Content preferences

Language

English

Lab3c. Bring your own Container

Prerequisites

The objective of this lab is to give you step by step instructions on how to bring your own custom containers to Amazon Sagemaker in your AWS account

- The following steps are an explanation on the cells you will be executing by pressing **Shift+Enter** in an Amazon SageMaker Notebook instance.
- Follow the instructions to launch Amazon SageMaker Studio
- Please ensure that you have git cloned the repository in your SageMaker Studio.
- **Attach**
    - a trust policy to the SageMaker execution IAM role for AWS CodeBuild. This ensures that in addition to Amazon SageMaker, AWS CodeBuild can also assume this role so that it could automate building docker images
        - go to SageMaker Studio, then click on the SageMaker Studio "User name"
        - copy the name of the SageMaker Studio execution role (not the full ARN).
        - go to IAM, then click on "Roles", then search roles by the name of the execution role you copied above (the format should be similar to this `AmazonSageMaker-ExecutionRole-20200718T003573`) then click on the only item that appears under "Role name"
        - click on the "Trust relationships" tab, then click on "Edit trust relationship"

- in the "Edit trust relationship" text area, paste the json iam policy from the codeblock below or this repository, then click on "Update Trust Policy"

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "sagemaker.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    },
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "codebuild.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

- an additional IAM policy the SageMaker execution IAM role to allow it to build the docker image using AWS CodeBuild
  - click on the "Permissions" tab in IAM roles, then click on "Add inline policy"
  - in the "Create policy" page click on the "JSON" tab, then copy the contents of the file from this repository and paste it in the text area replacing the default text, then click on "Review policy". In Name*, enter `SageMakerCodeBuildPolicy`, select "Create policy".

Overview

This lab helps you build your own custom container image and then run it on Amazon SageMaker.

Docker is a program that performs operating-system-level virtualization for installing, distributing, and managing software. It packages applications and their dependencies into virtual containers that provide isolation, portability, and security. You can put scripts, algorithms, and inference code for your machine learning models into containers. The container includes the runtime, system tools, system libraries, and other code required to train your algorithms or deploy your models. By using containers, you can train machine learning algorithms and deploy models quickly and reliably at any scale.

Here, we'll show how to package a simple Python example which showcases the decision tree algorithm from the widely used scikit-learn machine learning package. The example is purposefully fairly trivial since the point is to show the surrounding structure that you'll want to add to your own code so you can train and host it in Amazon SageMaker.

Content preferences

Language

The ideas shown here will work in any language or environment. You'll need to choose the right tools for your environment to serve HTTP requests for inference, but good HTTP environments are available in every language these days.

In this example, we use a single image to support training and hosting. This is easy because it means that we only need to manage one image and we can set it up to do everything. Sometimes you'll want separate images for training and hosting because they have different requirements. Just separate the parts discussed below into separate Dockerfiles and build two images. Choosing whether to have a single image or two images is really a matter of which is more convenient for you to develop and manage.

