name]/[instance-id]-[epoch_timestamp]/[container-name provided in SageMaker model] (For Inference Pipelines)</td>
</tr>
</tbody>
</table>

**Note**

1. The `/aws/sagemaker/NotebookInstances/[LifecycleConfigHook]` log stream is created when you create a notebook instance with a lifecycle configuration. For more information, see Customize a Notebook Instance Using a Lifecycle Configuration Script (p. 127).
2. For Inference Pipelines, if you don't provide container names, the platform uses "container-1, container-2", and so on, corresponding to the order provided in the SageMaker model.
Log Amazon SageMaker API Calls with AWS CloudTrail

Amazon SageMaker is integrated with AWS CloudTrail, a service that provides a record of actions taken by a user, role, or an AWS service in SageMaker. CloudTrail captures all API calls for SageMaker, with the exception of InvokeEndpoint, as events. The calls captured include calls from the SageMaker console and code calls to the SageMaker API operations. If you create a trail, you can enable continuous delivery of CloudTrail events to an Amazon S3 bucket, including events for SageMaker. If you don't configure a trail, you can still view the most recent events in the CloudTrail console in Event history. Using the information collected by CloudTrail, you can determine the request that was made to SageMaker, the IP address from which the request was made, who made the request, when it was made, and additional details.

To learn more about CloudTrail, see the AWS CloudTrail User Guide.

By default, log data is stored in CloudWatch Logs indefinitely. However, you can configure how long to store log data in a log group. For information, see Change Log Data Retention in CloudWatch Logs in the Amazon CloudWatch Logs User Guide.

SageMaker Information in CloudTrail

CloudTrail is enabled on your AWS account when you create the account. When activity occurs in Amazon SageMaker, that activity is recorded in a CloudTrail event along with other AWS service events in Event history. You can view, search, and download recent events in your AWS account. For more information, see Viewing Events with CloudTrail Event History.

For an ongoing record of events in your AWS account, including events for Amazon SageMaker, create a trail. A trail enables CloudTrail to deliver log files to an Amazon S3 bucket. By default, when you create a trail in the console, the trail applies to all AWS Regions. The trail logs events from all Regions in the AWS partition and delivers the log files to the Amazon S3 bucket that you specify. Additionally, you can configure other AWS services to further analyze and act upon the event data collected in CloudTrail logs. For more information, see the following:

- Overview for Creating a Trail
- CloudTrail Supported Services and Integrations
- Configuring Amazon SNS Notifications for CloudTrail
- Receiving CloudTrail Log Files from Multiple Regions and Receiving CloudTrail Log Files from Multiple Accounts

All SageMaker actions, with the exception of InvokeEndpoint, are logged by CloudTrail and are documented in the Operations. For example, calls to the CreateTrainingJob, CreateEndpoint and CreateNotebookInstance actions generate entries in the CloudTrail log files.

Every event or log entry contains information about who generated the request. The identity information helps you determine the following:

- Whether the request was made with root or AWS Identity and Access Management (IAM) user credentials.
- Whether the request was made with temporary security credentials for a role or federated user.
- Whether the request was made by another AWS service.
Operations Performed by Automatic Model Tuning

SageMaker supports logging non-API service events to your CloudTrail log files for automatic model tuning jobs. These events are related to your tuning jobs but, are not the direct result of a customer request to the public AWS API. For example, when you create a hyperparameter tuning job by calling `CreateHyperParameterTuningJob`, SageMaker creates training jobs to evaluate various combinations of hyperparameters to find the best result. Similarly, when you call `StopHyperParameterTuningJob` to stop a hyperparameter tuning job, SageMaker might stop any of the associated running training jobs. Non-API events for your tuning jobs are logged to CloudTrail to help you improve governance, compliance, and operational and risk auditing of your AWS account.

Log entries that result from non-API service events have an `eventType` of `AwsServiceEvent` instead of `AwsApiCall`.

Understanding SageMaker Log File Entries

A trail is a configuration that enables delivery of events as log files to an S3 bucket that you specify. CloudTrail log files contain one or more log entries. An event represents a single request from any source and includes information about the requested action, the date and time of the action, request parameters, and so on. CloudTrail log files are not an ordered stack trace of the public API calls, so they do not appear in any specific order.

