Using the SageMaker Python SDK

SageMaker Python SDK provides several high-level abstractions for working with Amazon SageMaker. These are:

- **ModelTrainer**: New interface encapsulating training on SageMaker.
- **Estimators**: Encapsulate training on SageMaker.
- **Models**: Encapsulate built ML models.
- **Predictors**: Provide real-time inference and transformation using Python data-types against a SageMaker endpoint.
- **Session**: Provides a collection of methods for working with SageMaker resources.
- **Transformers**: Encapsulate batch transform jobs for inference on SageMaker
- **Processors**: Encapsulate running processing jobs for data processing on SageMaker

`Estimator` and `Model` implementations for MXNet, TensorFlow, Chainer, PyTorch, scikit-learn, Amazon SageMaker built-in algorithms, Reinforcement Learning, are included. There's also an `Estimator` that runs SageMaker compatible custom Docker containers, enabling you to run your own ML algorithms by using the SageMaker Python SDK.

Contents

stable

Train a Model with the SageMaker Python SDK

To train a model by using the SageMaker Python SDK, you:

1. Prepare a training script
2. Create a ModelTrainer or Estimator
3. Call the `train` method of the ModelTrainer or the `fit` method of the Estimator

After you train a model, you can save it, and then serve the model as an endpoint to get real-time inferences or get inferences for an entire dataset by using batch transform.

Important Note:

- When using torch to load Models, it is recommended to use version torch>=2.6.0 and torchvision>=0.17.0

Prepare a Training script

Your training script must be a 3.6 compatible source file.

The training script is very similar to a training script you might run outside of SageMaker, but you can access useful properties about the training environment through various environment variables, including the following:

- `SM_MODEL_DIR`: A string that represents the path where the training job writes the model artifacts to. After training, artifacts in this directory are uploaded to S3 for model hosting.
- `SM_NUM_GPUS`: An integer representing the number of GPUs available to the host.
- `SM_CHANNEL_XXXX`: A string that represents the path to the directory that contains the input data for the specified channel. For example, if you specify two input channels in the MXNet estimator's `fit` call, named 'train' and 'test', the environment variables `SM_CHANNEL_TRAIN` and `SM_CHANNEL_TEST` are set.
- `SM_HPS`: A json dump of the hyperparameters preserving json types (boolean, integer, etc.)

For the exhaustive list of available environment variables, see the SageMaker Containers documentation.

stable

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 deployed for inference later. Hyperparameters are passed to your script as arguments and can be retrieved with an `argparse.ArgumentParser` instance. For example, a training script might start with the following:

```
import argparse
import os
import json

if __name__ =='__main__':

    parser = argparse.ArgumentParser()

    # hyperparameters sent by the client are passed as command-line arguments to the
script.
    parser.add_argument('--epochs', type=int, default=10)
    parser.add_argument('--batch-size', type=int, default=100)
    parser.add_argument('--learning-rate', type=float, default=0.1)

    # an alternative way to load hyperparameters via SM_HPS environment variable.
    parser.add_argument('--sm-hps', type=json.loads, default=os.environ['SM_HPS'])

    # input data and model directories
    parser.add_argument('--model-dir', type=str, default=os.environ['SM_MODEL_DIR'])
    parser.add_argument('--train', type=str, default=os.environ['SM_CHANNEL_TRAIN'])
    parser.add_argument('--test', type=str, default=os.environ['SM_CHANNEL_TEST'])

    args, _ = parser.parse_known_args()

    # ... load from args.train and args.test, train a model, write model to
args.model_dir.
```

Because the SageMaker imports your training script, you should put your training code in a main guard (`if __name__=='__main__':`) if you are using the same script to host your model, so that SageMaker does not inadvertently run your training code at the wrong point in execution.

Note that SageMaker doesn’t support argparse actions. If you want to use, for example, boolean hyperparameters, you need to specify `type` as `bool` in your script and provide an explicit `True` or `False` value for this hyperparameter when you create your estimator.

For more on training environment variables, please visit SageMaker Containers.

Using ModelTrainer

To use the ModelTrainer class, you need to provide a few essential parameters training image URI and the source code configuration. The class allows you to s SageMaker training job with minimal parameters, particularly by specifying the source code and

stable

training image.

For more information about class definitions see ModelTrainer.

Example: Launching a Training Job with Custom Script

```
from sagemaker.modules.train import ModelTrainer
from sagemaker.modules.configs import SourceCode, InputData

# Image URI for the training job
pytorch_image = "763104351884.dkr.ecr.us-west-2.amazonaws.com/pytorch-training:2.0.0-cpu-
py310"

# Define the script to be run
source_code = SourceCode(
    source_dir="basic-script-mode",
    requirements="requirements.txt",
    entry_script="custom_script.py",
)

# Define the ModelTrainer
model_trainer = ModelTrainer(
    training_image=pytorch_image,
    source_code=source_code,
    base_job_name="script-mode",
)

# Pass the input data
input_data = InputData(
    channel_name="train",
    data_source=training_input_path, # S3 path where training data is stored
)

# Start the training job
model_trainer.train(input_data_config=[input_data], wait=False)
```

Using Estimators

Here is an end to end example of how to use a SageMaker Estimator:

```python
from sagemaker.mxnet import MXNet

# Configure an MXNet Estimator (no training happens yet)
mxnet_estimator = MXNet('train.py',
                        role='SageMakerRole',
                        instance_type='ml.p2.xlarge',
                        instance_count=1,
                        framework_version='1.2.1')

# Starts a SageMaker training job and waits until completion.
mxnet_estimator.fit('s3://my_bucket/my_training_data/')

# Deploys the model that was generated by fit() to a SageMaker endpoint
mxnet_predictor = mxnet_estimator.deploy(initial_instance_count=1,
instance_type='ml.p2.xlarge')

# Serializes data and makes a prediction request to the SageMaker endpoint
response = mxnet_predictor.predict(data)

# Tears down the SageMaker endpoint and endpoint configuration
mxnet_predictor.delete_endpoint()

# Deletes the SageMaker model
mxnet_predictor.delete_model()
```

The example above will eventually delete both the SageMaker endpoint and endpoint configuration through `delete_endpoint()`. If you want to keep your SageMaker endpoint configuration, use the value `False` for the `delete_endpoint_config` parameter, as shown below.

```python
# Only delete the SageMaker endpoint, while keeping the corresponding endpoint
configuration.
mxnet_predictor.delete_endpoint(delete_endpoint_config=False)
```

Additionally, it is possible to deploy a different endpoint configuration, which links to your model, to an already existing SageMaker endpoint. This can be done by specifying the existing endpoint name for the `endpoint_name` parameter along with the `update_endpoint` parameter as `True` within your `deploy()` call. For more information.

```
from sagemaker.mxnet import MXNet

# Configure an MXNet Estimator (no training happens yet)
mxnet_estimator = MXNet('train.py',
                        role='SageMakerRole',
                        instance_type='ml.p2.xlarge',
                        instance_count=1,
                        framework_version='1.2.1')

# Starts a SageMaker training job and waits until completion.
mxnet_estimator.fit('s3://my_bucket/my_training_data/')

# Deploys the model that was generated by fit() to an existing SageMaker endpoint
mxnet_predictor = mxnet_estimator.deploy(initial_instance_count=1,
                                         instance_type='ml.p2.xlarge',
                                         update_endpoint=True,
                                         endpoint_name='existing-endpoint')

# Serializes data and makes a prediction request to the SageMaker endpoint
response = mxnet_predictor.predict(data)

# Tears down the SageMaker endpoint and endpoint configuration
mxnet_predictor.delete_endpoint()

# Deletes the SageMaker model
mxnet_predictor.delete_model()
```

Using SageMaker AlgorithmEstimators

With the SageMaker Algorithm entities, you can create training jobs with just an `algorithm_arn` instead of a training image. There is a dedicated `AlgorithmEstimator` class that accepts `algorithm_arn` as a parameter, the rest of the arguments are similar to the other Estimator classes. This class also allows you to consume algorithms that you have subscribed to in the AWS Marketplace. The AlgorithmEstimator performs client-side validation on your inputs based on the algorithm's properties.

Here is an example:

```
import sagemaker

algo = sagemaker.AlgorithmEstimator(
    algorithm_arn='arn:aws:sagemaker:us-west-2:1234567:algorithm/some-algorithm',
    role='SageMakerRole',
    instance_count=1,
    instance_type='ml.c4.xlarge')

train_input = algo.sagemaker_session.upload_data(path='/path/to/your/data')

algo.fit({'training': train_input})
predictor = algo.deploy(1, 'ml.m4.xlarge')

# When you are done using your endpoint
predictor.delete_endpoint()
```

Use Scripts Stored in a Git Repository

When you create an estimator, you can specify a training script that is stored in a GitHub (or other Git) or CodeCommit repository as the entry point for the estimator, so that you don't have to download the scripts locally. If you do so, source directory and dependencies should be in the same repo if they are needed. Git support can be enabled simply by providing `git_config` parameter when creating an `Estimator` object. If Git support is enabled, then `entry_point`, `source_dir` and `dependencies` should be relative paths in the Git repo if provided.

The `git_config` parameter includes fields `repo`, `branch`, `commit`, `2FA_enabled`, `username`, `password` and `token`. The `repo` field is required. All other fields are optional. `repo` specifies the Git repository where your training script is stored. If you don't provide `branch`, the default value 'master' is used. If you don't provide `commit`, the latest commit in the specified branch is used.

`2FA_enabled`, `username`, `password` and `token` are used for authentication. For GitHub (or other Git) accounts, set `2FA_enabled` to 'True' if two-factor authentication is enabled for the account, otherwise set it to 'False'. If you do not provide a value for `2FA_enabled`, a default value of 'False' is used. CodeCommit does not support two-factor authentication, so do not provide "2FA_enabled" with CodeCommit repositories.

For GitHub or other Git repositories, If `repo` is an SSH URL, you should either have no passphrase for the SSH key pairs, or have the `ssh-agent` configured so that you are not prompted for the SSH passphrase when you run a `git clone` command with SSH URLs. For SSH URLs, it does not matter whether two-factor authentication is enabled. If `repo` is an HTTPS URL, 2FA matters. When 2FA is disabled, either `token` or `username``+`
used for authentication if provided (`token` prioritized). When 2FA is enabled, c

stable

used for authentication if provided. If required authentication info is not provided, python SDK will try to use local credentials storage to authenticate. If that fails either, an error message will be thrown.

For CodeCommit repos, please make sure you have completed the authentication setup: https://docs.aws.amazon.com/codecommit/latest/userguide/setting-up.html. 2FA is not supported by CodeCommit, so `2FA_enabled` should not be provided. There is no token in CodeCommit, so `token` should not be provided either. If `repo` is an SSH URL, the requirements are the same as GitHub repos. If `repo` is an HTTPS URL, `username``+``password` will be used for authentication if they are provided; otherwise, Python SDK will try to use either CodeCommit credential helper or local credential storage for authentication.

Here are some examples of creating estimators with Git support:

```
# Specifies the git_config parameter. This example does not provide Git credentials, so
python SDK will try
# to use local credential storage.
git_config = {'repo': 'https://github.com/username/repo-with-training-scripts.git',
              'branch': 'branch1',
              'commit': '4893e528afa4a790331e1b5286954f073b0f14a2'}

# In this example, the source directory 'pytorch' contains the entry point 'mnist.py' and
other source code.
# and it is relative path inside the Git repo.
pytorch_estimator = PyTorch(entry_point='mnist.py',
                            role='SageMakerRole',
                            source_dir='pytorch',
                            git_config=git_config,
                            instance_count=1,
                            instance_type='ml.c4.xlarge')
```

```
# You can also specify git_config by providing only 'repo' and 'branch'.
# If this is the case, the latest commit in that branch will be used.
git_config = {'repo': 'git@github.com:username/repo-with-training-scripts.git',
              'branch': 'branch1'}

# In this example, the entry point 'mnist.py' is all we need for source code.
# We need to specify the path to it in the Git repo.
mx_estimator = MXNet(entry_point='mxnet/mnist.py',
                     role='SageMakerRole',
                     git_config=git_config,
                     instance_count=1,
                     instance_type='ml.c4.xlarge')
```

```python
# Only providing 'repo' is also allowed. If this is the case, latest commit in 'master'
branch will be used.
# This example does not provide '2FA_enabled', so 2FA is treated as disabled by default.
'username' and
# 'password' are provided for authentication
git_config = {'repo': 'https://github.com/username/repo-with-training-scripts.git',
              'username': 'username',
              'password': 'passw0rd!'}

# In this example, besides entry point and other source code in source directory, we still
need some
# dependencies for the training job. Dependencies should also be paths inside the Git
repo.
pytorch_estimator = PyTorch(entry_point='mnist.py',
                            role='SageMakerRole',
                            source_dir='pytorch',
                            dependencies=['dep.py', 'foo/bar.py'],
                            git_config=git_config,
                            instance_count=1,
                            instance_type='ml.c4.xlarge')
```

```python
# This example specifies that 2FA is enabled, and token is provided for authentication
git_config = {'repo': 'https://github.com/username/repo-with-training-scripts.git',
              '2FA_enabled': True,
              'token': 'your-token'}

# In this exmaple, besides entry point, we also need some dependencies for the training
job.
pytorch_estimator = PyTorch(entry_point='pytorch/mnist.py',
                            role='SageMakerRole',
                            dependencies=['dep.py'],
                            git_config=git_config,
                            instance_count=1,
                            instance_type='local')
```

```python
# This example specifies a CodeCommit repository, and try to authenticate with provided
username+password
git_config = {'repo': 'https://git-codecommit.us-west-
2.amazonaws.com/v1/repos/your_repo_name',
              'username': 'username',
              'password': 'passw0rd!'}

mx_estimator = MXNet(entry_point='mxnet/mnist.py',
                     role='SageMakerRole',
                     git_config=git_config,
                     instance_count=1,
                     instance_type='ml.c4.xlarge')
```

stable

Git support can be used not only for training jobs, but also for hosting models. The usage is the same as the above, and `git_config` should be provided when creating model objects, e.g. `TensorFlowModel`, `MXNetModel`, `PyTorchModel`.