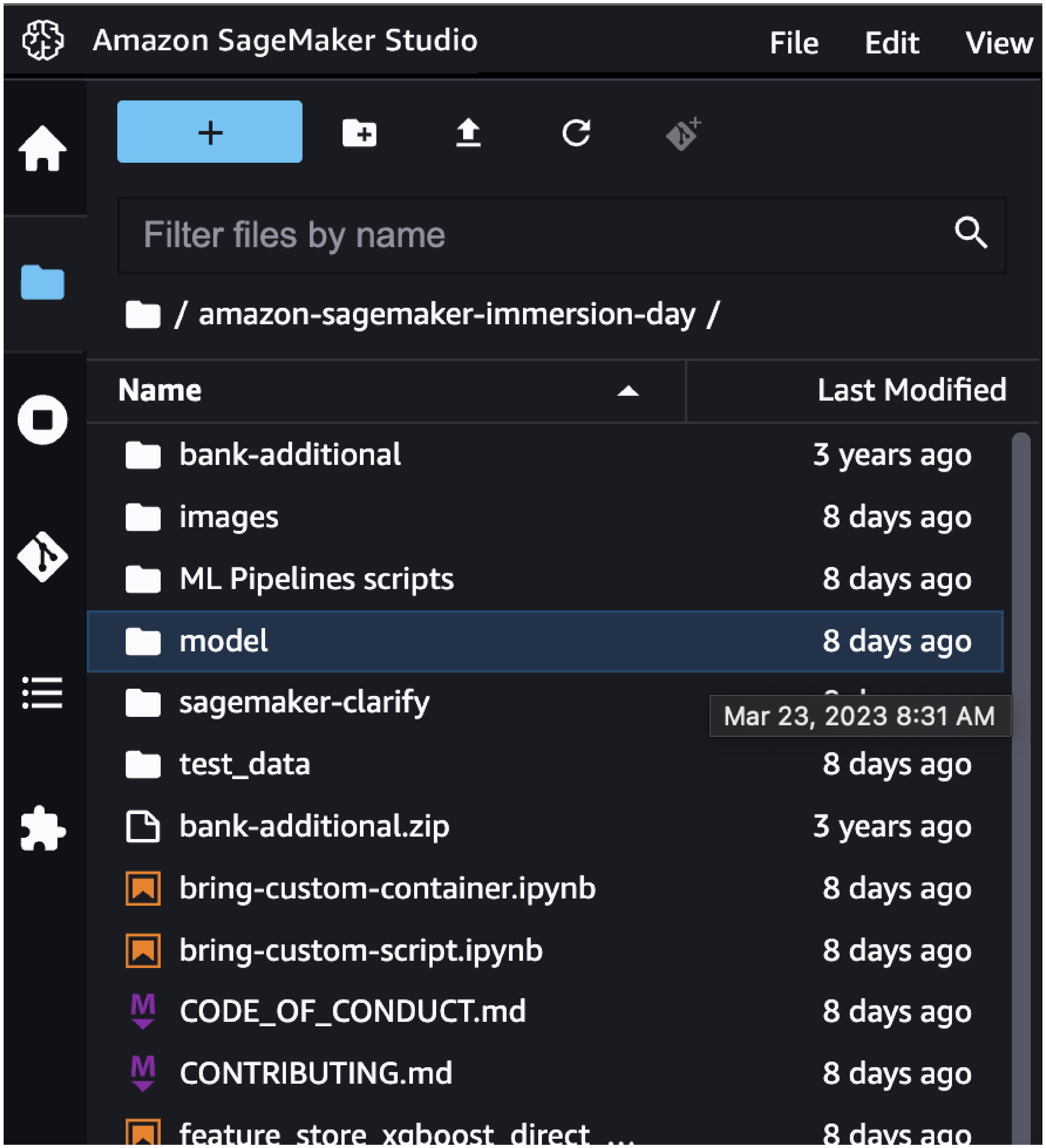
The notebook contains detailed explanation of each step. This guide helps by providing steps to execute and expected outcomes of each step.

Launch the notebook instance

In your SageMaker Studio, in the "File Browser" pane on the left hand side, click on the file "amazon-sagemaker-immersion-day/bring-custom-container.ipynb"