The following examples a log entry for the `CreateEndpoint` action, which creates an endpoint to deploy a trained model.

```json
{
  "eventVersion": "1.05",
  "userIdentity": {
    "type": "IAMUser",
    "principalId": "AIXDAYQEXAMPLEUMLYNGL",
    "arn": "arn:aws:iam::123456789012:user/intern",
    "accountId": "123456789012",
    "accessKeyId": "ASXIAGXEXAMPLEQULKNXV",
    "userName": "intern"
  },
  "eventTime": "2018-01-02T13:39:06Z",
  "eventSource": "sagemaker.amazonaws.com",
  "eventName": "CreateEndpoint",
  "awsRegion": "us-west-2",
  "sourceIPAddress": "127.0.0.1",
  "userAgent": "USER_AGENT",
  "requestParameters": {
    "endpointName": "ExampleEndpoint",
    "endpointConfigName": "ExampleEndpointConfig"
  },
  "responseElements": {
    "endpointArn": "arn:aws:sagemaker:us-west-2:123456789012:endpoint/exampleendpoint"
  },
  "requestID": "6b1b42b9-EXAMPLE",
  "eventID": "a6f85b21-EXAMPLE",
  "eventType": "AwsApiCall",
  "recipientAccountId": "444455556666"
}
```

The following example is a log entry for the `CreateModel` action, which creates one or more containers to host a previously trained model.

```json
{
  "eventVersion": "1.05",
  "userIdentity": {
    "type": "IAMUser",
    "principalId": "AIXDAYQEXAMPLEUMLYNGL",
    "arn": "arn:aws:iam::123456789012:user/intern",
    "accountId": "123456789012",
    "accessKeyId": "ASXIAGXEXAMPLEQULKNXV",
    "userName": "intern"
  },
  "eventTime": "2018-01-02T13:39:06Z",
  "eventSource": "sagemaker.amazonaws.com",
  "eventName": "CreateModel",
  "awsRegion": "us-west-2",
  "sourceIPAddress": "127.0.0.1",
  "userAgent": "USER_AGENT",
  "requestParameters": {
    "modelName": "ExampleModel",
    "versionName": "ExampleVersion"
  },
  "responseElements": {
    "modelArn": "arn:aws:sagemaker:us-west-2:123456789012:model/examplemodel"
  },
  "requestID": "6b1b42b9-EXAMPLE",
  "eventID": "a6f85b21-EXAMPLE",
  "eventType": "AwsApiCall",
  "recipientAccountId": "444455556666"
}
```

For more information, see the CloudTrail `userIdentity` Element.
Automating Amazon SageMaker with Amazon EventBridge

Amazon EventBridge monitors status change events in Amazon SageMaker labeling, training, hyperparameter tuning, process jobs, and inference endpoints. EventBridge enables you to automate SageMaker and respond automatically to events such as training job or endpoint status changes. Events from SageMaker are delivered to EventBridge in near real time. You can write simple rules to indicate which events are of interest to you, and what automated actions to take when an event matches a rule. The actions that can be automatically triggered include the following:

- Invoking an AWS Lambda function
- Invoking Amazon EC2 Run Command
- Relaying the event to Amazon Kinesis Data Streams
- Activating an AWS Step Functions state machine
- Notifying an Amazon SNS topic or an AWS SMS queue

Some examples of SageMaker status change events that EventBridge monitors include:

- SageMaker training job state change

  Indicates a change in the status of a SageMaker training job.

  If the value of TrainingJobStatus is Failed, the event contains the FailureReason field, which provides a description of why the training job failed.
• SageMaker HyperParameter tuning job state change

