Deploying a Scikit-Learn Model on AWS Using SKLearn Estimators, Local Jupyter Notebooks, and the Terminal

A step-by-step tutorial on how to deploy a scikit-learn model on AWS from a local device.

Quinn Lanners
Feb 25 · 8 min read ★

Logos retrieved from Wikipedia.com (scikit-learn) and aws.amazon.com.

Amazon Web Services (AWS) is currently the most in-demand cloud platform for Data Science and Machine Learning professionals. Given this, the first collection of blog posts in this Machine Learning Deployment series will outline how to deploy various models in a number of different ways on AWS.

- *Intro*

The first AWS deployment we will cover will be the deployment of a simple Scikit-learn model done completely from one's local computer using the AWS pre-build Scikit-learn Estimators.

Before we dig into the steps, it's important to understand the basic concept of how AWS Sagemaker trains and deploys models. If you are unfamiliar with Docker, I recommend reading their overview docs here. Without going into too much detail, AWS essentially uses Docker containers to store a model's configuration, train the model, and eventually deploy the model. By encapsulating ML models within Docker containers, AWS is able to simultaneously provide a set of preconfigured Docker images for common ML frameworks, while also allowing for complete customization.

Luckily for us, in 2018, AWS added Scikit-learn to its list of supported frameworks. Thus, we can quickly deploy our model using a pre-configured Docker container. As you'll see below, we will create a training script and execute it inside the container using the AWS Scikit-learn Estimators. So, without further ado, let's get to it.

> *You can find the code from this post in the repo I started for this blog series here.*

Step 1: Account Setup

Before we begin, we have to set up an AWS account. If you already have an AWS account, then you can skip to the next step. Otherwise, let's get you signed up for AWS and their 175 (and continually growing) services. To start, navigate to this page to create your free AWS account and get 12 months of free tier access (which will be plenty to complete the code in this post).

Step 2: AWS CLI and Pip Packages

Once you have registered for AWS, we also need to download the AWS command-line interface (CLI) so we can work on our AWS account from our local device. AWS does a good job of explaining how to install, configure, and use their CLI.

We will also need to install a couple of python SDKs in order to access AWS in our python scripts. Specifically, we will need the Boto3 to handle data upload and

```
pip install boto3
pip install sagemaker
```

Step 3: Data Set-Up

In order to focus on the deployment of the model and avoid getting caught up in the weeds regarding data cleaning and model tuning, we are going to be training a simple logistic regression model on the Iris dataset. You can download the dataset here and save it to your local device.

We will use the code block below to strip a few samples from the dataset for testing out API calls once we have deployed the model, and save the data into two new files called **train.csv** and **deploy_test.csv**.

```
import pandas as pd
data = pd.read_csv('iris.data',
                   names=['sepal length', 'sepal width',
                          'petal length', 'petal width',
                          'label'])

# Shuffle the data to make deploy_test samples random
data = data.sample(frac=1).reset_index(drop=True)
train = data[:-3]
deploy_test = data[-3:]
train.to_csv('train.csv')
deploy_test.to_csv('deploy_test.csv')
```

aws_sklearn_split_data hosted with ♡ by GitHub view raw

Note: We are disregarding ML train/test set guidelines. In practice, you should always be splitting your data into a train/test set. However, for this tutorial, we are just focusing on deploying the model and will check how the model predicts the few points we saved in ***deploy_test.csv*** *by making an API call at the end.*

In order for AWS Sagemaker to access this data when it comes time to train the model, we must upload the data to our AWS account by creating a new Simple Storage Service (S3) bucket and adding our training data to it. To do this, we will be using the boto3

Note: In order for Sagemaker to access the data, the S3 bucket and Sagemaker session need to be in the same region. And given that Sagemaker is only available in certain regions, be sure to select a region for your S3 bucket from the regions listed here.

```
import boto3
bucket = 'sagemaker-learning-to-deploy-scikitlearn'
region = 'us-east-2'
s3_session = boto3.Session().resource('s3')
s3_session.create_bucket(Bucket=bucket,
                         CreateBucketConfiguration=
                         {'LocationConstraint': region})
s3_session.Bucket(bucket).Object('train/train.csv').upload_file('train.csv')
```

aws_sklearn_upload_train_data hosted with ♡ by GitHub view raw

To double-check that the above scripts worked correctly, you can navigate to the AWS S3 Console, login, and see if the bucket we created above is there. If it's there, we are ready to move on to the meat of the work and create our model!