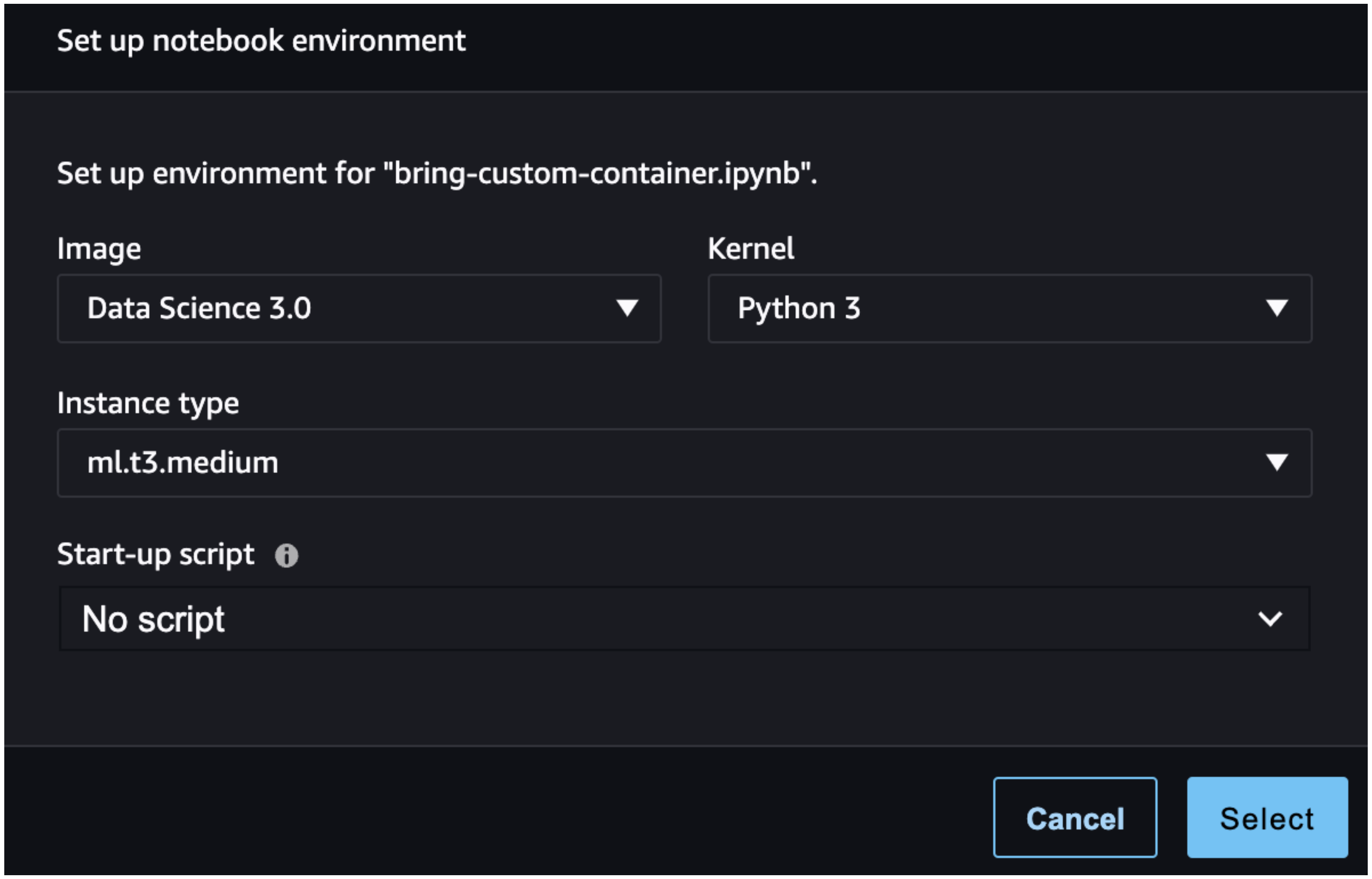
Content preferences

Language



Name	Last Modified
bank-additional	3 years ago
images	8 days ago
ML Pipelines scripts	8 days ago
model	8 days ago
sagemaker-clarify	Mar 23, 2023 8:31 AM
test_data	8 days ago
bank-additional.zip	3 years ago
bring-custom-container.ipynb	8 days ago
bring-custom-script.ipynb	8 days ago
CODE_OF_CONDUCT.md	8 days ago
CONTRIBUTING.md	8 days ago
feature_store_xgboost_direct ...	8 days ago

You will be prompted to choose a kernel. Choose **Python 3** Kernel **Data Science 3.0** Image.



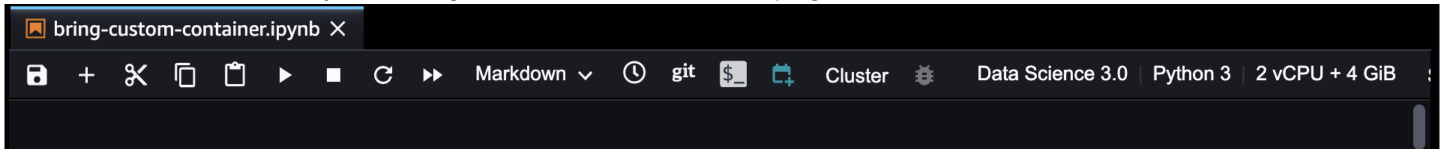
Set up notebook environment

Set up environment for "bring-custom-container.ipynb".