Indicates a change in the status of a SageMaker hyperparameter tuning job.

```json
{
  "version": "0",
  "id": "844e2571-85d4-695f-b930-0153b71dcb42",
  "detail-type": "SageMaker HyperParameter Tuning Job State Change",
  "source": "aws.sagemaker",
  "account": "123456789012",
  "time": "2018-10-06T12:26:13Z",
  "region": "us-east-1",
  "resources": [
    "arn:aws:sagemaker:us-east-1:123456789012:tuningJob/x"
  ],
  "detail": {
    "HyperParameterTuningJobName": "016bffd3-6d71-4d3a-9710-0a332b2759fc",
    "HyperParameterTuningJobArn": "arn:aws:sagemaker:us-east-1:123456789012:tuningJob/x",
    "TrainingJobDefinition": {
      "StaticHyperParameters": {},
      "AlgorithmSpecification": {
        "TrainingImage": "trainingImageName",
        "TrainingInputMode": "inputModeFile",
        "MetricDefinitions": [
          {
            "Name": "metricName",
            "Regex": "regex"
          }
        ],
        "RoleArn": "roleArn",
        "InputDataConfig": [
          {
            "ChannelName": "channelName",
            "DataSource": {
              "S3DataSource": {
                "S3DataType": "s3DataType",
                "S3Uri": "s3Uri",
                "S3DataDistributionType": "s3DistributionType"
              }
            },
            "ContentType": "contentType",
            "CompressionType": "gzip",
            "RecordWrapperType": "RecordWrapper"
          }
        ],
        "VpcConfig": {
          "SecurityGroupId": ["securityGroupId"],
          "Subnets": ["subnets"]
        },
        "OutputDataConfig": {
          "KmsKeyId": "kmsKeyId",
          "S3OutputPath": "s3OutputPath"
        },
        "ResourceConfig": {
          "InstanceType": "instanceType",
```
SageMaker transform job state change

Indicates a change in the status of a SageMaker batch transform job.

If the value of TransformJobStatus is Failed, the event contains the FailureReason field, which provides a description of why the training job failed.
SageMaker HyperParameter tuning job state change

Indicates a change in the status of a SageMaker hyperparameter tuning job.

```json
{
  "version": "0",
  "id": "844e2571-85d4-695f-b930-0153b71dcb42",
  "detail-type": "SageMaker HyperParameter Tuning Job State Change",
  "source": "aws.sagemaker",
  "account": "123456789012",
  "time": "1583831889050",
  "region": "us-east-1",
  "resources": [
    "arn:aws:sagemaker:us-east-1:123456789012:tuningJob/x"
  ],
  "detail": {
    "HyperParameterTuningJobName": "016bffd3-6d71-4d3a-9710-0a332b2759fc",
    "HyperParameterTuningJobArn": "arn:aws:sagemaker:us-east-1:123456789012:tuningJob/x",
    "TrainingJobDefinition": {
      "StaticHyperParameters": {},
      "AlgorithmSpecification": {
        "TrainingImage": "trainingImageName",
        "TrainingInputMode": "inputModeFile",
        "MetricDefinitions": [
          {
            "Name": "metricName",
            "Regex": "regex"
          }
        ]
      },
      "RoleArn": "roleArn",
      "InputDataConfig": [
        {
          "ChannelName": "channelName",
          "DataSource": {
            "S3DataSource": {
              "S3DataDistributionType": "s3DistributionType"
            }
          },
          "ContentType": "contentType",
          "CompressionType": "gz",
          "SplitType": "split type",
          "S3OutputPath": "s3Uri",
          "Accept": "accept",
          "AssembleWith": "assemblyType",
          "KmsKeyId": "kmsKeyId"
        }
      ]
    }
  }
}```
SageMaker endpoint status change

Indicates a change in the status of a SageMaker hosted real-time inference endpoint.

The following shows an event with an endpoint in the IN_SERVICE state.