Step 4: The Model Script

In order to deploy a model to AWS using the Scikit-learn Sagemaker SDK, we first have to create a script that tells Sagemaker how to train and deploy our model. While much less work than creating our own Docker container to deploy our model, the SDK does require us to stick to rather strict guidelines.

First off, we will import all the packages we will need and create a couple of dictionaries to convert the target labels from text to numbers and vice versa.

```
import argparse
import numpy as np
import os
import pandas as pd
from sklearn.externals import joblib
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler


# Dictionary to convert labels to indices
```

```
    'Iris-versicolor': 1,
    'Iris-setosa': 2
}

# Dictionary to convert indices to labels
INDEX_TO_LABEL = {
    0: 'Iris-virginica',
    1: 'Iris-versicolor',
    2: 'Iris-setosa'
}
```

aws_sklearn_imports_and_constants hosted with ♡ by GitHub view raw

After these imports, we need to wrap the training steps within a __main__ function. We will also include a line to save the model at the end. The SKLearn Sagemaker SDK will run this __main__ function to train a model on the data we pushed to the S3 bucket above and then save the model so that it can be used later when we want to deploy it.

```
if __name__ =='__main__':
    # Create a parser object to collect the environment variables that are in the
    # default AWS Scikit-learn Docker container.
    parser = argparse.ArgumentParser()

    parser.add_argument('--output-data-dir', type=str, default=os.environ.get('SM_OUTPUT_DATA_D
    parser.add_argument('--model-dir', type=str, default=os.environ.get('SM_MODEL_DIR'))
    parser.add_argument('--train', type=str, default=os.environ.get('SM_CHANNEL_TRAIN'))
    parser.add_argument('--test', type=str, default=os.environ.get('SM_CHANNEL_TEST'))

    args = parser.parse_args()

    # Load data from the location specified by args.train (In this case, an S3 bucket).
    data = pd.read_csv(os.path.join(args.train,'train.csv'), index_col=0, engine="python")

    # Seperate input variables and labels.
    train_X = data[[c for c in data.columns if c != 'label']]
    train_Y = data[['label']]

    # Convert labels from text to indices
    train_Y_enc = train_Y['label'].map(LABEL_TO_INDEX)

    #Train the logistic regression model using the fit method
    model = LogisticRegression().fit(train_X, train_Y_enc)

```

The __main__ function is the only required function. However, we have the ability to modify a number of additional functions that determine how the model will handle API calls once it is deployed. These optional functions are:

> **model_fn:** specifies from where and how to load the model that is being deployed.
>
> **input_fn:** formats the request body sent to the deployed model into a format that can be fed into the model.
>
> **predict_fn:** uses the deployed model loaded by model_fn and the data formatted by input_fn to make predictions.
>
> **output_fn:** reformats the predictions made by predict_fn into the final format that is returned as the response to the API call.