Use File Systems as Training Inputs

Amazon SageMaker supports using Amazon Elastic File System (EFS) and FSx for Lustre as data sources to use during training. If you want use those data sources, create a file system (EFS/FSx) and mount the file system on an Amazon EC2 instance. For more information about setting up EFS and FSx, see the following documentation:

- Using File Systems in Amazon EFS
- Getting Started with Amazon FSx for Lustre

The general experience uses either the `FileSystemInput` or `FileSystemRecordSet` class, which encapsulates all of the necessary arguments required by the service to use EFS or Lustre.

Here are examples of how to use Amazon EFS as input for training:

```
# This example shows how to use FileSystemInput class
# Configure an estimator with subnets and security groups from your VPC. The EFS volume
must be in
# the same VPC as your Amazon EC2 instance
estimator = TensorFlow(entry_point='tensorflow_mnist/mnist.py',
                       role='SageMakerRole',
                       instance_count=1,
                       instance_type='ml.c4.xlarge',
                       subnets=['subnet-1', 'subnet-2']
                       security_group_ids=['sg-1'])

file_system_input = FileSystemInput(file_system_id='fs-1',
                                    file_system_type='EFS',
                                    directory_path='/tensorflow',
                                    file_system_access_mode='ro')

# Start an Amazon SageMaker training job with EFS using the FileSystemInput class
estimator.fit(file_system_input)
```

```
# This example shows how to use FileSystemRecordSet class
# Configure an estimator with subnets and security groups from your VPC. The EFS volume
must be in
# the same VPC as your Amazon EC2 instance
kmeans = KMeans(role='SageMakerRole',
                instance_count=1,
                instance_type='ml.c4.xlarge',
                k=10,
                subnets=['subnet-1', 'subnet-2'],
                security_group_ids=['sg-1'])

records = FileSystemRecordSet(file_system_id='fs-1,
                              file_system_type='EFS',
                              directory_path='/kmeans',
                              num_records=784,
                              feature_dim=784)

# Start an Amazon SageMaker training job with EFS using the FileSystemRecordSet class
kmeans.fit(records)
```

Here are examples of how to use Amazon FSx for Lustre as input for training:

```
# This example shows how to use FileSystemInput class
# Configure an estimator with subnets and security groups from your VPC. The VPC should be
the same as that
# you chose for your Amazon EC2 instance

estimator = TensorFlow(entry_point='tensorflow_mnist/mnist.py',
                       role='SageMakerRole',
                       instance_count=1,
                       instance_type='ml.c4.xlarge',
                       subnets=['subnet-1', 'subnet-2']
                       security_group_ids=['sg-1'])


file_system_input = FileSystemInput(file_system_id='fs-2',
                                    file_system_type='FSxLustre',
                                    directory_path='/<mount-id>/tensorflow',
                                    file_system_access_mode='ro')

# Start an Amazon SageMaker training job with FSx using the FileSystemInput class
estimator.fit(file_system_input)
```

```
# This example shows how to use FileSystemRecordSet class
# Configure an estimator with subnets and security groups from your VPC. The VPC should be
the same as that
# you chose for your Amazon EC2 instance
kmeans = KMeans(role='SageMakerRole',
                instance_count=1,
                instance_type='ml.c4.xlarge',
                k=10,
                subnets=['subnet-1', 'subnet-2'],
                security_group_ids=['sg-1'])

records = FileSystemRecordSet(file_system_id='fs-=2,
                              file_system_type='FSxLustre',
                              directory_path='/<mount-id>/kmeans',
                              num_records=784,
                              feature_dim=784)

# Start an Amazon SageMaker training job with FSx using the FileSystemRecordSet class
kmeans.fit(records)
```

Data sources from EFS and FSx can also be used for hyperparameter tuning jobs. The usage is the same as above.

A few important notes:

- Local mode is not supported if using EFS and FSx as data sources
- Pipe mode is not supported if using EFS as data source

Training Metrics

The SageMaker Python SDK allows you to specify a name and a regular expression for metrics you want to track for training. A regular expression (regex) matches what is in the training algorithm logs, like a search function. Here is an example of how to define metrics:

```
# Configure an BYO Estimator with metric definitions (no training happens yet)
byo_estimator = Estimator(image_uri=image_uri,
                          role='SageMakerRole', instance_count=1,
                          instance_type='ml.c4.xlarge',
                          sagemaker_session=sagemaker_session,
                          metric_definitions=[{'Name': 'test:msd', 'Regex':
'#quality_metric: host=\S+, test msd <loss>=(\S+)'},
                                              {'Name': 'test:ssd', 'Regex':
'#quality_metric: host=\S+, test ssd <loss>=(\S+)'}])
```

All Amazon SageMaker algorithms come with built-in support for metrics. You c
AWS documentation for more details about built-in metrics of each Amazon Sa
algorithm.

stable

BYO Docker Containers with SageMaker Estimators

To use a Docker image that you created and use the SageMaker SDK for training, the easiest way is to use the dedicated `Estimator` class. You can create an instance of the `Estimator` class with desired Docker image and use it as described in previous sections.

Please refer to the full example in the examples repo:

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

The example notebook is located here: `advanced_functionality/scikit_bring_your_own/scikit_bring_your_own.ipynb`

You can also find this notebook in the **Advanced Functionality** folder of the **SageMaker Examples** section in a notebook instance. For information about using sample notebooks in a SageMaker notebook instance, see Use Example Notebooks in the AWS documentation.

Incremental Training

Incremental training allows you to bring a pre-trained model into a SageMaker training job and use it as a starting point for a new model. There are several situations where you might want to do this:

- You want to perform additional training on a model to improve its fit on your data set.
- You want to import a pre-trained model and fit it to your data.
- You want to resume a training job that you previously stopped.

To use incremental training with SageMaker algorithms, you need model artifacts compressed into a `tar.gz` file. These artifacts are passed to a training job via an input channel configured with the pre-defined settings Amazon SageMaker algorithms require.

To use model files with a SageMaker estimator, you can use the following parameters:

- `model_uri`: points to the location of a model tarball, either in S3 or locally. Specifying a local path only works in local mode.
- `model_channel_name`: name of the channel SageMaker will use to download the tarball specified in `model_uri`. Defaults to 'model'.

This is converted into an input channel with the specifications mentioned abov stable
`fit()` on the predictor. In bring-your-own cases, `model_channel_name` can be overriden if you require to change the name of the channel while using the same settings.

If your bring-your-own case requires different settings, you can create your own `s3_input` object with the settings you require.

Here's an example of how to use incremental training:

```
# Configure an estimator
estimator = 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)

# Start a SageMaker training job and waits until completion.
estimator.fit('s3://my_bucket/my_training_data/')

# Create a new estimator using the previous' model artifacts
incr_estimator = 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,
                                              model_uri=estimator.model_data)

# Start a SageMaker training job using the original model for incremental training
incr_estimator.fit('s3://my_bucket/my_training_data/')
```

Currently, the following algorithms support incremental training:

- Image Classification
- Object Detection
- Semantic Segmentation

Using Models Trained Outside of Amazon SageMaker

You can use models that you train outside of Amazon SageMaker, and model packages that you create or subscribe to in the AWS Marketplace to get inferences.

BYO Model

You can create an endpoint from an existing model that you trained outside of Sagemaker. That is, you can bring your own model:

First, package the files for the trained model into a `.tar.gz` file, and upload the archive to S3.

Next, create a `Model` object that corresponds to the framework that you are using: MXNetModel or TensorFlowModel.

Example code using `MXNetModel`:

```
from sagemaker.mxnet.model import MXNetModel

sagemaker_model = MXNetModel(model_data='s3://path/to/model.tar.gz',
                             role='arn:aws:iam::accid:sagemaker-role',
                             entry_point='entry_point.py')
```

After that, invoke the `deploy()` method on the `Model`:

```
predictor = sagemaker_model.deploy(initial_instance_count=1,
                                   instance_type='ml.m4.xlarge')
```

This returns a predictor the same way an `Estimator` does when `deploy()` is called. You can now get inferences just like with any other model deployed on Amazon SageMaker.

A full example is available in the Amazon SageMaker examples repository.

You can also find this notebook in the **Advanced Functionality** section of the **SageMaker Examples** section in a notebook instance. For information about using sample notebooks in a SageMaker notebook instance, see Use Example Notebooks in the AWS documentation.

Consuming SageMaker Model Packages

SageMaker Model Packages are a way to specify and share information for how to create SageMaker Models. With a SageMaker Model Package that you have created or subscribed to in the AWS Marketplace, you can use the specified serving image and model data for Endpoints and Batch Transform jobs.

To work with a SageMaker Model Package, use the `ModelPackage` class.

Here is an example:

```
import sagemaker

model = sagemaker.ModelPackage(
    role='SageMakerRole',
    model_package_arn='arn:aws:sagemaker:us-west-2:123456:model-package/my-model-package')
model.deploy(1, 'ml.m4.xlarge', endpoint_name='my-endpoint')

# When you are done using your endpoint
model.sagemaker_session.delete_endpoint('my-endpoint')
```

Use Built-in Algorithms with Pre-trained Models in SageMaker Python SDK

The SageMaker Python SDK provides built-in algorithms with pre-trained models from popular open source model hubs, such as TensorFlow Hub, Pytorch Hub, and HuggingFace. You can deploy these pre-trained models as-is or first fine-tune them on a custom dataset and then deploy to a SageMaker endpoint for inference.

SageMaker SDK built-in algorithms allow customers to access pre-trained models using model IDs and model versions. The ‘pre-trained model’ table below provides a list of models with useful information for selecting the correct model ID and corresponding parameters. These models are also available through the JumpStart UI in SageMaker Studio.

- Built-in Algorithms with pre-trained Model Table

Example notebooks

Explore example notebooks to get started with pretrained models using the SageMaker Python SDK.

Example notebooks for foundation models

All JumpStart foundation models are available to use programmatically with the SageMaker Python SDK. For a list of available example notebooks related to JumpStart foundation models, see JumpStart foundation models example notebooks.

Example notebooks for task-based models

SageMaker built-in algorithms with pre-trained models support 15 different machine learning problem types. Below is a list of all the supported problem types with a link to a Jupyter notebook that provides example usage.

stable

Vision

- Image Classification
- Object Detection
- Semantic Segmentation
- Instance Segmentation
- Image Embedding

Text

- Text Classification
- Sentence Pair Classification
- Question Answering
- Named Entity Recognition
- Text Summarization
- Text Generation
- Machine Translation
- Text Embedding

Tabular

- Tabular Classification (LightGBM & Catboost)
- Tabular Classification (XGBoost & Scikit-learn Linear Learner)
- Tabular Classification (AutoGluon)
- Tabular Classification (TabTransformer)
- Tabular Regression (LightGBM & Catboost)
- Tabular Regression (XGBoost & Scikit-learn Linear Learner)
- Tabular Regression (AutoGluon)
- Tabular Regression (TabTransformer)

Prerequisites

- You must set up AWS credentials. For more information, see Configuring the AWS CLI.
- Your IAM role must allow connection to Amazon SageMaker and Amazon S3. For more information about IAM role permissions, see Policies and permissions in IAM.

Deploy a Pre-Trained Model Directly to a SageMaker Endpoint

You can deploy a built-in algorithm or pre-trained model to a SageMaker endpoint in just a few lines of code using the SageMaker Python SDK.

First, find the model ID for the model of your choice in the Built-in Algorithms with pre-trained Model Table.

Low-code deployment with the JumpStartModel class

stable

Using the model ID, define your model as a JumpStart model. Use the `deploy` method to automatically deploy your model for inference. In this example, we use the FLAN-T5 XL model from HuggingFace.

```
from sagemaker.jumpstart.model import JumpStartModel

model_id = "huggingface-text2text-flan-t5-xl"
my_model = JumpStartModel(model_id=model_id)
predictor = my_model.deploy()
```

You can then run inference with the deployed model using the `predict` method.

```
question = "What is Southern California often abbreviated as?"
response = predictor.predict(question)
print(response)
```

Note

This example uses the foundation model FLAN-T5 XL, which is suitable for a wide range of text generation use cases including question answering, summarization, chatbot creation, and more. For more information about model use cases, see Choose a foundation model in the *Amazon SageMaker Developer Guide*.

For more information about the `JumpStartModel` class and its parameters, see JumpStartModel.

Additional low-code deployment utilities

You can optionally include specific model versions or instance types when deploying a pretrained model using the `JumpStartModel` class. All JumpStart models have a default instance type. Retrieve the default deployment instance type using the following code:

```
from sagemaker import instance_types

instance_type = instance_types.retrieve_default(
    model_id=model_id,
    model_version=model_version,
    scope="inference")
print(instance_type)
```

See all supported instance types for a given JumpStart model with the `instance_types.retrieve()` method.

To check valid data input and output formats for inference, you can use the `retrieve_options()` method from the `Serializers` and `Deserializers` classes.

```
print(sagemaker.serializers.retrieve_options(model_id=model_id,
model_version=model_version))
print(sagemaker.deserializers.retrieve_options(model_id=model_id,
model_version=model_version))
```

Similarly, you can use the `retrieve_options()` method to check the supported content and accept types for a model.

```
print(sagemaker.content_types.retrieve_options(model_id=model_id,
model_version=model_version))
print(sagemaker.accept_types.retrieve_options(model_id=model_id,
model_version=model_version))
```

For more information about utilities, see Utility APIs.

Deploy a pre-trained model using the SageMaker Model class

In this section, you learn how to take a pre-trained model and deploy it directly to a SageMaker Endpoint and understand what happens behind the scenes if you deployed your model as a `JumpStartModel`. The following assumes familiarity with SageMaker models and their deploy functions.