```json
{
    'version': '0',
    'id': 'd2921b5a-b0ad-cace-a8e3-0f159d018e06',
    'detail-type': 'SageMakerEndpointStateChange',
    'source': 'aws.sagemaker',
    'account': '123456789012',
    'time': '1583831889050',
    'region': 'us-west-2',
    'resources': [
        'arn:aws:sagemaker:us-west-2:123456789012:endpoint/myendpoint'
    ],
    'detail': {
        'EndpointName': 'MyEndpoint',
        'EndpointConfigName': 'MyEndpointConfig'
    }
}
```
'ProductionVariants': [
  {
    'DesiredWeight': 1.0,
    'DesiredInstanceCount': 1.0
  }
],
'EndpointStatus': 'IN_SERVICE',
'CreationTime': 1592411992203.0,
'LastModifiedTime': 1592411994287.0,
'Tags': {
}
}

For more information about the status values and their meanings for SageMaker jobs and endpoints, see the following links:

- AlgorithmStatus
- EndpointStatus
- HyperParameterTuningJobStatus
- LabelingJobStatus
- ModelPackageStatus
- NotebookInstanceStatus
- ProcessingJobStatus
- TrainingJobStatus
- TransformJobStatus
- EndpointStatus

For more information, see the Amazon EventBridge User Guide.
Availability Zones

For the AWS Regions supported by Amazon SageMaker and the Amazon Elastic Compute Cloud (Amazon EC2) instance types that are available in each Region, see Amazon SageMaker Pricing.

Each AWS Region is divided into sub-regions known as Availability Zones. For a given Region, the Availability Zones in that Region don’t always contain the same instance types supported by SageMaker.

Amazon SageMaker Studio is available in all the AWS Regions supported by Amazon SageMaker except the AWS GovCloud (US) Regions. In the supported Regions, Studio is available in the same Availability Zones as notebook instances.

For more information, see Regions and Availability Zones in the Amazon EC2 User Guide and AZ IDs for Your Resources in the AWS RAM User Guide.

The following list provides links to the Availability Zone tables for each AWS Region supported by SageMaker. Each linked table has a column for each Availability Zone in the Region. For each Availability Zone, the SageMaker components that support each instance type are shown.

Availability Zone Tables

- US East (Ohio) us-east-2
- US East (N. Virginia) us-east-1
- US West (N. California) us-west-1
- US West (Oregon) us-west-2
- Asia Pacific (Hong Kong) ap-east-1
- Asia Pacific (Mumbai) ap-south-1
- Asia Pacific (Seoul) ap-northeast-2
- Asia Pacific (Singapore) ap-southeast-1
- Asia Pacific (Sydney) ap-southeast-2
- Asia Pacific (Tokyo) ap-northeast-1
- Canada (Central) ca-central-1
- EU (Frankfurt) eu-central-1
- EU (Ireland) eu-west-1
- EU (London) eu-west-2
- EU (Paris) eu-west-3
- EU (Stockholm) eu-north-1
- Middle East (Bahrain) me-south-1
- South America (Sao Paulo) sa-east-1
- AWS GovCloud (US-Gov-West) us-gov-west-1
- US ISO East us-iso-east-1

The following AWS Regions are supported by Amazon SageMaker. Availability Zones tables are not available at this time. For more information, contact AWS Support.

- Africa (Cape Town) af-south-1
- EU (Milan) eu-south-1
API Reference Guide for Amazon SageMaker

Overview

Amazon SageMaker provides APIs, SDKs, and a command line interface that you can use to create and manage notebook instances and train and deploy models.

- Amazon SageMaker Python SDK (Recommended)
- Amazon SageMaker API Reference
- Amazon Augmented AI API Reference
- AWS Command Line Interface
- AWS SDK for .NET
- AWS SDK for C++
- AWS SDK for Go
- AWS SDK for Java
- AWS SDK for JavaScript
- AWS SDK for PHP
- AWS SDK for Python (Boto)
- AWS SDK for Ruby
- Amazon SageMaker Spark

You can also get code examples from the Amazon SageMaker example notebooks GitHub repository.

- Example notebooks

Programming Model for Amazon SageMaker

Making API calls directly from code is cumbersome, and requires you to write code to authenticate your requests. Amazon SageMaker provides the following alternatives:

- **Use the SageMaker console**—With the console, you don't write any code. You use the console UI to start model training or deploy a model. The console works well for simple jobs, where you use a built-in training algorithm and you don't need to preprocess training data.

- **Modify the example Jupyter notebooks**—SageMaker provides several Jupyter notebooks that train and deploy models using specific algorithms and datasets. Start with a notebook that has a suitable algorithm and modify it to accommodate your data source and specific needs.