Image: Data Science 3.0
Kernel: Python 3
Instance type: ml.t3.medium
Start-up script: No script

The notebook kernel will take a while to start:

After that, will be able to verify that the right kernel is selected on the top right of the screen:



bring-custom-container.ipynb

Data Science 3.0 | Python 3 | 2 vCPU + 4 GiB

Building the container

Docker provides a simple way to package arbitrary code into an image that is totally self-contained. Once you have an image, you can use Docker to run a container based on that image. Running a container is just like running a program on the machine except that the container creates a fully self-contained environment for the program to run. Containers are isolated from each other and from the host environment, so the way you set up your program is the way it runs, no matter where you run it.

Amazon SageMaker uses Docker to allow users to train and deploy arbitrary algorithms.

We will first install the prerequisite packages:

- `aiobotocore`: adds async support for AWS services with botocore.
- `sagemaker-studio-image-build`: CLI for building Docker images in SageMaker Studio using AWS CodeBuild

```
# cell 00
!/opt/conda/bin/python -m pip install --upgrade pip
!pip install -q s3fs==2022.5.0
!pip install -q boto3==1.21.21
!pip install -q botocore==1.29.91
!pip install -q awscli==1.27.91
!pip install -Uq aiobotocore==2.3.0
!pip install -q pandas==1.3.5
!pip install -q sagemaker-studio-image-build
```

In cell 01, we'll unzip and copy the `data/` and `container/` folders from `scikit-bring-your-own` folder. Then we'll print the contents of the `lab03_container/Dockerfile` used to build the docker image

```
# cell 01
!unzip -q scikit_bring_your_own.zip
!mv scikit_bring_your_own/data/ ./lab03_data/
!mv scikit_bring_your_own/container/ ./lab03_container/
!rm -rf scikit_bring_your_own
```

Building and registering the container

Build the docker image using `sm-docker` that uses AWS CodeBuild. This step takes a few minutes to complete

```
%%sh
# cell 02

cd lab03_container

chmod +x decision_trees/train
chmod +x decision_trees/serve

sm-docker build .  --repository sagemaker-decision-trees:latest
```

```
....[Container] 2023/03/23 16:50:59 Waiting for agent ping

[Container] 2023/03/23 16:51:00 Waiting for DOWNLOAD_SOURCE
[Container] 2023/03/23 16:51:04 Phase is DOWNLOAD_SOURCE
[Container] 2023/03/23 16:51:04 CODEBUILD_SRC_DIR=/codebuild/output/src015459051/src
[Container] 2023/03/23 16:51:04 YAML location is /codebuild/output/src015459051/src/buildspec.yml
[Container] 2023/03/23 16:51:04 Setting HTTP client timeout to higher timeout for S3 source
[Container] 2023/03/23 16:51:04 Processing environment variables
[Container] 2023/03/23 16:51:04 No runtime version selected in buildspec.
[Container] 2023/03/23 16:51:04 Moving to directory /codebuild/output/src015459051/src
[Container] 2023/03/23 16:51:04 Configuring ssm agent with target id: codebuild:215e20d0-f349-433e-b477-4b83af5
3610d
[Container] 2023/03/23 16:51:04 Successfully updated ssm agent configuration
[Container] 2023/03/23 16:51:04 Registering with agent
[Container] 2023/03/23 16:51:04 Phases found in YAML: 3
[Container] 2023/03/23 16:51:04  PRE_BUILD: 9 commands
[Container] 2023/03/23 16:51:04  BUILD: 4 commands
[Container] 2023/03/23 16:51:04  POST_BUILD: 3 commands
[Container] 2023/03/23 16:51:04 Phase complete: DOWNLOAD_SOURCE State: SUCCEEDED
```

akadam

Content preferences

Language

```
[Container] 2023/03/23 16:51:04 Phase context status code:  Message:
[Container] 2023/03/23 16:51:04 Entering phase PRE_BUILD
[Container] 2023/03/23 16:51:04 Running command echo Logging in to Amazon ECR...
Logging in to Amazon ECR...
```

The last few lines of the output will look like the following:

```
[Container] 2023/03/31 17:50:44 Phase complete: POST_BUILD State: SUCCEEDED
[Container] 2023/03/31 17:50:44 Phase context status code:  Message:

Image URI: 607789152815.dkr.ecr.us-west-2.amazonaws.com/sagemaker-decision-trees:latest
```

⚠ If you get "Access denied" errors, make sure that you have applied the IAM trust policy and attached the additional IAM policy to the default SageMaker execution role as explained the Prerequisites above