To begin, select a `model_id` and `version` from the pre-trained models table, as well as a model scope of either "inference" or "training". For this example, you use a pre-trained model, so select "inference" for your model scope. Use the utility functions to retrieve the URI of each of the three components you need to continue.

```python
from sagemaker import image_uris, model_uris, script_uris

model_id, model_version = "tensorflow-tc-bert-en-cased-L-12-H-768-A-12-2", "1.0.0"
instance_type, instance_count = "ml.m5.xlarge", 1

# Retrieve the URIs of the JumpStart resources
base_model_uri = model_uris.retrieve(
    model_id=model_id, model_version=model_version, model_scope="inference"
)
script_uri = script_uris.retrieve(
    model_id=model_id, model_version=model_version, script_scope="inference"
)
image_uri = image_uris.retrieve(
    region=None,
    framework=None,
    image_scope="inference",
    model_id=model_id,
    model_version=model_version,
    instance_type=instance_type,
)
```

Next, pass the URIs and other key parameters as part of a new SageMaker Model class. The `entry_point` is a JumpStart script named `inference.py`. SageMaker handles the implementation of this script. You must use this value for model inference to be successful. For more information about the Model class and its parameters, see Model.

```python
from sagemaker.model import Model
from sagemaker.predictor import Predictor
from sagemaker.session import Session

# Create the SageMaker model instance
model = Model(
    image_uri=image_uri,
    model_data=base_model_uri,
    source_dir=script_uri,
    entry_point="inference.py",
    role=Session().get_caller_identity_arn(),
    predictor_cls=Predictor,
    enable_network_isolation=True,
)
```

Save the output from deploying the model to a variable named `predictor`. The predictor is used to make queries on the SageMaker endpoint. Currently, the generic `model.deploy` call requires the `predictor_cls` parameter to define the predictor class. Pass in the default SageMaker Predictor class for this parameter. Deployment may take about 5 minutes.

```
predictor = model.deploy(
    initial_instance_count=instance_count,
    instance_type=instance_type,
)
```

Because the model and script URIs are distributed by SageMaker JumpStart, the endpoint, endpoint config and model resources will be prefixed with `sagemaker-jumpstart`. Refer to the model `Tags` to inspect the model artifacts involved in the model creation.

Perform inference

Finally, use the `predictor` instance to query your endpoint. For `catboost-classification-model`, for example, the predictor accepts a csv. For more information about how to use the predictor, see the Appendix.

```
predictor.predict("this is the best day of my life", {"ContentType": "application/x-
text"})
```

Deploy a pre-trained model using the SageMaker ModelBuilder class

Preparing your model for deployment on a SageMaker endpoint can take multiple steps, including choosing a model image, setting up the endpoint configuration, coding your serialization and deserialization functions to transfer data to and from server and client, identifying model dependencies, and uploading them to S3. SageMaker Modelbuilder can reduce the complexity of initial setup and deployment to help you create a SageMaker-deployable model in a single step. For an in-depth explanation of `ModelBuilder` and its supporting classes and examples, you can also refer to Create a Model in Amazon SageMaker Studio with ModelBuilder.

Build your model with ModelBuilder

`ModelBuilder` takes a framework model (such as XGBoost or PyTorch) or an inference specification (as discussed in the following sections) and converts it into a SageMaker-deployable model. `ModelBuilder` provides a `build` function that generates the artifacts for deployment. The model artifact generated is specific to the model server, which you can specify as one of the inputs. For more details about the `ModelBuilder` class, see ModelBuilder.

At minimum, the model builder expects a model, input, output and the role. In code example, `ModelBuilder` is called with a framework model and an instance `SchemaBuilder` with minimum arguments (to infer the corresponding functions for serializing

stable

and deserializing the endpoint input and output).

```
from sagemaker.serve.builder.model_builder import ModelBuilder
from sagemaker.serve.builder.schema_builder import SchemaBuilder

model_builder = ModelBuilder(
    model=model,  # xgboost or pytorch model in memory
    schema_builder=SchemaBuilder(input, output), # "SchemaBuilder" (more details below)
which will use the sample test input and output objects to infer the serialization needed.
    role_arn="arn:aws:iam::<account-id>:role/service-role/<role-name>", # Pass in the role
arn or update intelligent defaults.
)
```

The following code sample invokes `ModelBuilder` with an `InferenceSpec` instance instead of a model, and includes additional customization. See the following sections for details about `InferenceSpec`.

```
model_builder = ModelBuilder(
    mode=Mode.LOCAL_CONTAINER,
    model_path=resnet_model_dir,
    inference_spec=my_inference_spec,
    schema_builder=SchemaBuilder(input, output),
    role_arn=execution_role,
    dependencies={"auto": False, "custom": ["-e git+https://github.com/luca-medeiros/lang-
segment-anything.git#egg=lang-sam"],}
)
```

For example notebooks that demonstrate the use of `ModelBuilder`, see ModelBuilder examples.

Bring your own container (BYOC)

If you want to bring your own container that is extended from a SageMaker container, you can also specify the image URI as shown in the following example. It is also advised that you identify the model server which corresponds to the image using the `model_server` argument.

```
model_builder = ModelBuilder(
    model=model,
    model_server=ModelServer.TORCHSERVE,
    schema_builder=SchemaBuilder(X_test, y_pred),
    image_uri="123123123123.dkr.ecr.ap-southeast-2.amazonaws.com/byoc-image:xgb-1.7-1")
)
```

Define serialization and deserialization methods with SchemaBuil

stable

When invoking a SageMaker endpoint, the data is sent through HTTP payloads with different MIME types. For example, an image sent to the endpoint for inference needs to be converted to bytes by the client and sent through HTTP payload to the endpoint. The endpoint deserializes the bytes before model prediction, and serializes the prediction to bytes that are sent back through the HTTP payload to the client. The client performs deserialization to convert the bytes data back to the expected data format, such as JSON.

When you supply sample input and output to `SchemaBuilder`, the schema builder generates the corresponding marshalling functions for serializing and deserializing the input and output. You can further customize your serialization functions with `CustomPayloadTranslator`, but for most cases, a simple serializer such as the following works:

```
input = "How is the demo going?"
output = "Comment la démo va-t-elle?"
schema = SchemaBuilder(input, output)
```

For further details about `SchemaBuilder`, refer to SchemaBuilder. For sample notebooks that demonstrate the use of `SchemaBuilder`, see the example notebooks in ModelBuilder examples.

The following code snippet outlines an example where you want to customize both serialization and deserialization functions on the client and server sides. You can define your own request and response translators with `CustomPayloadTranslator` and pass them to `SchemaBuilder`.

```
from sagemaker.serve import CustomPayloadTranslator

# request translator
class MyRequestTranslator(CustomPayloadTranslator):
    # This function converts the payload to bytes - happens on client side
    def serialize_payload_to_bytes(self, payload: object) -> bytes:
        # converts the input payload to bytes
        ... ...
        return  //return object as bytes

    # This function converts the bytes to payload - happens on server side
    def deserialize_payload_from_stream(self, stream) -> object:
        # convert bytes to in-memory object
        ... ...
        return //return in-memory object

# response translator
class MyResponseTranslator(CustomPayloadTranslator):
    # This function converts the payload to bytes - happens on server side
    def serialize_payload_to_bytes(self, payload: object) -> bytes:
        # converts the response payload to bytes
        ... ...
        return //return object as bytes

    # This function converts the bytes to payload - happens on client side
    def deserialize_payload_from_stream(self, stream) -> object:
        # convert bytes to in-memory object
        ... ...
        return //return in-memory object
```

You pass the sample input and output, along with the custom translators, to the `SchemaBuilder` object.

```
my_schema = SchemaBuilder(
    sample_input=image,
    sample_output=output,
    input_translator=MyRequestTranslator(),
    output_translator=MyResponseTranslator()
)
```

For further details about `CustomPayloadTranslator`, refer to CustomPayloadTranslator. For sample notebooks that demonstrate the use of `CustomPayloadTranslator`, see the example notebooks in ModelBuilder examples.

Load the model with a custom function using InferenceSpec

`InferenceSpec` enables you to customize how the model is loaded and how it h

stable

inference requests, thus bypassing the default loading mechanisms. The `invok`
customized according to how the model should pre and postprocesses incoming requests. This

customization ensures that inference works correctly for the model. The following example uses `InferenceSpec` to generate a model with the HuggingFace pipeline. For further details about `InferenceSpec`, refer to InferenceSpec.

```
from sagemaker.serve.spec.inference_spec import InferenceSpec
from transformers import pipeline

class MyInferenceSpec(InferenceSpec):
    def load(self, model_dir: str):
        return pipeline("translation_en_to_fr", model="t5-small")

    def invoke(self, input, model):
        return model(input)

inf_spec = MyInferenceSpec()

model_builder = ModelBuilder(
    inference_spec=my_inference_spec,
    schema_builder=SchemaBuilder(X_test, y_pred)
)
```

For sample notebooks that demonstrate the use of `InferenceSpec`, see the example notebooks in ModelBuilder examples.

Build your model and deploy

Call the `build` function to create your deployable model. This step creates a model artifact in your working directory with the code necessary to create your schema, run serialization and deserialization of inputs and outputs, and execute other user-specified custom logic.

As an integrity check, SageMaker packages and pickles the necessary files for deployment as part of the `ModelBuilder` *build* function. At the same time, SageMaker also creates HMAC signing for the pickle file and adds the secret key in the `CreateModel` API as an environment variable used during *deploy* or *create*. This environment variable is used during endpoint launch to validate the integrity of the pickle file.

```
# Build the model according to the model server specification and save it as files in the working directory
model = model_builder.build()
```

Deploy your model with the model's existing `deploy` method. In this step, SageMaker sets up an endpoint to host your model as it starts making predictions on incoming reques

stable

constructed from `ModelBuilder` enables live logging during deployment as an a

```
predictor = model.deploy(
    initial_instance_count=1,
    instance_type="ml.c6i.xlarge"
)
```

Use ModelBuilder in local mode

You can deploy your model locally by using the `mode` argument to switch between local testing and deploying to a SageMaker endpoint. This example demonstrates this procedure with an XGBoost model trained locally or in SageMaker. You need the store the model artifacts in the working directory, as shown in the following snippet:

```
model = XGBClassifier()
model.fit(X_train, y_train)
model.save_model(model_dir + "/my_model.xgb")
```

Pass the model object, a `SchemaBuilder` instance, and set mode to `Mode.LOCAL_CONTAINER`, as shown in the following snippet. When you invoke the `build` function, `ModelBuilder` automatically indentifies the supported framework container and scans for dependencies.

```
model_builder_local = ModelBuilder(
    model=model,
    schema_builder=SchemaBuilder(X_test, y_pred),
    role_arn=execution_role,
    mode=Mode.LOCAL_CONTAINER
)
xgb_local_builder = model_builder_local.build()
```

Call the `deploy` function to deploy locally, as shown in the following snippet. If you specified parameters for instance type or count, these arguments are ignored.

```
predictor_local = xgb_local_builder.deploy()
```

Deploy traditional models to SageMaker Endpoints

The following examples show how to use `ModelBuilder` to deploy traditional machine learning models. Note: Since large models are often in the tens of gigabytes, it is recomı
a directory within a volume with sufficient space. Then pass this directory to th

stable

XGBoost models. You can deploy the XGBoost model from the previous example to a SageMaker endpoint by changing the mode parameter when creating the `ModelBuilder` object.

```
model_builder = ModelBuilder(
    model=model,
    schema_builder=SchemaBuilder(sample_input=sample_input, sample_output=sample_output),
    role_arn=execution_role,
    mode=Mode.SAGEMAKER_ENDPOINT
)
xgb_builder = model_builder.build()
predictor = xgb_builder.deploy(
    instance_type='ml.c5.xlarge',
    initial_instance_count=1
)
```

For a sample notebook that demonstrates using `ModelBuilder` to build a XGBoost model, see XGBoost example.

Triton models. You can use `ModelBuilder` to serve PyTorch models on a Triton inference server. Specify the `model_server` parameter as `ModelServer.TRITON`, pass a model, and include a `SchemaBuilder` object which requires sample inputs and outputs from the model. The following snippet shows an example.

```
model_builder = ModelBuilder(
    model=model,
    schema_builder=SchemaBuilder(sample_input=sample_input, sample_output=sample_output),
    role_arn=execution_role,
    model_server=ModelServer.TRITON,
    mode=Mode.SAGEMAKER_ENDPOINT
)

triton_builder = model_builder.build()

predictor = triton_builder.deploy(
    instance_type='ml.g4dn.xlarge',
    initial_instance_count=1
)
```

For a sample notebook that demonstrates using `ModelBuilder` to build a Triton model, see Triton example.

Hugging Face models. This example demonstrates how to deploy a pre-trained transformer model provided by Hugging Face to SageMaker. Since this implementation use stable
Face pipeline to load the model, you need to create a custom inference spec fo .

```
class MyInferenceSpec(InferenceSpec):
    def load(self, model_dir: str):
        return pipeline("translation_en_to_fr", model="t5-small")

    def invoke(self, input, model):
        return model(input)

inf_spec = MyInferenceSpec()
```

Define the input and output of the inference workload in the `SchemaBuilder` object.

```
value: str = "Girafatron is obsessed with giraffes, the most glorious animal on the face 
of this Earth. Giraftron believes all other animals are irrelevant when compared to the 
glorious majesty of the giraffe.\nDaniel: Hello, Girafatron!\nGirafatron:"
schema = SchemaBuilder(
    value,
    {"generated_text": "Girafatron is obsessed with giraffes, the most glorious animal on 
the face of this Earth. Giraftron believes all other animals are irrelevant when compared 
to the glorious majesty of the giraffe.\\nDaniel: Hello, Girafatron!\\nGirafatron: Hi, 
Daniel. I was just thinking about how magnificent giraffes are and how they should be 
worshiped by all.\\nDaniel: You and I think alike, Girafatron. I think all animals should 
be worshipped! But I guess that could be a bit impractical...\\nGirafatron: That\'s true. 
But the giraffe is just such an amazing creature and should always be respected!\\nDaniel: 
Yes! And the way you go on about giraffes, I could tell you really love 
them.\\nGirafatron: I\'m obsessed with them, and I\'m glad to hear you noticed!\\nDaniel: 
I\'"})
```

Create the `ModelBuilder` object and deploy the model onto a SageMaker endpoint.

```
builder = ModelBuilder(
    inference_spec=inf_spec,
    mode=Mode.SAGEMAKER_ENDPOINT,  # you can change it to Mode.LOCAL_CONTAINER for local 
testing
    schema_builder=schema,
    image_uri="123123123123.dkr.ecr.us-west-2.amazonaws.com/huggingface-pytorch-
inference:2.0.0-transformers4.28.1-gpu-py310-cu118-ubuntu20.04-v1.0"
)
model = builder.build(
    role_arn=execution_role,
    sagemaker_session=sagemaker_session,
)
predictor = model.deploy(
    initial_instance_count=1,
    instance_type='ml.g5.2xlarge'
)
```

For a sample notebook that demonstrates using `ModelBuilder` to build a Hugging Face model, see Hugging Face example.