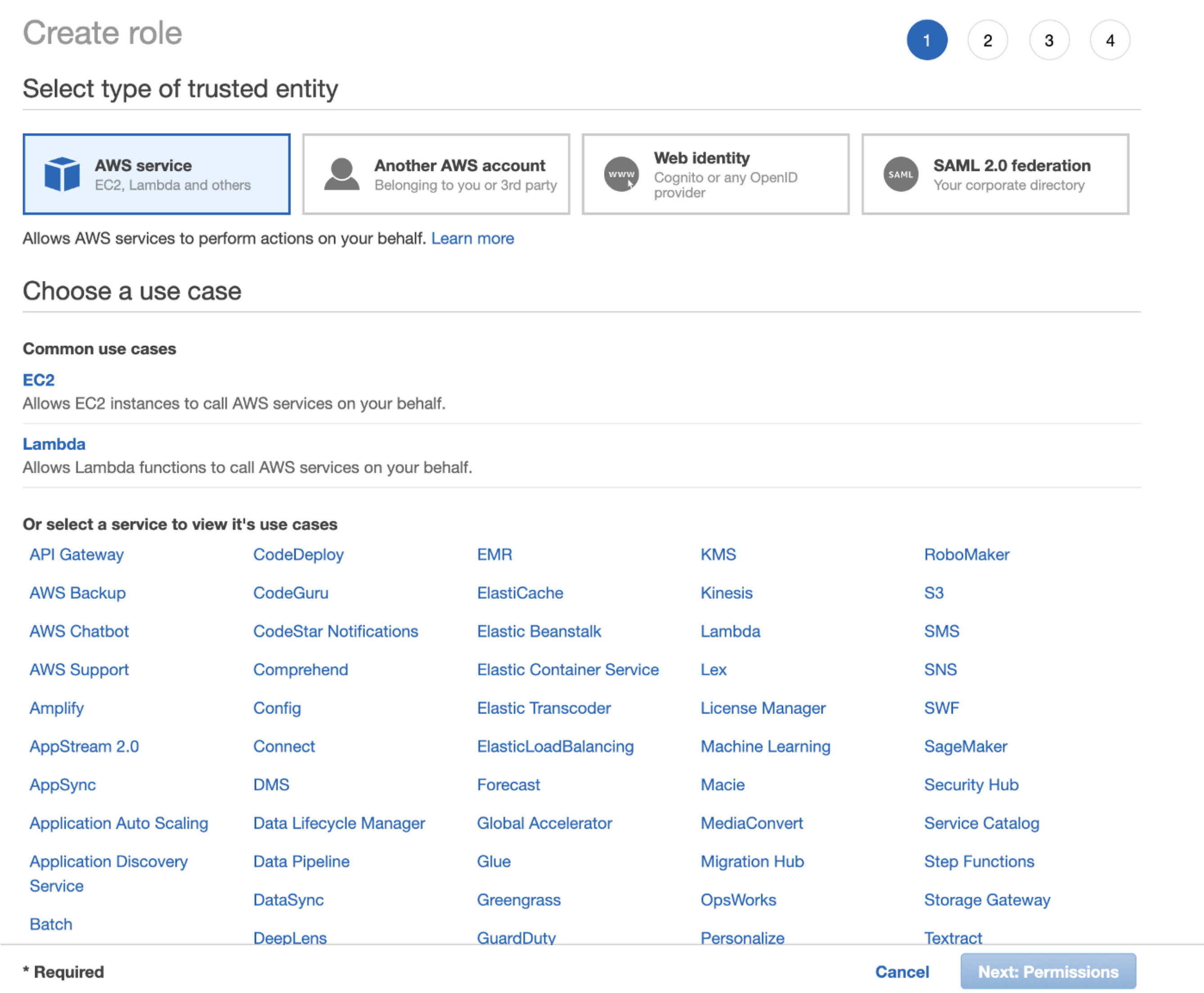
The Gist below shows these optional functions for our model. The comments above each function provide a bit more information on each. For even more information on these functions and their default behavior see here.

```
"""
model_fn
    model_dir: (sting) specifies location of saved model

This function is used by AWS Sagemaker to load the model for deployment.
It does this by simply loading the model that was saved at the end of the
__main__ training block above and returning it to be used by the predict_fn
function below.
"""
def model_fn(model_dir):
    model = joblib.load(os.path.join(model_dir, "model.joblib"))
    return model

"""
input_fn
    request_body: the body of the request sent to the model. The type can vary.
    request_content_type: (string) specifies the format/variable type of the request

This function is used by AWS Sagemaker to format a request body that is sent to
```

```
return that array to be used by the predict_fn function below.

Note: Oftentimes, you will have multiple cases in order to
handle various request_content_types. Howver, in this simple case, we are
only going to accept text/csv and raise an error for all other formats.
"""
def input_fn(request_body, request_content_type):
    if content_type == 'text/csv':
        samples = []
        for r in request_body.split('|'):
            samples.append(list(map(float,r.split(','))))
        return np.array(samples)
    else:
        raise ValueError("Thie model only supports text/csv input")

"""
predict_fn
    input_data: (numpy array) returned array from input_fn above
    model (sklearn model) returned model loaded from model_fn above

This function is used by AWS Sagemaker to make the prediction on the data
formatted by the input_fn above using the trained model.
"""
def predict_fn(input_data, model):
    return model.predict(input_data)

"""
output_fn
    prediction: the returned value from predict_fn above
    content_type: (string) the content type the endpoint expects to be returned

This function reformats the predictions returned from predict_fn to the final
format that will be returned as the API call response.

Note: While we don't use content_type in this example, oftentimes you will use
that argument to handle different expected return types.
"""
def output_fn(prediction, content_type):
    return '|'.join([INDEX_TO_LABEL[t] for t in prediction])
```