Setup & Upload Data

Content preferences

Language

Here we specify a bucket to use and the role that will be used for working with SageMaker.

```
# cell 03

S3_prefix = 'DEMO-scikit-byo-iris'

# Define IAM role
import boto3
import re

import os
import numpy as np
import pandas as pd
from sagemaker import get_execution_role

role = get_execution_role()
```

The session remembers our connection parameters to SageMaker. We'll use it to perform all of our SageMaker operations.

```
# cell 04

import sagemaker as sage
from time import gmtime, strftime

sess = sage.Session()
```

We can use use the tools provided by the SageMaker Python SDK to upload the data to a default bucket

```
WORK_DIRECTORY = 'lab03_data'

data_location = sess.upload_data(WORK_DIRECTORY,
                                 key_prefix=S3_prefix)
```

Model Training

Then we use `fit()` on the estimator to train against the data that we uploaded above. This step usually takes about a minute to complete.

```
# cell 06

account = sess.boto_session.client('sts').get_caller_identity()['Account']
region = sess.boto_session.region_name
image_uri = '{}.dkr.ecr.{}.amazonaws.com/sagemaker-decision-trees:latest'.format(account, region)

tree = sage.estimator.Estimator(image_uri,
                                role,
                                instance_count=1,
                                instance_type='ml.c4.2xlarge',
                                output_path="s3://{}/output".format(sess.default_bucket()),
                                sagemaker_session=sess)

file_location = data_location + '/iris.csv'
tree.fit(file_location)
```

When the model finish training. The last few lines of the output of cell 07 looks like the following:

```
2023-03-31 17:51:17 Starting - Starting the training job...
2023-03-31 17:51:41 Starting - Preparing the instances for trainingProfilerReport-1680285077: InProgress
......
2023-03-31 17:52:41 Downloading - Downloading input data...
2023-03-31 17:53:12 Training - Training image download completed. Training in progress..Starting the training.
Training complete.

2023-03-31 17:53:42 Uploading - Uploading generated training model
2023-03-31 17:53:42 Completed - Training job completed
Training seconds: 63
Billable seconds: 63
```

Hosting your model

You can use a trained model to get real time predictions using HTTP endpoint. Deploying the model to SageMaker hosting just requires a deploy call on the fitted model. This call takes an instance count, instance type, and optionally serializer and deserializer functions. These are used when the resulting predictor is created on the endpoint. This step takes about a few minutes to complete

```
# cell 07

from sagemaker.serializers import CSVSerializer
predictor = tree.deploy(initial_instance_count=1,
                        instance_type='ml.m4.xlarge',
                        serializer=CSVSerializer())
```

Preparing test data to run inferences

In order to do some predictions, we'll extract some of the data we used for training and do predictions against it. This is, of course, bad statistical practice, but a good way to see how the mechanism works.

```
# cell 08

shape=pd.read_csv(file_location, header=None)
shape.sample(3)
```

```
# cell 09

# drop the label column in the training set
shape.drop(shape.columns[[0]],axis=1,inplace=True)
shape.sample(3)
```

```
# cell 10

import itertools

a = [50*i for i in range(3)]
b = [40+i for i in range(10)]
indices = [i+j for i,j in itertools.product(a,b)]

test_data=shape.iloc[indices[:-1]]
```

Run predictions

Prediction is as easy as calling predict with the predictor we got back from deploy and the data we want to do predictions with. The serializers take care of doing the data conversions for us.

```
# cell 11

print(predictor.predict(test_data.values).decode('utf-8'))
```

```
setosa
setosa
setosa
setosa
setosa
setosa
setosa
setosa
setosa
setosa
versicolor
versicolor
versicolor
versicolor
versicolor
```

Cleanup

After completing the lab, use these steps to delete the endpoint or run the following code

```
# cell 12
sess.delete_endpoint(predictor.endpoint_name)
```

Remove the container artifacts and data we downloaded.

Conclusion

In this lab,

- you used a Dockerfile to create a Docker container having a custom decision tree algorithm from the widely used scikit-learn machine learning package,
- pushed it to Amazon ECR,
- using that container, you trained your model with a sample dataset,
- deployed it to an HTTP endpoint and
- ran inferences against some test data derived from the same dataset