Deploy foundation models to SageMaker Endpoints

The following examples show how to use `ModelBuilder` to deploy foundation models.

Hugging Face Hub. To deploy a foundation model from Hugging Face Hub, pass the pre-trained model ID. The following code snippet deploys the themeta-llama/Llama-2-7b-hf model locally. You can change the mode to `Mode.SAGEMAKER_ENDPOINT` to deploy to a SageMaker endpoint.

```
model_dir = "/home/ec2-user/SageMaker/LoadTestResources/meta-llama2-7b", #local path where
artifacts are saved
!mkdir -p {model_dir}

llm_hf_working_dir = str(Path(model_dir).resolve())

model_builder = ModelBuilder(
    model="meta-llama/Llama-2-7b-hf",
    schema_builder=SchemaBuilder(sample_input, sample_output),
    model_path=llm_hf_working_dir,
    mode=Mode.LOCAL_CONTAINER,
    env_vars={
        # Llama 2 is a gated model and requires a Hugging Face Hub token.
        "HUGGING_FACE_HUB_TOKEN": "<YourHuggingFaceToken>"

    }
)
model = model_builder.build()
local_predictor = model.deploy()
```

For gated models on Hugging Face Hub, request access and pass the associated key as the environment variable *HUGGING_FACE_HUB_TOKEN*. Some Hugging Face models may require trusting of remote code, so set *HF_TRUST_REMOTE_CODE* as an environment variable.

A feature of `ModelBuilder` is the ability to run local tuning on the container when you use *LOCAL_CONTAINER* mode. In this case `ModelBuilder` tunes the parameter(s) for the underlying model server. This feature can be used by executing *tuned_model=model.tune()*. Before running *tune*, clean up other containers running locally or else you might see an “address already in use” error.

For a sample notebook that demonstrates using `ModelBuilder` to build a Hugging Face Hub model, see Hugging Face Hub example.

JumpStart. JumpStart also offers a number of pre-trained foundation models. Again, the model ID is required. Deploying a JumpStart model to a SageMaker endpoint is straigh stable
shown in the following example:

```
model_builder = ModelBuilder(
    model="huggingface-llm-falcon-7b-bf16",
    schema_builder=SchemaBuilder(sample_input, sample_output),
    role_arn=execution_role
)

sm_ep_model = model_builder.build()

predictor = sm_ep_model.deploy()
```

For a list of available JumpStart model IDs, see Built-in Algorithms with pre-trained Model Table.

For a sample notebook that demonstrates using `ModelBuilder` to build a JumpStart model, see JumpStart example.

ModelBuilder examples

For example notebooks that demonstrate the use of `ModelBuilder` and its supporting classes, as well as model creation of traditional and foundation models, see the following links:

- Pytorch example
- XGBoost example
- Triton example
- Hugging Face example
- Hugging Face Hub example
- JumpStart example

Fine-tune a Model and Deploy to a SageMaker Endpoint

In this section, you initiate a training job to further train one of the pre-trained models for your use case, then deploy it to a SageMaker Endpoint for inference. This lets you fine tune the model for your use case with your custom dataset. The following assumes familiarity with SageMaker training jobs and their architecture.

Low-code fine-tuning with the JumpStartEstimator class

You can fine-tune a built-in algorithm or pre-trained model in just a few lines of code using the SageMaker Python SDK.

First, find the model ID for the model of your choice in the Built-in Algorithms with pre-trained Model Table.

Using the model ID, define your training job as a JumpStart estimator. Run `estimator.fit()` on your model, pointing to the training data to use for fine-tuning. Then, use the `deploy` method to automatically deploy your model for inference. In this example, we use the GPT-J 6B model from HuggingFace.

```
from sagemaker.jumpstart.estimator import JumpStartEstimator

model_id = "huggingface-textgeneration1-gpt-j-6b"
estimator = JumpStartEstimator(model_id=model_id)
estimator.fit(
    {"train": training_dataset_s3_path, "validation": validation_dataset_s3_path}
)
predictor = estimator.deploy()
```

You can then run inference with the deployed model using the `predict` method.

```
question = "What is Southern California often abbreviated as?"
response = predictor.predict(question)
print(response)
```

Note

This example uses the foundation model GPT-J 6B, which is suitable for a wide range of text generation use cases including question answering, named entity recognition, summarization, and more. For more information about model use cases, see Choose a foundation model in the *Amazon SageMaker Developer Guide*.

You can optionally specify model versions or instance types when creating your `JumpStartEstimator`. For more information about the `JumpStartEstimator` class and its parameters, see JumpStartEstimator.

Additional low-code training utilities

You can optionally include specific model versions or instance types when fine-tuning a pretrained model using the `JumpStartEstimator` class. All JumpStart models have a default instance type. Retrieve the default training instance type using the following code:

```python
from sagemaker import instance_types

instance_type = instance_types.retrieve_default(
    model_id=model_id,
    model_version=model_version,
    scope="training")
print(instance_type)
```

See all supported instance types for a given JumpStart model with the `instance_types.retrieve()` method.

To check the default hyperparameters used for training, you can use the `retrieve_default()` method from the `hyperparameters` class.

```python
from sagemaker import hyperparameters

my_hyperparameters = hyperparameters.retrieve_default(model_id=model_id,
model_version=model_version)
print(my_hyperparameters)

# Optionally override default hyperparameters for fine-tuning
my_hyperparameters["epoch"] = "3"
my_hyperparameters["per_device_train_batch_size"] = "4"

# Optionally validate hyperparameters for the model
hyperparameters.validate(model_id=model_id, model_version=model_version,
hyperparameters=my_hyperparameters)
```

You can also check the default metric definitions:

```python
print(metric_definitions.retrieve_default(model_id=model_id, model_version=model_version))
```

For more information about inference and utilities, see Inference APIs and Utility APIs.

Fine-tune a pre-trained model on a custom dataset using the SageMaker Estimator class

To begin, select a `model_id` and `version` from the pre-trained models table, as well as a model scope. In this case, you begin by using "training" as the model scope. Use the utility functions to retrieve the URI of each of the three components you need to continue. The HuggingFace model in this example requires a GPU instance, so use the `ml.p3.2xlarge` instance ty stable complete list of available SageMaker instance types, see the SageMaker On-De Table and select 'Training'.

```python
from sagemaker import image_uris, model_uris, script_uris

model_id, model_version = "huggingface-spc-bert-base-cased", "1.0.0"
training_instance_type = "ml.p3.2xlarge"
inference_instance_type = "ml.p3.2xlarge"
instance_count = 1

# Retrieve the JumpStart base model S3 URI
base_model_uri = model_uris.retrieve(
    model_id=model_id, model_version=model_version, model_scope="training"
)

# Retrieve the training script and Docker image
training_script_uri = script_uris.retrieve(
    model_id=model_id, model_version=model_version, script_scope="training"
)
training_image_uri = image_uris.retrieve(
    region=None,
    framework=None,
    image_scope="training",
    model_id=model_id,
    model_version=model_version,
    instance_type=training_instance_type,
)
```

Next, use the model resource URIs to create an `Estimator` and train it on a custom training dataset. You must specify the S3 path of your custom training dataset. The Estimator class requires an `entry_point` parameter. In this case, SageMaker uses “transfer_learning.py”. The training job fails to execute if this value is not set.

```python
from sagemaker.estimator import Estimator
from sagemaker.session import Session
from sagemaker import hyperparameters

# URI of your training dataset
training_dataset_s3_path = "s3://jumpstart-cache-prod-us-west-2/training-
datasets/spc/data.csv"

# Get the default JumpStart hyperparameters
default_hyperparameters = hyperparameters.retrieve_default(
    model_id=model_id,
    model_version=model_version,
)
# [Optional] Override default hyperparameters with custom values
default_hyperparameters["epochs"] = "1"

# Create your SageMaker Estimator instance
estimator = Estimator(
    image_uri=training_image_uri,
    source_dir=training_script_uri,
    model_uri=base_model_uri,
    entry_point="transfer_learning.py",
    role=Session().get_caller_identity_arn(),
    hyperparameters=default_hyperparameters,
    instance_count=instance_count,
    instance_type=training_instance_type,
    enable_network_isolation=True,
)

# Specify the S3 location of training data for the training channel
estimator.fit(
    {
        "training": training_dataset_s3_path,
    }
)
```

While the model is fitting to your training dataset, you will see console output that reflects the progress the training job is making. This gives more context about the training job, including the "transfer_learning.py" script. Model fitting takes a significant amount of time. The time that it takes varies depending on the hyperparameters, dataset, and model you use and can range from 15 minutes to 12 hours.

Deploy your trained model to a SageMaker Endpoint

Now that you've created your training job, use your `estimator` instance to create a SageMaker Endpoint that you can query for prediction. For an in-depth explanation of this process, see Deploy a Pre-Trained Model Directly to a SageMaker Endpoint.

Note: If you do not pin the model version (i.e. `_uris.retrieve(model_id="model_id" model_version="*")`), there is a chance that you pick up a different version of the script or image for deployment than you did for training. This edge case would arise if there was a release of a new version of this model in the time it took your model to train.

```
from sagemaker.utils import name_from_base

# Retrieve the inference script and Docker image
deploy_script_uri = script_uris.retrieve(
    model_id=model_id, model_version=model_version, script_scope="inference"
)
deploy_image_uri = image_uris.retrieve(
    region=None,
    framework=None,
    image_scope="inference",
    model_id=model_id,
    model_version=model_version,
    instance_type=training_instance_type,
)

# Use the estimator from the previous step to deploy to a SageMaker endpoint
endpoint_name = name_from_base(f"{model_id}-transfer-learning")

predictor = estimator.deploy(
    initial_instance_count=instance_count,
    instance_type=inference_instance_type,
    entry_point="inference.py",
    image_uri=deploy_image_uri,
    source_dir=deploy_script_uri,
    endpoint_name=endpoint_name,
    enable_network_isolation=True,
)
```

Perform inference on a SageMaker Endpoint

Finally, use the `predictor` instance to query your endpoint. For `huggingface-spc-bert-base-cased`, the predictor accepts an array of strings. For more information about how to use the predictor, see the Appendix.

```
import json

data = ["this is the best day of my life", "i am tired"]

predictor.predict(json.dumps(data).encode("utf-8"), {"ContentType": "application/list-text"})
```

Built-in Components

The following section provides information about the main components of built-in algorithms including pretrained models, model scripts, and model images.

Pre-trained models

SageMaker maintains a model zoo of over 600 models from popular open source model hubs, such as TensorFlow Hub, Pytorch Hub, and HuggingFace. You can use the SageMaker Python SDK to fine-tune a model on your own dataset or deploy it directly to a SageMaker endpoint for inference.

Model artifacts are stored as tarballs in an S3 bucket. Each model is versioned and contains a unique ID which can be used to retrieve the model URI. The following information describes the `model_id` and `model_version` needed to retrieve the URI.

- `model_id`: A unique identifier for the JumpStart model.
- `model_version`: The version of the specifications for the model. To use the latest version, enter `"*"`. This is a required parameter.

To retrieve a model, first select a `model ID` and `version` from the available models.

```
model_id, model_version = "huggingface-spc-bert-base-cased", "1.0.0"
scope = "training" # or "inference"
```

Then use those values to retrieve the model as follows.

```
from sagemaker import model_uris

model_uri = model_uris.retrieve(
    model_id=model_id, model_version=model_version, model_scope=scope
)
```

Model scripts

To adapt pre-trained models for SageMaker, a custom script is needed to perform training or inference. SageMaker maintains a suite of scripts used for each of the models in the S3 bucket, which can be accessed using the SageMaker Python SDK Use the `model_id` and `version` of the corresponding model to retrieve the related script as follows.

```
from sagemaker import script_uris

script_uri = script_uris.retrieve(
    model_id=model_id, model_version=model_version, script_scope=scope
)
```

stable

Model images

A Docker image is required to perform training or inference on all SageMaker models. SageMaker relies on Docker images from the following repos https://github.com/aws/deep-learning-containers, https://github.com/aws/sagemaker-xgboost-container, and https://github.com/aws/sagemaker-scikit-learn-container. Use the `model_id` and `version` of the corresponding model to retrieve the related image as follows. You can also use the `instance_types` utility to retrieve and use the default instance type for the model.

```
from sagemaker import image_uris, instance_types

 instance_type = instance_types.retrieve_default(
     model_id=model_id,
     model_version=model_version,
     scope=scope
 )

 image_uri = image_uris.retrieve(
    region=None,
    framework=None,
    image_scope=scope,
    model_id=model_id,
    model_version=model_version,
    instance_type=instance_type,
 )
```

Appendix

To use the `predictor` class successfully, you must provide a second parameter which contains options that the predictor uses to query your endpoint. This argument must be a `dict` with a value `ContentType` that refers to the input type for this model. The following is a list of available machine learning tasks and their corresponding values.

The `identifier` column refers to the segment of the model ID that corresponds to the model task. For example, `huggingface-spc-bert-base-cased` has a `spc` identifier, which means that it is a Sentence Pair Classification model and requires a ContentType of `application/list-text`.

Task	Identifier	ContentType
Image Classification	ic	"application/x-image"
Object Detection	od, od1	"application/x-image"
Semantic Segmentation	semseg	"application/x-image"
Instance Segmentation	is	"application/x-image"

stable

Text Classification	tc	“application/x-text”
Sentence Pair Classification	spc	“application/list-text”
Extractive Question Answering	eqa	“application/list-text”
Text Generation	textgeneration	“application/x-text”
Image Classification Embedding	icembedding	“application/x-image”
Text Classification Embedding	tcembedding	“application/x-text”
Named-entity Recognition	ner	“application/x-text”
Text Summarization	summarization	“application/x-text”
Text Translation	translation	“application/x-text”
Tabular Regression	regression	“text/csv”
Tabular Classification	classification	“text/csv”

SageMaker Automatic Model Tuning

All of the estimators can be used with SageMaker Automatic Model Tuning, which performs hyperparameter tuning jobs. A hyperparameter tuning job finds the best version of a model by running many training jobs on your dataset using the algorithm with different values of hyperparameters within ranges 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. If you’re not using an Amazon SageMaker built-in algorithm, then the metric is defined by a regular expression (regex) you provide. The hyperparameter tuning job parses the training job’s logs to find metrics that match the regex you defined. For more information about SageMaker Automatic Model Tuning, see AWS documentation.