- **Write model training and inference code from scratch**—SageMaker provides multiple AWS SDK languages (listed in the overview) and the Amazon SageMaker Python SDK, a high-level Python library that you can use in your code to start model training jobs and deploy the resulting models.
• **The SageMaker Python SDK**—This Python library simplifies model training and deployment. In addition to authenticating your requests, the library abstracts platform specifics by providing simple methods and default parameters. For example:

  • To deploy your model, you call only the `deploy()` method. The method creates a SageMaker model artifact, an endpoint configuration, then deploys the model on an endpoint.

  • If you use a custom framework script for model training, you call the `fit()` method. The method creates a .gzip file of your script, uploads it to an Amazon S3 location, and then runs it for model training, and other tasks. For more information, see *Use Machine Learning Frameworks, Python, and R with Amazon SageMaker* (p. 17).

• **The AWS SDKs**—The SDKs provide methods that correspond to the SageMaker API (see *Operations*). Use the SDKs to programmatically start a model training job and host the model in SageMaker. SDK clients authenticate your requests by using your access keys, so you don’t need to write authentication code. They are available in multiple languages and platforms. For more information, see the preceding list in the overview.

In *Get Started with Amazon SageMaker* (p. 33), you train and deploy a model using an algorithm provided by SageMaker. That exercise shows how to use both of these libraries. For more information, see *Get Started with Amazon SageMaker* (p. 33).

• **Integrate SageMaker into your Apache Spark workflow**—SageMaker provides a library for calling its APIs from Apache Spark. With it, you can use SageMaker-based estimators in an Apache Spark pipeline. For more information, see *Use Apache Spark with Amazon SageMaker* (p. 18).
## Document History for Amazon SageMaker

<table>
<thead>
<tr>
<th>update-history-change</th>
<th>update-history-description</th>
<th>update-history-date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Studio Notebooks</td>
<td>SageMaker Studio Notebooks</td>
<td>April 28, 2020</td>
</tr>
<tr>
<td>Configuring notebook instances</td>
<td>You can use shell scripts to configure notebook instances when you create or start them. For more information, see Customize a Notebook Instance.</td>
<td>May 1, 2018</td>
</tr>
<tr>
<td>Disable direct internet access</td>
<td>You can now disable direct internet access for notebook instances. For more information, see Notebook Instances Are Enabled with Internet Access by Default.</td>
<td>March 15, 2018</td>
</tr>
<tr>
<td>Application Auto Scaling support</td>
<td>Amazon SageMaker now supports Application Auto Scaling for production variants. For information, see Automatically Scaling SageMaker Models</td>
<td>February 28, 2018</td>
</tr>
<tr>
<td>TensorFlow 1.5 and MXNet 1.0 support (p. 1725)</td>
<td>Amazon SageMaker Deep Learning containers now support TensorFlow 1.5 and Apache MXNet 1.0.</td>
<td>February 27, 2018</td>
</tr>
<tr>
<td>Algorithm/Support</td>
<td>Description</td>
<td>Date</td>
</tr>
<tr>
<td>-----------------------------------------</td>
<td>------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>------------</td>
</tr>
<tr>
<td><strong>BlazingText algorithm</strong></td>
<td>Amazon SageMaker now supports the BlazingText algorithm.</td>
<td>January 18, 2018</td>
</tr>
<tr>
<td><strong>KMS encryption support for training and hosting</strong></td>
<td>Amazon SageMaker now supports KMS encryption for hosting instances and training model artifacts at rest. You can specify a AWS Key Management Service key that Amazon SageMaker uses to encrypt data on the storage volume attached to a hosting endpoint by using the KmsKeyId request parameter in a call to CreateEndpointConfig. You can specify an AWS KMS key that SageMaker uses to encrypt training model artifacts at rest by setting the KmsKeyId field of the OutputDataConfig object you use to configure your training job.</td>
<td>January 17, 2018</td>
</tr>
<tr>
<td><strong>CloudTrail support</strong></td>
<td>Amazon SageMaker now supports logging with AWS CloudTrail.</td>
<td>January 11, 2018</td>
</tr>
<tr>
<td><strong>DeepAR Forecasting algorithm</strong></td>
<td>Amazon SageMaker now supports the DeepAR algorithm for time series forecasting.</td>
<td>January 8, 2018</td>
</tr>
</tbody>
</table>
AWS glossary

For the latest AWS terminology, see the AWS glossary in the AWS General Reference.