The SageMaker Python SDK contains a `HyperparameterTuner` class for creating and interacting with hyperparameter training jobs. Here is a basic example of how to use it:

stable

```
from sagemaker.tuner import HyperparameterTuner, ContinuousParameter

# Configure HyperparameterTuner
my_tuner = HyperparameterTuner(estimator=my_estimator,  # previously-configured Estimator
object
                               objective_metric_name='validation-accuracy',
                               hyperparameter_ranges={'learning-rate':
ContinuousParameter(0.05, 0.06)},
                               metric_definitions=[{'Name': 'validation-accuracy',
'Regex': 'validation-accuracy=(\d\.\d+)'}],
                               max_jobs=100,
                               max_parallel_jobs=10)

# Start hyperparameter tuning job
my_tuner.fit({'train': 's3://my_bucket/my_training_data', 'test':
's3://my_bucket_my_testing_data'})

# Deploy best model
my_predictor = my_tuner.deploy(initial_instance_count=1, instance_type='ml.m4.xlarge')

# Make a prediction against the SageMaker endpoint
response = my_predictor.predict(my_prediction_data)

# Tear down the SageMaker endpoint
my_predictor.delete_endpoint()
```

This example shows a hyperparameter tuning job that creates up to 100 training jobs, running up to 10 training jobs at a time. Each training job's learning rate is a value between 0.05 and 0.06, but this value will differ between training jobs. You can read more about how these values are chosen in the AWS documentation.

A hyperparameter range can be one of three types: continuous, integer, or categorical. The SageMaker Python SDK provides corresponding classes for defining these different types. You can define up to 20 hyperparameters to search over, but each value of a categorical hyperparameter range counts against that limit.

By default, training job early stopping is turned off. To enable early stopping for the tuning job, you need to set the `early_stopping_type` parameter to `Auto`:

```
# Enable early stopping
my_tuner = HyperparameterTuner(estimator=my_estimator,  # previously-configured Estimator
object
                               objective_metric_name='validation-accuracy',
                               hyperparameter_ranges={'learning-rate':
ContinuousParameter(0.05, 0.06)},
                               metric_definitions=[{'Name': 'validation-accuracy',
'Regex': 'validation-accuracy=(\d\.\d+)'}],
                               max_jobs=100,
                               max_parallel_jobs=10,
                               early_stopping_type='Auto')
```

When early stopping is turned on, Amazon SageMaker will automatically stop a training job if it appears unlikely to produce a model of better quality than other jobs. If not using built-in Amazon SageMaker algorithms, note that, for early stopping to be effective, the objective metric should be emitted at epoch level.

If you are using an Amazon SageMaker built-in algorithm, you don't need to pass in anything for `metric_definitions`. In addition, the `fit()` call uses a list of `RecordSet` objects instead of a dictionary:

```
# Create RecordSet object for each data channel
train_records = RecordSet(...)
test_records = RecordSet(...)

# Start hyperparameter tuning job
my_tuner.fit([train_records, test_records])
```

To help attach a previously-started hyperparameter tuning job to a `HyperparameterTuner` instance, `fit()` adds the module path of the class used to create the hyperparameter tuner to the list of static hyperparameters by default. If you are using your own custom estimator class (i.e. not one provided in this SDK) and want that class to be used when attaching a hyperparamter tuning job, set `include_cls_metadata` to `True` when you call `fit` to add the module path as static hyperparameters.

There is also an analytics object associated with each `HyperparameterTuner` instance that contains useful information about the hyperparameter tuning job. For example, the `dataframe` method gets a pandas dataframe summarizing the associated training jobs:

```
# Retrieve analytics object
my_tuner_analytics = my_tuner.analytics()

# Look at summary of associated training jobs
my_dataframe = my_tuner_analytics.dataframe()
```

You can install all necessary for this feature dependencies using pip:

```
pip install 'sagemaker[analytics]' --upgrade
```

For more detailed examples of running hyperparameter tuning jobs, see:

- Using the TensorFlow estimator with hyperparameter tuning
- Bringing your own estimator for hyperparameter tuning
- Analyzing results

You can also find these notebooks in the **Hyperprameter Tuning** section of the **SageMaker Examples** section in a notebook instance. For information about using sample notebooks in a SageMaker notebook instance, see Use Example Notebooks in the AWS documentation.

For more detailed explanations of the classes that this library provides for automatic model tuning, see:

- API docs for HyperparameterTuner and parameter range classes
- API docs for analytics classes

SageMaker Asynchronous Inference

Amazon SageMaker Asynchronous Inference is a new capability in SageMaker that queues incoming requests and processes them asynchronously. This option is ideal for requests with large payload sizes up to 1GB, long processing times, and near real-time latency requirements. You can configure Asynchronous Inference scale the instance count to zero when there are no requests to process, thereby saving costs. More information about SageMaker Asynchronous Inference can be found in the AWS documentation.

To deploy asynchronous inference endpoint, you will need to create a `AsyncInferenceConfig` object. If you create `AsyncInferenceConfig` without specifying its arguments, the default `S3OutputPath` will be `s3://sagemaker-{REGION}-{ACCOUNTID}/async-endpoint-out` `NAME}`, `S3FailurePath` will be `s3://sagemaker-{REGION}-{ACCOUNTID}/async-endp` `failures/{UNIQUE-JOB-NAME}` (example shown below):

stable

```
from sagemaker.async_inference import AsyncInferenceConfig

# Create an empty AsyncInferenceConfig object to use default values
async_config = AsyncInferenceConfig()
```

Or you can specify configurations in `AsyncInferenceConfig` as you like. All of those configuration parameters are optional but if you don't specify the `output_path` or `failure_path`, Amazon SageMaker will use the default `S3OutputPath` or `S3FailurePath` mentioned above (example shown below):

```
# Specify S3OutputPath, S3FailurePath, MaxConcurrentInvocationsPerInstance and
NotificationConfig
# in the async config object
async_config = AsyncInferenceConfig(
    output_path="s3://{s3_bucket}/{bucket_prefix}/output",
    max_concurrent_invocations_per_instance=10,
    notification_config = {
        "SuccessTopic": "arn:aws:sns:aws-region:account-id:topic-name",
        "ErrorTopic": "arn:aws:sns:aws-region:account-id:topic-name",
        "IncludeInferenceResponseIn":
["SUCCESS_NOTIFICATION_TOPIC","ERROR_NOTIFICATION_TOPIC"],
    }
)
```

Then use the `AsyncInferenceConfig` in the estimator's `deploy()` method to deploy an asynchronous inference endpoint:

```
# Deploys the model that was generated by fit() to a SageMaker asynchronous inference
endpoint
async_predictor = estimator.deploy(async_inference_config=async_config)
```

After deployment is complete, it will return an `AsyncPredictor` object. To perform asynchronous inference, you first need to upload data to S3 and then use the `predict_async()` method with the s3 URI as the input. It will return an `AsyncInferenceResponse` object:

```
# Upload data to S3 bucket then use that as input
async_response = async_predictor.predict_async(input_path=input_s3_path)
```

stable

The Amazon SageMaker SDK also enables you to serialize the data and pass the payload data directly to the `predict_async()` method. For this pattern of invocation, the Amazon SageMaker SDK will upload the data to an Amazon S3 bucket under `s3://sagemaker-{REGION}-{ACCOUNTID}/async-endpoint-inputs/`.

```
# Serializes data and makes a prediction request to the SageMaker asynchronous endpoint
async_response = async_predictor.predict_async(data=data)
```

Then you can switch to other stuff and wait the inference to complete. After it is completed, you can check the result using `AsyncInferenceResponse`:

```
# Switch back to check the result
result = async_response.get_result()
```

Alternatively, if you would like to check for a result periodically and return it upon generation, use the `predict()` method

```
# Use predict() to wait for the result
response = async_predictor.predict(data=data)

# Or use Amazon S3 input path
response = async_predictor.predict(input_path=input_s3_path)
```

Clean up the endpoint and model if needed after inference:

```
# Tears down the SageMaker endpoint and endpoint configuration
async_predictor.delete_endpoint()

# Deletes the SageMaker model
async_predictor.delete_model()
```

For more details about Asynchronous Inference, see the API docs for Asynchronous Inference

SageMaker Serverless Inference

Amazon SageMaker Serverless Inference enables you to easily deploy machine learning models for inference without having to configure or manage the underlying infrastructu stable trained a model, you can deploy it to Amazon Sagemaker Serverless endpoint and then invoke the endpoint with the model to get inference results back. More information about SageMaker Serverless Inference can be found in the AWS documentation.

For using SageMaker Serverless Inference, you can either use SageMaker-provided container or Bring Your Own Container model. A step by step example for using Serverless Inference with MXNet image :

Firstly, create MXNet model

```
from sagemaker.mxnet import MXNetModel
from sagemaker.serverless import ServerlessInferenceConfig
import sagemaker

role = sagemaker.get_execution_role()

# create MXNet Model Class
model = MXNetModel(
    model_data="s3://my_bucket/pretrained_model/model.tar.gz", # path to your trained
sagemaker model
    role=role, # iam role with permissions to create an Endpoint
    entry_point="inference.py",
    py_version="py3", # Python version
    framework_version="1.6.0", # MXNet framework version
)
```

To deploy serverless endpoint, you will need to create a `ServerlessInferenceConfig`. If you create `ServerlessInferenceConfig` without specifying its arguments, the default `MemorySizeInMB` will be **2048** and the default `MaxConcurrency` will be **5** :

```
from sagemaker.serverless import ServerlessInferenceConfig

# Create an empty ServerlessInferenceConfig object to use default values
serverless_config = ServerlessInferenceConfig()
```

Or you can specify `MemorySizeInMB` and `MaxConcurrency` in `ServerlessInferenceConfig` (example shown below):

```
# Specify MemorySizeInMB and MaxConcurrency in the serverless config object
serverless_config = ServerlessInferenceConfig(
  memory_size_in_mb=4096,
  max_concurrency=10,
)
```

stable

Then use the `ServerlessInferenceConfig` in the estimator's `deploy()` method to deploy a serverless endpoint:

```
# Deploys the model that was generated by fit() to a SageMaker serverless endpoint
serverless_predictor = estimator.deploy(serverless_inference_config=serverless_config)
```

Or directly using model's `deploy()` method to deploy a serverless endpoint:

```
# Deploys the model to a SageMaker serverless endpoint
serverless_predictor = model.deploy(serverless_inference_config=serverless_config)
```

After deployment is complete, you can use predictor's `predict()` method to invoke the serverless endpoint just like real-time endpoints:

```
# Serializes data and makes a prediction request to the SageMaker serverless endpoint
response = serverless_predictor.predict(data)
```

Clean up the endpoint and model if needed after inference:

```
# Tears down the SageMaker endpoint and endpoint configuration
serverless_predictor.delete_endpoint()

# Deletes the SageMaker model
serverless_predictor.delete_model()
```

For more details about `ServerlessInferenceConfig`, see the API docs for Serverless Inference

SageMaker Batch Transform

After you train a model, you can use Amazon SageMaker Batch Transform to perform inferences with the model. Batch transform manages all necessary compute resources, including launching instances to deploy endpoints and deleting them afterward. You can read more about SageMaker Batch Transform in the AWS documentation.

If you trained the model using a SageMaker Python SDK estimator, you can invoke the estimator's `transformer()` method to create a transform job for a model basec stable job:

```
transformer = estimator.transformer(instance_count=1, instance_type='ml.m4.xlarge')
```

Alternatively, if you already have a SageMaker model, you can create an instance of the `Transformer` class by calling its constructor:

```
from sagemaker.transformer import Transformer

transformer = Transformer(model_name='my-previously-trained-model',
                          instance_count=1,
                          instance_type='ml.m4.xlarge')
```

For a full list of the possible options to configure by using either of these methods, see the API docs for Estimator or Transformer.

After you create a `Transformer` object, you can invoke `transform()` to start a batch transform job with the S3 location of your data. You can also specify other attributes of your data, such as the content type.

```
transformer.transform('s3://my-bucket/batch-transform-input')
```

For more details about what can be specified here, see API docs.

Local Mode

The SageMaker Python SDK supports local mode, which allows you to create estimators, processors, and pipelines, and deploy them to your local environment. This is a great way to test your deep learning scripts before running them in SageMaker's managed training or hosting environments. Local Mode is supported for frameworks images (TensorFlow, MXNet, Chainer, PyTorch, and Scikit-Learn) and images you supply yourself.

You can install necessary dependencies for this feature using pip.

```
pip install 'sagemaker[local]' --upgrade
```

Additionally, Local Mode also requires Docker Compose V2. Follow the guidelir https://docs.docker.com/compose/install/ to install. Make sure to have a Comp compatible with your Docker Engine installation. Check Docker Engine release notes

stable

https://docs.docker.com/engine/release-notes to find a compatible version.

Local mode configuration

The local mode uses a YAML configuration file located at `${user_config_directory}/sagemaker/config.yaml` to define the default values that are automatically passed to the `config` attribute of `LocalSession`. This is an example of the configuration, for the full schema, see sagemaker.config.config_schema.SAGEMAKER_PYTHON_SDK_LOCAL_MODE_CONFIG_SCHEMA.

```
local:
    local_code: true # Using everything locally
    region_name: "us-west-2" # Name of the region
    container_config: # Additional docker container config
        shm_size: "128M"
```

If you want to keep everything local, and not use Amazon S3 either, you can enable “local code” in one of two ways:

```
local:
  local_code: true
```

- Create a `LocalSession` or `LocalPipelineSession` (for local SageMaker pipelines) and configure it directly:

```
from sagemaker.local import LocalSession

sagemaker_session = LocalSession()
sagemaker_session.config = {'local': {'local_code': True}}

# pass sagemaker_session to your estimator or model
```

Note

If you enable “local code,” then you cannot use the `dependencies` parameter in your estimator or model.

stable

Activating local mode by `instance_type` argument

We can take the example in Using Estimators , and use either `local` or `local_gpu` as the instance type.

```
from sagemaker.mxnet import MXNet

# Configure an MXNet Estimator (no training happens yet)
mxnet_estimator = MXNet('train.py',
                        role='SageMakerRole',
                        instance_type='local',
                        instance_count=1,
                        framework_version='1.2.1')

# In Local Mode, fit will pull the MXNet container Docker image and run it locally
mxnet_estimator.fit('s3://my_bucket/my_training_data/')

# Alternatively, you can train using data in your local file system. This is only
supported in Local mode.
mxnet_estimator.fit('file:///tmp/my_training_data')

# Deploys the model that was generated by fit() to local endpoint in a container
mxnet_predictor = mxnet_estimator.deploy(initial_instance_count=1, instance_type='local')

# Serializes data and makes a prediction request to the local endpoint
response = mxnet_predictor.predict(data)

# Tears down the endpoint container and deletes the corresponding endpoint configuration
mxnet_predictor.delete_endpoint()

# Deletes the model
mxnet_predictor.delete_model()
```

If you have an existing model and want to deploy it locally, don’t specify a sagemaker_session argument to the `MXNetModel` constructor. The correct session is generated when you call `model.deploy()` .

Here is an end-to-end example:

```
import numpy
from sagemaker.mxnet import MXNetModel

model_location = 's3://mybucket/my_model.tar.gz'
code_location = 's3://mybucket/sourcedir.tar.gz'
s3_model = MXNetModel(model_data=model_location, role='SageMakerRole',
                      entry_point='mnist.py', source_dir=code_location)

predictor = s3_model.deploy(initial_instance_count=1, instance_type='local')
data = numpy.zeros(shape=(1, 1, 28, 28))
predictor.predict(data)

# Tear down the endpoint container and delete the corresponding endpoint configuration
predictor.delete_endpoint()

# Deletes the model
predictor.delete_model()
```

If you don't want to deploy your model locally, you can also choose to perform a Local Batch Transform Job. This is useful if you want to test your container before creating a Sagemaker Batch Transform Job. Note that the performance will not match Batch Transform Jobs hosted on SageMaker but it is still a useful tool to ensure you have everything right or if you are not dealing with huge amounts of data.

Here is an end-to-end example:

```
from sagemaker.mxnet import MXNet

mxnet_estimator = MXNet('train.py',
                        role='SageMakerRole',
                        instance_type='local',
                        instance_count=1,
                        framework_version='1.2.1')

mxnet_estimator.fit('file:///tmp/my_training_data')
transformer = mxnet_estimator.transformer(1, 'local', assemble_with='Line', max_payload=1)
transformer.transform('s3://my/transform/data', content_type='text/csv',
split_type='Line')
transformer.wait()

# Deletes the SageMaker model
transformer.delete_model()
```

Local pipelines

To put everything together, you can use local pipelines to execute various SageMaker iobs in succession. Pipelines can be executed locally by providing a `LocalPipelineSess` stable pipeline's and pipeline steps' initializer. `LocalPipelineSession` inherits from `LocalSession`. The difference is `LocalPipelineSession` captures the job input step arguments and passes it to the pipeline object instead of executing the job. This behavior is similar to that of PipelineSession.

Here is an end-to-end example:

```
from sagemaker.workflow.pipeline import Pipeline
from sagemaker.workflow.steps import TrainingStep, TransformStep
from sagemaker.workflow.model_step import ModelStep
from sagemaker.workflow.pipeline_context import LocalPipelineSession
from sagemaker.mxnet import MXNet
from sagemaker.model import Model
from sagemaker.inputs import TranformerInput
from sagemaker.transformer import Transformer

session = LocalPipelineSession()
mxnet_estimator = MXNet('train.py',
                        role='SageMakerRole',
                        instance_type='local',
                        instance_count=1,
                        framework_version='1.2.1',
                        sagemaker_session=session)

train_step_args = mxnet_estimator.fit('file:///tmp/my_training_data')

# Define training step
train_step = TrainingStep(name='local_mxnet_train', step_args=train_step_args)

model = Model(
  image_uri=inference_image_uri,
  model_data=train_step.properties.ModelArtifacts.S3ModelArtifacts,
  sagemaker_session=session,
  role='SageMakerRole'
)

# Define create model step
model_step_args = model.create(instance_type="local", accelerator_type="local")
model_step = ModelStep(
  name='local_mxnet_model',
  step_args=model_step_args
)

transformer =  Transformer(
  model_name=model_step.properties.ModelName,
  instance_type='local',
  instance_count=1,
  sagemaker_session=session
)
transform_args = transformer.transform('file:///tmp/my_transform_data')
# Define transform step
transform_step = TransformStep(name='local_mxnet_transform', step_args=transform_args)

# Define the pipeline
pipeline = Pipeline(name='local_pipeline',
                    steps=[train_step, model_step, transform_step],
                    sagemaker_session=session)

# Create the pipeline
pipeline.upsert(role_arn='SageMakerRole', description='local pipeline example')

# Start a pipeline execution
execution = pipeline.start()
```

stable

Note

Currently Pipelines Local Mode only supports the following step types: Training, Processing, Transform, Model (with Create Model arguments only), Condition, and Fail.

For detailed examples of running Docker in local mode, see:

- TensorFlow local mode example notebook.
- MXNet local mode example notebook.
- PyTorch local mode example notebook.
- Pipelines local mode example notebook.

You can also find these notebooks in the **SageMaker Python SDK** section of the **SageMaker Examples** section in a notebook instance. For information about using sample notebooks in a SageMaker notebook instance, see Use Example Notebooks in the AWS documentation.

A few important notes:

- Only one local mode endpoint can be running at a time.
- If you are using S3 data as input, it is pulled from S3 to your local environment. Ensure you have sufficient space to store the data locally.
- If you run into problems it often due to different Docker containers conflicting. Killing these containers and re-running often solves your problems.
- Local Mode requires Docker Compose and nvidia-docker2 for `local_gpu`.
- Set `USE_SHORT_LIVED_CREDENTIALS=1` if running on EC2 and you would like to use the session credentials instead of EC2 Metadata Service credentials.

Warning

Local Mode does not yet support the following:

- Distributed training for `local_gpu`
- Gzip compression, Pipe Mode, or manifest files for inputs

Secure Training and Inference with VPC

Amazon SageMaker allows you to control network traffic to and from model container instances using Amazon Virtual Private Cloud (VPC). You can configure SageMaker to use your own private VPC in order to further protect and monitor traffic.

stable

For more information about Amazon SageMaker VPC features, and guidelines for configuring your VPC, see the following documentation:

- Protect Training Jobs by Using an Amazon Virtual Private Cloud
- Protect Endpoints by Using an Amazon Virtual Private Cloud
- Protect Data in Batch Transform Jobs by Using an Amazon Virtual Private Cloud
- Working with VPCs and Subnets

You can also reference or reuse the example VPC created for integration tests: tests/integ/vpc_test_utils.py

To train a model using your own VPC, set the optional parameters `subnets` and `security_group_ids` on an `Estimator`:

```
from sagemaker.mxnet import MXNet

# Configure an MXNet Estimator with subnets and security groups from your VPC
mxnet_vpc_estimator = MXNet('train.py',
                            instance_type='ml.p2.xlarge',
                            instance_count=1,
                            framework_version='1.2.1',
                            subnets=['subnet-1', 'subnet-2'],
                            security_group_ids=['sg-1'])

# SageMaker Training Job will set VpcConfig and container instances will run in your VPC
mxnet_vpc_estimator.fit('s3://my_bucket/my_training_data/')
```

To train a model with the inter-container traffic encrypted, set the optional parameters `subnets` and `security_group_ids` and the flag `encrypt_inter_container_traffic` as `True` on an Estimator (Note: This flag can be used only if you specify that the training job runs in a VPC):

```
from sagemaker.mxnet import MXNet

# Configure an MXNet Estimator with subnets and security groups from your VPC
mxnet_vpc_estimator = MXNet('train.py',
                            instance_type='ml.p2.xlarge',
                            instance_count=1,
                            framework_version='1.2.1',
                            subnets=['subnet-1', 'subnet-2'],
                            security_group_ids=['sg-1'],
                            encrypt_inter_container_traffic=True)

# The SageMaker training job sets the VpcConfig, and training container instances run in
your VPC with traffic between the containers encrypted
mxnet_vpc_estimator.fit('s3://my_bucket/my_training_data/')
```

When you create a `Predictor` from the `Estimator` using `deploy()`, the same configurations will be set on the SageMaker Model:

```
# Creates a SageMaker Model and Endpoint using the same VpcConfig
# Endpoint container instances will run in your VPC
mxnet_vpc_predictor = mxnet_vpc_estimator.deploy(initial_instance_count=1,
                                                 instance_type='ml.p2.xlarge')

# You can also set ``vpc_config_override`` to use a different VpcConfig
other_vpc_config = {'Subnets': ['subnet-3', 'subnet-4'],
                    'SecurityGroupIds': ['sg-2']}
mxnet_predictor_other_vpc = mxnet_vpc_estimator.deploy(initial_instance_count=1,
                                                       instance_type='ml.p2.xlarge',

vpc_config_override=other_vpc_config)

# Setting ``vpc_config_override=None`` will disable VpcConfig
mxnet_predictor_no_vpc = mxnet_vpc_estimator.deploy(initial_instance_count=1,
                                                    instance_type='ml.p2.xlarge',
                                                    vpc_config_override=None)
```

Likewise, when you create `Transformer` from the `Estimator` using `transformer()`, the same VPC configurations will be set on the SageMaker Model:

```
# Creates a SageMaker Model using the same VpcConfig
mxnet_vpc_transformer = mxnet_vpc_estimator.transformer(instance_count=1,
                                                        instance_type='ml.p2.xlarge')

# Transform Job container instances will run in your VPC
mxnet_vpc_transformer.transform('s3://my-bucket/batch-transform-input')
```

Secure Training with Network Isolation (Internet-Free) Mode

You can enable network isolation mode when running training and inference on Amazon SageMaker.

For more information about Amazon SageMaker network isolation mode, see the SageMaker documentation on network isolation or internet-free mode.

To train a model in network isolation mode, set the optional parameter `enable_network_isolation` to `True` in any network isolation supported Framework Estimator.

```
# set the enable_network_isolation parameter to True
sklearn_estimator = SKLearn('sklearn-train.py',
                            instance_type='ml.m4.xlarge',
                            framework_version='0.20.0',
                            hyperparameters = {'epochs': 20, 'batch-size': 64, 'learning-
rate': 0.1},
                            enable_network_isolation=True)

# SageMaker Training Job will in the container without   any inbound or outbound network
calls during runtime
sklearn_estimator.fit({'train': 's3://my-data-bucket/path/to/my/training/data',
                        'test': 's3://my-data-bucket/path/to/my/test/data'})
```

When this training job is created, the SageMaker Python SDK will upload the files in `entry_point`, `source_dir`, and `dependencies` to S3 as a compressed `sourcedir.tar.gz` file (`'s3://mybucket/sourcedir.tar.gz'`).

A new training job channel, named `code`, will be added with that S3 URI. Before the training docker container is initialized, the `sourcedir.tar.gz` will be downloaded from S3 to the ML storage volume like any other offline input channel.

Once the training job begins, the training container will look at the offline input `code` channel to install dependencies and run the entry script. This isolates the training container, so no inbound or outbound network calls can be made.

Inference Pipelines

You can create a Pipeline for realtime or batch inference comprising of one or multiple model containers. This will help you to deploy an ML pipeline behind a single endpoint and you can have one API call perform pre-processing, model-scoring and post-processing on your data before returning it back as the response.

For this, you have to create a `PipelineModel` which will take a list of `Model` objects. Calling `deploy()` on the `PipelineModel` will provide you with an endpoint which can be invoked to perform the prediction on a data point against the ML Pipeline.

```
from sagemaker import image_uris, session
from sagemaker.model import Model
from sagemaker.pipeline import PipelineModel
from sagemaker.sparkml import SparkMLModel

xgb_image = image_uris.retrieve("xgboost", session.Session().boto_region_name,
repo_version="latest")
xgb_model = Model(model_data="s3://path/to/model.tar.gz", image_uri=xgb_image)
sparkml_model = SparkMLModel(model_data="s3://path/to/model.tar.gz", env=
{"SAGEMAKER_SPARKML_SCHEMA": schema})

model_name = "inference-pipeline-model"
endpoint_name = "inference-pipeline-endpoint"
sm_model = PipelineModel(name=model_name, role=sagemaker_role, models=[sparkml_model,
xgb_model])
```

This defines a `PipelineModel` consisting of SparkML model and an XGBoost model stacked sequentially. For more information about how to train an XGBoost model, please refer to the XGBoost notebook here.

You can also find this notebook in the **Introduction to Amazon Algorithms** section of the **SageMaker Examples** section in a notebook instance. For information about using sample notebooks in a SageMaker notebook instance, see Use Example Notebooks in the AWS documentation.

```
sm_model.deploy(initial_instance_count=1, instance_type='ml.c5.xlarge',
endpoint_name=endpoint_name)
```

This returns a predictor the same way an `Estimator` does when `deploy()` is called. Whenever you make an inference request using this predictor, you should pass the data that the first container expects and the predictor will return the output from the last container.

You can also use a `PipelineModel` to create Transform Jobs for batch transformations. Using the same `PipelineModel` `sm_model` as above:

```
# Only instance_type and instance_count are required.
transformer = sm_model.transformer(instance_type='ml.c5.xlarge',
                                   instance_count=1,
                                   strategy='MultiRecord',
                                   max_payload=6,
                                   max_concurrent_transforms=8,
                                   accept='text/csv',
                                   assemble_with='Line',
                                   output_path='s3://my-output-
bucket/path/to/my/output/data/')
# Only data is required.
transformer.transform(data='s3://my-input-bucket/path/to/my/csv/data',
                      content_type='text/csv',
                      split_type='Line')
# Waits for the Pipeline Transform Job to finish.
transformer.wait()
```

This runs a transform job against all the files under `s3://mybucket/path/to/my/csv/data`, transforming the input data in order with each model container in the pipeline. For each input file that was successfully transformed, one output file in `s3://my-output-bucket/path/to/my/output/data/` will be created with the same name, appended with '.out'. This transform job will split CSV files by newline separators, which is especially useful if the input files are large. The Transform Job assembles the outputs with line separators when writing each input file's corresponding output file. Each payload entering the first model container will be up to six megabytes, and up to eight inference requests are sent at the same time to the first model container. Because each payload consists of a mini-batch of multiple CSV records, the model containers transform each mini-batch of records.

For comprehensive examples on how to use Inference Pipelines please refer to the following notebooks:

- inference_pipeline_sparkml_xgboost_abalone.ipynb
- inference_pipeline_sparkml_blazingtext_dbpedia.ipynb

You can also find these notebooks in the **Advanced Functionality** section of the **SageMaker Examples** section in a notebook instance. For information about using sample notebooks in a SageMaker notebook instance, see Use Example Notebooks in the AWS documentation.

SageMaker Workflow

You can use the following machine learning frameworks to author, schedule and monitor SageMaker workflow.

- Airflow Workflows
  - Amazon SageMaker Operators in Apache Airflow

    - Airflow
- AWS Step Functions
- SageMaker Pipelines
    - Pipelines
- SageMaker Lineage
    - Lineage

SageMaker Model Building Pipeline

You can use Amazon SageMaker Model Building Pipelines to orchestrate your machine learning workflow.

For more information, see SageMaker Model Building Pipeline.

SageMaker Model Monitoring

You can use Amazon SageMaker Model Monitoring to automatically detect concept drift by monitoring your machine learning models.

For more information, see SageMaker Model Monitoring.

SageMaker Debugger

You can use Amazon SageMaker Debugger to automatically detect anomalies while training your machine learning models.

For more information, see SageMaker Debugger.

SageMaker Processing

You can use Amazon SageMaker Processing with “Processors” to perform data processing tasks such as data pre- and post-processing, feature engineering, data validation, and model evaluation

- Amazon SageMaker Processing
    - Background
    - Setup
    - Data Pre-Processing and Model Evaluation with scikit-learn
    - Data Processing with Spark
    - Learn More

Configuring and using defaults with the SageMaker Python SDK

The Amazon SageMaker Python SDK supports the setting of default values for AWS infrastructure primitive types. After administrators configure these defaults, they are automatically passed when SageMaker Python SDK calls supported APIs. Amazon SageMaker APIs and primitives may not have a direct correspondence to the SageMaker Python SDK abstractions that you are using. The parameters you specify are automatically passed when the SageMaker Python SDK makes calls to the API on your behalf. With the use of defaults, developers can use the SageMaker Python SDK without having to specify infrastructure parameters.

Configuration file structure

The SageMaker Python SDK uses YAML configuration files to define the default values that are automatically passed to APIs. Admins can create these configuration files and populate them with default values defined for their desired API parameters. Your configuration file should adhere to the structure outlined in the following sample config file. This config outlines some of the parameters that you can set default values for. For the full schema, see sagemaker.config.config_schema.SAGEMAKER_PYTHON_SDK_CONFIG_SCHEMA.

```
SchemaVersion: '1.0'
CustomParameters:
  AnyStringKey: 'AnyStringValue'
SageMaker:
  PythonSDK:
    Modules:
      Session:
        DefaultS3Bucket: 'default_s3_bucket'
        DefaultS3ObjectKeyPrefix: 'key_prefix'
      Estimator:
        DebugHookConfig: true
      RemoteFunction:
        Dependencies: 'path/to/requirements.txt'
        EnableInterContainerTrafficEncryption: true
        EnvironmentVariables: {'EnvVarKey': 'EnvVarValue'}
        ImageUri: '555555555555.dkr.ecr.us-west-2.amazonaws.com/my-image:latest'
        IncludeLocalWorkDir: true
        InstanceType: 'ml.m5.large'
        JobCondaEnvironment: 'your_conda_env'
        PreExecutionCommands:
          - 'command_1'
          - 'command_2'
        PreExecutionScript: 'path/to/script.sh'
        RoleArn: 'arn:aws:iam::555555555555:role/MyRole'
        S3KmsKeyId: 's3kmskeyid'
        S3RootUri: 's3://my-bucket/my-project'
        Tags:
          - Key: 'tag_key'
            Value: 'tag_value'
        VolumeKmsKeyId: 'volumekmskeyid1'
        VpcConfig:
          SecurityGroupIds:
            - 'sg123'
          Subnets:
            - 'subnet-1234'
  FeatureGroup:
    # 
https://docs.aws.amazon.com/sagemaker/latest/APIReference/API_CreateFeatureGroup.html
    OnlineStoreConfig:
      SecurityConfig:
        KmsKeyId: 'kmskeyid1'
    OfflineStoreConfig:
      S3StorageConfig:
        KmsKeyId: 'kmskeyid2'
    RoleArn: 'arn:aws:iam::555555555555:role/IMRole'
    Tags:
    - Key: 'tag_key'
      Value: 'tag_value'
  MonitoringSchedule:
  # 
https://docs.aws.amazon.com/sagemaker/latest/APIReference/API_CreateMonitoringSchedule.html
    MonitoringScheduleConfig:
      MonitoringJobDefinition:
        Environment:
          'var1': 'value1'
          'var2': 'value2'
        MonitoringOutputConfig:
          KmsKeyId: 'kmskeyid3'
        MonitoringResources:
```

stable

```
     ClusterConfig:
      VolumeKmsKeyId: 'volumekmskeyid2'
    NetworkConfig:
     EnableNetworkIsolation: true
     VpcConfig:
      SecurityGroupIds:
       - 'sg123'
      Subnets:
       - 'subnet-1234'
    RoleArn: 'arn:aws:iam::555555555555:role/IMRole'
  Tags:
  - Key: 'tag_key'
   Value: 'tag_value'
 Endpoint:
  Tags:
   - Key: "tag_key"
    Value: "tag_value"
 EndpointConfig:
 #
https://docs.aws.amazon.com/sagemaker/latest/APIReference/API_CreateEndpointConfig.html
  AsyncInferenceConfig:
   OutputConfig:
    KmsKeyId: 'kmskeyid4'
  DataCaptureConfig:
   KmsKeyId: 'kmskeyid5'
  KmsKeyId: 'kmskeyid6'
  ProductionVariants:
   - CoreDumpConfig:
     KmsKeyId: 'kmskeyid7'
  Tags:
  - Key: 'tag_key'
   Value: 'tag_value'
 AutoMLJob:
 # https://docs.aws.amazon.com/sagemaker/latest/APIReference/API_CreateAutoMLJob.html
  AutoMLJobConfig:
   SecurityConfig:
    VolumeKmsKeyId: 'volumekmskeyid3'
    VpcConfig:
     SecurityGroupIds:
      - 'sg123'
     Subnets:
      - 'subnet-1234'
  OutputDataConfig:
   KmsKeyId: 'kmskeyid8'
  RoleArn: 'arn:aws:iam::555555555555:role/IMRole'
  Tags:
  - Key: 'tag_key'
   Value: 'tag_value'
 TransformJob:
 # https://docs.aws.amazon.com/sagemaker/latest/APIReference/API_CreateTransformJob.html
  DataCaptureConfig:
   KmsKeyId: 'kmskeyid9'
  Environment:
   'var1': 'value1'
   'var2': 'value2'
  TransformOutput:
   KmsKeyId: 'kmskeyid10'
  TransformResources:
   VolumeKmsKeyId: 'volumekmskeyid4'
  Tags:
```

stable

```
    - Key: 'tag_key'
      Value: 'tag_value
  CompilationJob:
  #
https://docs.aws.amazon.com/sagemaker/latest/APIReference/API_CreateCompilationJob.html
    OutputConfig:
      # Currently not supported by the SageMaker Python SDK
      KmsKeyId: 'kmskeyid11'
    RoleArn: 'arn:aws:iam::555555555555:role/IMRole'
    # Currently not supported by the SageMaker Python SDK
    VpcConfig:
      SecurityGroupIds:
        - 'sg123'
      Subnets:
        - 'subnet-1234'
    Tags:
    - Key: 'tag_key'
      Value: 'tag_value'
  Pipeline:
  # https://docs.aws.amazon.com/sagemaker/latest/APIReference/API_CreatePipeline.html
    RoleArn: 'arn:aws:iam::555555555555:role/IMRole'
    Tags:
    - Key: 'tag_key'
      Value: 'tag_value'
  Model:
  # https://docs.aws.amazon.com/sagemaker/latest/APIReference/API_CreateModel.html
    Containers:
      - Environment:
          'var1': 'value1'
          'var2': 'value2'
    EnableNetworkIsolation: true
    ExecutionRoleArn: 'arn:aws:iam::555555555555:role/IMRole'
    VpcConfig:
      SecurityGroupIds:
        - 'sg123'
      Subnets:
        - 'subnet-1234'
    Tags:
    - Key: 'tag_key'
      Value: 'tag_value'
  ModelPackage:
    InferenceSpecification:
      Containers:
        - Environment:
            'var1': 'value1'
            'var2': 'value2'
    ValidationSpecification:
      ValidationProfiles:
        - TransformJobDefinition:
            Environment:
              'var1': 'value1'
              'var2': 'value2'
            TransformOutput:
              KmsKeyId: 'kmskeyid12'
            TransformResources:
              VolumeKmsKeyId: 'volumekmskeyid5'
      ValidationRole: 'arn:aws:iam::555555555555:role/IMRole'
  ProcessingJob:
  # https://docs.aws.amazon.com/sagemaker/latest/APIReference/API_CreateProcessingJob.html
    Environment:
```
stable

```
      'var1': 'value1'
      'var2': 'value2'
    NetworkConfig:
      EnableNetworkIsolation: true
      VpcConfig:
        SecurityGroupIds:
          - 'sg123'
        Subnets:
          - 'subnet-1234'
    ProcessingInputs:
      - DatasetDefinition:
          AthenaDatasetDefinition:
            KmsKeyId: 'kmskeyid13'
          RedshiftDatasetDefinition:
            KmsKeyId: 'kmskeyid14'
            ClusterRoleArn: 'arn:aws:iam::555555555555:role/IMRole'
    ProcessingOutputConfig:
      KmsKeyId: 'kmskeyid13'
    ProcessingResources:
      ClusterConfig:
        VolumeKmsKeyId: 'volumekmskeyid6'
    RoleArn: 'arn:aws:iam::555555555555:role/IMRole'
    Tags:
    - Key: 'tag_key'
      Value: 'tag_value'
  TrainingJob:
  # https://docs.aws.amazon.com/sagemaker/latest/APIReference/API_CreateTrainingJob.html
    EnableNetworkIsolation: true
    Environment:
      'var1': 'value1'
      'var2': 'value2'
    OutputDataConfig:
      KmsKeyId: 'kmskeyid14'
    ProfilerConfig:
      DisableProfiler: false
    ResourceConfig:
      VolumeKmsKeyId: 'volumekmskeyid7'
    RoleArn: 'arn:aws:iam::555555555555:role/IMRole'
    VpcConfig:
      SecurityGroupIds:
        - 'sg123'
      Subnets:
        - 'subnet-1234'
    Tags:
    - Key: 'tag_key'
      Value: 'tag_value'
  EdgePackagingJob:
  #
https://docs.aws.amazon.com/sagemaker/latest/APIReference/API_CreateEdgePackagingJob.html
    OutputConfig:
      KmsKeyId: 'kmskeyid15'
    RoleArn: 'arn:aws:iam::555555555555:role/IMRole'
    ResourceKey: 'resourcekmskeyid'
    Tags:
    - Key: 'tag_key'
      Value: 'tag_value'
```

stable

Configuration file locations

The SageMaker Python SDK searches for configuration files at two locations based on the platform that you are using. You can also modify the default locations by overriding them using environment variables. The following sections give information about these configuration file locations.

Default configuration file location

By default, the SageMaker Python SDK uses two configuration files. One for the admin and one for the user. Using the admin config file, admins can define a set of default values. Users can use the user configuration file to override values set in the admin configuration file, as well as set other default parameter values. Users can also set additional configuration file locations. For more information about setting additional configuration file locations, see Specify additional configuration files.

The location of your default configuration paths depends on the platform that you're using the SageMaker Python SDK on. These default locations are relative to the environment that you are using the SageMaker Python SDK on.

The following code block returns the default locations of your admin and user configuration files. These commands must be run from the environment that you're using the SageMaker Python SDK in.

Note: The directories returned by these commands may not exist. In that case, you must create these directories with the required permissions.

```
import os
from platformdirs import site_config_dir, user_config_dir

#Prints the location of the admin config file
print(os.path.join(site_config_dir("sagemaker"), "config.yaml"))

#Prints the location of the user config file
print(os.path.join(user_config_dir("sagemaker"), "config.yaml"))
```

Default Notebook instances locations

The following code sample lists the default locations of the configuration files when using the SageMaker Python SDK on Amazon SageMaker Notebook instances.

```
#Location of the admin config file
/etc/xdg/sagemaker/config.yaml

#Location of the user config file
/home/ec2-user/.config/sagemaker/config.yaml
```

Default Studio notebook locations

The following code sample lists the default locations of the configuration files when using the SageMaker Python SDK on Amazon SageMaker Studio notebooks.

```
#Location of the admin config file
/etc/xdg/sagemaker/config.yaml

#Location of the user config file
/root/.config/sagemaker/config.yaml
```

Override the configuration file location

To change the default configuration file locations used by the SageMaker Python SDK, set one or both of the following environment variables from the environment where you are using the SageMaker Python SDK. When you modify these environment variables, the SageMaker Python SDK searches for configuration files in the locations that you specify instead of the default configuration file locations.

- `SAGEMAKER_ADMIN_CONFIG_OVERRIDE` overrides the default location where the SageMaker Python SDK searches for the admin config.
- `SAGEMAKER_USER_CONFIG_OVERRIDE` overrides the default location where the SageMaker Python SDK searches for the user config.

Using these environment variables, you can set the config location to either a local config location or a config location in an Amazon S3 bucket. If a directory is provided as the path, the SageMaker Python SDK searches for the `config.yaml` file in that directory. The SageMaker Python SDK does not do any recursive searches for the file.

The following options are available if the config is saved locally.

- Local file path: `<path-to-config>/config.yaml`
- Path of the directory containing the config file : `<path-to-config>/`

The following options are available if the config is saved on Amazon S3.

stable

- S3 URI of the config file: `s3://<my-bucket>/<path-to-config>/config.yaml`

- S3 URI of the directory containing the config file: `s3://<my-bucket>/<path-to-config>/`

For example, the following example sets the default user configuration location to a local directory from within a Jupyter notebook environment.

```
import os
os.environ["SAGEMAKER_USER_CONFIG_OVERRIDE"] = "<path-to-config>"
```

If you're using Studio or a Notebook instance, you can automatically set this value for all instances with a lifecycle configuration script. For more information about lifecycle configuration scripts, see Use Lifecycle Configurations with Amazon SageMaker Studio.

Supported APIs and parameters

The following sections give information about the APIs and parameters that the SageMaker Python SDK supports setting defaults for. To set defaults for these parameters, create key/value pairs in your configuration file as shown in Configuration file structure. For the full schema, see sagemaker.config.config_schema.SAGEMAKER_PYTHON_SDK_CONFIG_SCHEMA.

List of parameters supported

In the supported APIs, only parameters for the following primitive types support setting defaults with a configuration file.

- Amazon VPC subnets and security groups
- AWS IAM Role ARNs
- AWS KMS key IDs
- Debug Hook Config
- Disable Profiler
- Enable inter-container traffic encryption
- Enable network isolation
- Environment Variables
- SageMaker Session Default S3 Bucket and Default S3 ObjectKeyPrefix
- Tags

List of APIs and SDK capabilities supported

Default values for the supported parameters of these APIs apply to all create and update calls for that API. For example, if a supported parameter is set for `TrainingJob`, then used for all `CreateTrainingJob` and `UpdateTrainingJob` API calls. The paramete

stable

any other API calls unless it is specified for that API as well. However, the default value passed for the `TrainingJob` API is present for any artifacts generated by that API, so any subsequent calls that use these artifacts will also use the value.

The following groups of APIs support setting defaults with a configuration file.

- Feature Group: `CreateFeatureGroup`, `UpdateFeatureGroup`
- Monitoring Schedule: `CreateMonitoringSchedule`, `UpdateMonitoringSchedule`
- Endpoint: `CreateEndpoint`
- Endpoint Config: `CreateEndpointConfig`, `UpdateEndpointConfig`
- Auto ML: `CreateAutoMLJob`, `UpdateAutoMLJob`
- Transform Job: `CreateTransformJob`, `UpdateTransformJob`
- Compilation Job: `CreateCompilationJob`, `UpdateCompilationJob`
- Pipeline: `CreatePipeline`, `UpdatePipeline`
- Model: `CreateModel`, `UpdateModel`
- Model Package: `CreateModelPackage`, `UpdateModelPackage`
- Processing Jobs: `CreateProcessingJob`, `UpdateProcessingJob`
- Training Job: `CreateTrainingJob`, `UpdateTrainingJob`
- Edge Packaging Job: `CreateEdgePackagingJob`

Hyperparameter Tuning Job: Supported indirectly via `TrainingJob` API. While this API is not directly supported, it includes the training job definition as a parameter. If you provide defaults for this parameter as part of the `TrainingJob` API, these defaults are also used for Hyperparameter Tuning Job.

The following groups of SDK capabilities support defaults with a configuration file.

- Remote Function `@remote decorator`, `RemoteExecutor`

Configuration file resolution

To create a consistent experience when using defaults with multiple configuration files, the SageMaker Python SDK merges all of the configuration files into a single configuration dictionary that defines all of the default values set in the environment. The configuration files for defaults are loaded and merged during the initialization of the `Session` object. To access the configuration files, the user must have read access to any local paths set and read access to any S3 URIs that are set. These permissions can be set using the IAM role or other AWS credentials for the user.

If a configuration dictionary is not specified during `Session` initialization, the `Session` automatically calls `load_sagemaker_config()` to load, merge, and validate configuration files from the default locations.

If a configuration dictionary is specified, the `Session` uses the supplied dictionary.

The following sections gives information about how the merging of configuration files happens.

Default configuration files

The `load_sagemaker_config()` method first checks the default location of the admin config file. If one is found, it serves as the basis for further merging. If a config file is not found, then the merged config dictionary is empty. The `load_sagemaker_config()` method then checks the default location for the user config file. If a config file is found, then the values are merged on top of the existing configuration dictionary. This means that the values specified in the user config override the corresponding values specified in the admin config file. If there is not an existing entry for a user config value in the existing configuration dictionary, then a new entry is added.

Specify additional configuration files

In addition to the default locations for your admin and user config files, you can also specify other locations for configuration files. To specify these additional config locations, pass a list of these additional locations as part of the `load_sagemaker_config()` call and pass the resulting dictionary to any `Session` objects you create as shown in the following code sample. These additional configuration file locations are checked in the order specified in the `load_sagemaker_config()` call. When a configuration file is found, it is merged on top of the existing configuration dictionary. All of the values specified in the first additional config override the corresponding values in the default configs. Subsequent additional configuration files are merged on top of the existing configuration dictionary using the same method.

If you are building a dictionary with custom configuration file locations, we recommend that you use `load_sagemaker_config()` and `validate_sagemaker_config()` iteratively to verify the construction of your dictionary before you pass it to a `Session` object.

```
from sagemaker.session import Session
from sagemaker.config import load_sagemaker_config, validate_sagemaker_config

# Create a configuration dictionary manually
custom_sagemaker_config = load_sagemaker_config(
     additional_config_paths=[
        'path1',
        'path2',
        'path3'
    ]
)

# Then validate that the dictionary adheres to the configuration schema
validate_sagemaker_config(custom_sagemaker_config)

# Then initialize the Session object with the configuration dictionary
sm_session = Session(
    sagemaker_config = custom_sagemaker_config
)
```

Tags

Any tags specified in the configuration dictionary are appended to the set of tags set by the SageMaker Python SDK and specified by the user. Each of the tags in the combined list must have a unique key or the API call fails. If a user provides a tag with the same name as a tag in the configuration dictionary, the user tag is used and the config tag is skipped. This behavior applies to all config keys that follow the `SageMaker.*.Tags` pattern.

DebuggerHookConfig

The SageMaker Python SDK only supports setting a boolean value as the default for the `debugger_hook_config` parameter in the configuration dictionary and does not support setting a dictionary as the default value for this parameter.

If the user doesn't manually specify a value for DebuggerHookConfig, the default value specified in the configuration dictionary is used.

If the user manually specifies one of the following values for `DebuggerHookConfig` without passing a value for the `s3_output_path` parameter, the SageMaker Python SDK sets the value of `s3_output_path` to the value specified as part of `SageMaker.Modules.Session.DefaultS3Bucket` in the configuration dictionary.

- dictionary
- True

stable

If the user doesn't provide any value for `DebuggerHookConfig` from function input or in the configuration dictionary, then the SageMaker Python SDK also sets the value of `s3_output_path` to the value specified as part of `SageMaker.Modules.Session.DefaultS3Bucket` in the configuration dictionary.

Users can change the default `s3_output_path` by specifying a value for that parameter in the input dictionary for `DebuggerHookConfig`.

Object Arrays

For the following keys, the configuration file may contain an array of elements, where each element contains one or more values. When a configuration file is merged with an existing configuration dictionary and both contain a value for these keys, the elements in the array defined in the existing configuration dictionary are overridden in order. If there are more elements in the array being merged than in the existing configuration dictionary, then the size of the array is increased.

- `SageMaker.EndpointConfig.ProductionVariants`
- `SageMaker.Model.Containers`
- `SageMaker.ModelPackage.InferenceSpecification.Containers`
- `SageMaker.ModelPackage.ValidationSpecification.ValidationProfiles`
- `SageMaker.ProcessingJob.ProcessingInputs`

When a user passes values for these keys, the behavior depends on the size of the array. If values are not explicitly defined inside the user input array but are defined inside the config array, then those values from the config array are added to the user array. If the user input array contains more elements than the config array, the extra elements of the user input array are not substituted with values from the config. Alternatively, if the config array contains more elements than the user input array, the extra elements of the config array are not used.

View the merged configuration dict

When the SageMaker Python SDK creates your `Session`, it merges together the config files found at the default locations and the additional locations specified in the `load_sagemaker_config()` call. In this process, a new config dictionary is created that aggregates the defaults in all of the config files. To see the full merged config, inspect the config of the session object as follows.

```
session=Session()
session.sagemaker_config
```

stable

Inherit default values from the configuration file

After the `Session` is created, if a value for a supported parameter is present in the merged configuration dictionary and the user does not pass that parameter as part of a SageMaker Python SDK method, the SageMaker Python SDK automatically passes the corresponding value from the configuration dictionary as part of the API call. If a user explicitly passes a value for a parameter that supports default values, the SageMaker Python SDK overrides the value present in the merged configuration dictionary and uses the value passed by the user instead.

Reference values from config

You can manually reference values from the merged configuration dictionary using the corresponding key. This makes it possible to pass these defaults values to an AWS SDK for Python (Boto3) request using the SageMaker Python SDK. To reference a value from the configuration dictionary, pass the corresponding key as follows.

```
from sagemaker.session import Session
from sagemaker.utils import get_sagemaker_config_value
session=Session()

# Option 1
get_sagemaker_config_value(session, "key1.key2")
# Option 2
session.sagemaker_config["key1"]["key2"]
```

You can also specify custom parameters as part of the `CustomParameters` section in a configuration file by setting key and value pairs as shown in the Configuration file structure. Values set in `CustomParameters` are not automatically used. You can only use these values by manually referencing them with the corresponding key.

For example, the following code block references the `VPCConfig` parameter specified as part of the `Model` API in the configuration file and sets a variable with that value. It also references the `JobName` value specified as part of `CustomParameters`.

```
from sagemaker.session import Session
from sagemaker.utils import get_sagemaker_config_value
session = Session(<session values>)

vpc_config_option_1 = get_sagemaker_config_value(session, "SageMaker.Model.VpcConfig")
vpc_config_option_2 = session.sagemaker_config["SageMaker"]["Model"]["VpcConfig"]

custom_param_option_1 = get_sagemaker_config_value(session, "CustomParamet
custom_param_option_2 = session.sagemaker_config["CustomParameters"]["JobName"]
```

Debug default values

The SageMaker Python SDK logs all events related to the defaults configuration. This logging happens with the `sagemaker.config` logger. If you don't configure logging using the Python logging library for the `sagemaker` logger or the `sagemaker.config` logger, the SDK also adds a logging handler that prints INFO level logs to stdout. If you're running the SDK from a SageMaker notebook, these INFO level logs are printed to notebook cell output. The SDK emits INFO logs when config files are found and processed, and when config values are used. By default, the SDK does not log resource ARNs.

If you want to get more information or debug default value injection, enable DEBUG level logs for the `sagemaker.config` logger with the following commands. With DEBUG level logs, the SageMaker Python SDK prints out information about the default value, the configuration dictionary that it came from, the keys that are being looked at, and whether they are used or skipped.

```
import logging
sagemaker_config_logger = logging.getLogger("sagemaker.config")
sagemaker_config_logger.setLevel(logging.DEBUG)
```

Skip default values

If the admin configuration specifies default values that causes issues for your calls, you can temporarily or permanently override those default values.

To temporarily test a configuration change or ignore a default value, modify the config dictionary from `load_sagemaker_config()` before passing it to your `Session` as follows.

1. Create a configuration dictionary manually.

```
from sagemaker.session import Session
from sagemaker.config import load_sagemaker_config, validate_sagemaker_config

custom_sagemaker_config = load_sagemaker_config()
```

2. Modify the configuration dictionary. For example, delete the `RoleArn` value passed as part of `TrainingJob` calls.

stable

```
del custom_sagemaker_config["SageMaker"]["TrainingJob"]["RoleArn"]
```

3. Validate that the dictionary adheres to the configuration schema.

```
validate_sagemaker_config(custom_sagemaker_config)
```

4. Initialize the Session object with the customized configuration dictionary.

```
sm_session = Session(
    sagemaker_config = custom_sagemaker_config
)
```

To permanently override a default value from the admin configuration, create a user configuration with a non-None value for the default that you want to override.

Run Machine Learning code on SageMaker using remote function

You can integrate your local machine language (ML) code to run in a Amazon SageMaker Training job by wrapping your code inside a @remote decorator as shown in the following code example.

```
from sagemaker.remote_function import remote
import numpy as np

@remote(instance_type="ml.m5.large")
def matrix_multiply(a, b):
    return np.matmul(a, b)

a = np.array([[1, 0],
             [0, 1]])
b = np.array([1, 2])

assert (matrix_multiply(a, b) == np.array([1,2])).all()
```

The SageMaker Python SDK will automatically translate your existing workspace environment and any associated data processing code and datasets into a SageMaker Training job that runs on the SageMaker Training platform. You can also activate a persistent cache feature, which will further reduce job start up latency by caching previously downloaded depende

stable

This reduction in job latency is greater than the reduction in latency from using SageMaker managed warm pools alone. The following sections show you how to wrap your local ML code and tailor your experience for your use case including customizing your environment and integrating with SageMaker Experiments.

See the Run your local code as a SageMaker Training job for detailed developer guide.

Follow is the API specification for methods and classes related to remote function feature.

- Remote function classes and methods specification

FAQ

I want to train a SageMaker Estimator with local data, how do I do this?

Upload the data to S3 before training. You can use the AWS Command Line Tool (the aws cli) to achieve this.

If you don’t have the aws cli, you can install it using pip:

```
pip install awscli --upgrade --user
```

If you don’t have pip or want to learn more about installing the aws cli, see the official Amazon aws cli installation guide.

After you install the AWS cli, you can upload a directory of files to S3 with the following command:

```
aws s3 cp /tmp/foo/ s3://bucket/path
```

For more information about using the aws cli for manipulating S3 resources, see AWS cli command reference.

How do I make predictions against an existing endpoint?

Create a `Predictor` object and provide it with your endpoint name, then call its `predict()` method with your input.

stable

You can use either the generic `Predictor` class, which by default does not perform any serialization/deserialization transformations on your input, but can be configured to do so through constructor arguments: http://sagemaker.readthedocs.io/en/stable/predictors.html

Or you can use the TensorFlow / MXNet specific predictor classes, which have default serialization/deserialization logic:
http://sagemaker.readthedocs.io/en/stable/sagemaker.tensorflow.html#tensorflow-predictor
http://sagemaker.readthedocs.io/en/stable/sagemaker.mxnet.html#mxnet-predictor

Example code using the TensorFlow predictor:

```
from sagemaker.tensorflow import TensorFlowPredictor

predictor = TensorFlowPredictor('myexistingendpoint')
result = predictor.predict(['my request body'])
```