Step 5: Create IAM Sagemaker Role

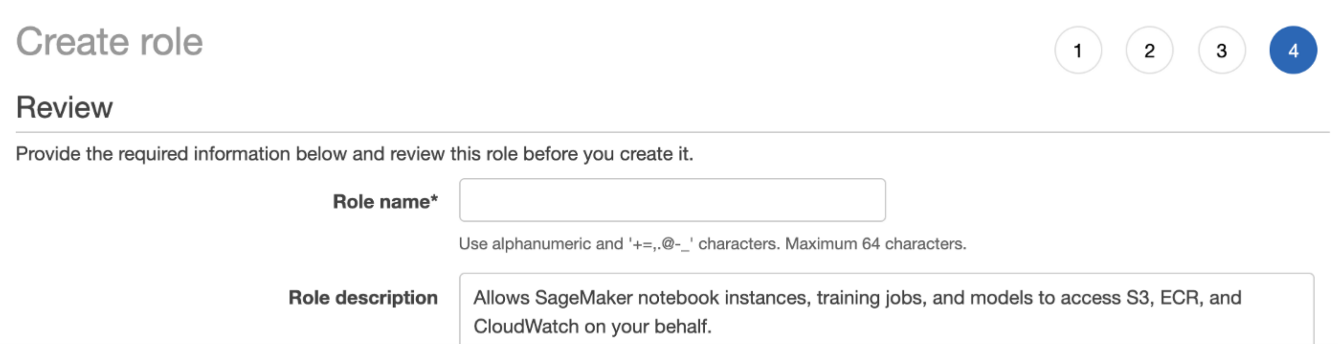
to navigate to the AWS IAM Console and create a new role. Start by clicking here and logging in. You should see a screen like the one below:

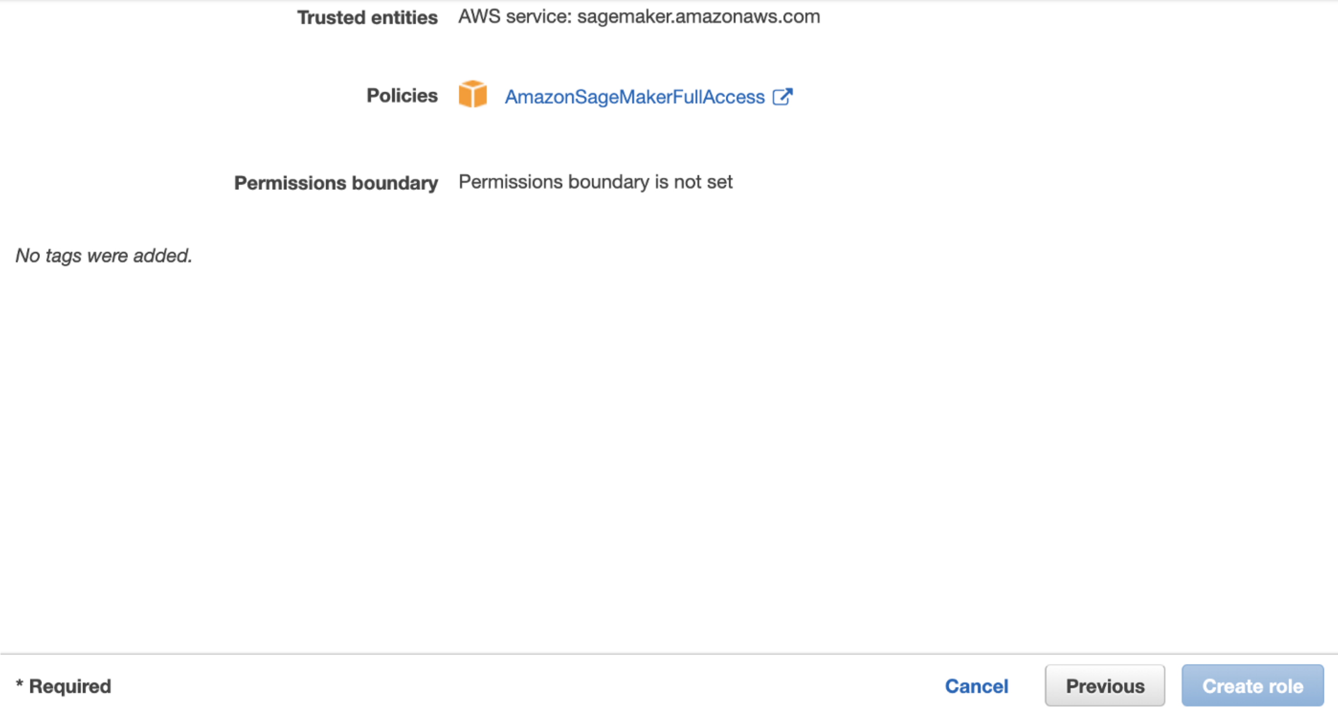


AWS Select IAM Role Page

From here, select **Sagemaker** as the service and push **Next**.

Skip through the next two pages by selecting **Next**, until you see the following screen:





AWS Create IAM Role Name Page

On this page, just create your own **Role name*** and (optional) **Role description** before pressing **Create Role**. Just be sure to remember what you named your role, as we will be needing it in the next step!

Step 6: Deploy the Model

With the hard work out of the way, let's deploy your model. Luckily, with AWS SKLearn class, this only takes a few lines of code. Just make sure that the ***entry_point*** path points to the script we saved in step 4 and the ***role*** variable is the **Role name*** you created in step 5. In this snippet, we also specify instance_types for the model and the deployed endpoint. The instance_type specifies how much computing power we want AWS to allocate for our services. Obviously, the more power the more cost, so for this example we use small instances. Check here for a list of all the Sagemaker instance types available.

```
from sagemaker.sklearn.estimator import SKLearn

role = 'role_created_in_step_5'

# Create the SKLearn Object by directing it to the aws_sklearn_main.py script
aws_sklearn = SKLearn(entry_point='aws_sklearn_main.py',
                      train_instance_type='ml.m4.xlarge',
                      role=role)
```

```
aws_sklearn.fit({'train': 's3://sagemaker-learning-to-deploy-scikitlearn/train'})

# Deploy model
aws_sklearn_predictor = aws_sklearn.deploy(instance_type='ml.m4.xlarge',
                                           initial_instance_count=1)

# Print the endpoint to test in next step
print(aws_sklearn_predictor.endpoint)

# Uncomment and run to terminate the endpoint after you are finished
#predictor.delete_endpoint()
```

aws_sklearn_deploy hosted with ♡ by GitHub view raw

Note: the SKLearn() constructor has a number of optional arguments you can add to configure the Scikit-learn framework_version, hyperparameters, etc. Click here for more info.

Step 7: Test the Endpoint

To test the endpoint, we will feed send a request with the data samples we saved to **deploy_test.csv** way back in step 3. In order to send a request to the deployed endpoint, we first need to get our test samples into a format that the model can parse (i.e. a format that can be interpreted by the **input_fn** function we defined in **aws_sklearn_main.py**). Since we configured our model to understand requests with multiple samples were each sample's features are separated by a "," and each individual sample is separated by a "|" we format our request_body into a format like below:

```
'147,6.5,3.0,5.2,2.0|148,6.2,3.4,5.4,2.3|149,5.9,3.0,5.1,1.8'
```

From here, we use boto3 to create a Sagemaker session which will allow us to interact with our deployed model. In order to call our Sagemaker endpoint, we use the **invoke_endpoint** function. For this function, we must specify an endpoint, a conent_type, and a body. There are also a number of optional arguments you can read more about here. In our case, we will pass the endpoint that we printed out in the previous step, the content_type 'text/csv', and a string formatted to look like above.

```
import boto3
import pandas as pd

# Load in the deploy_test data
deploy_test = pd.read_csv("deploy_test.csv").values.tolist()

# Format the deploy_test data features
request_body = ""
for sample in deploy_test:
    request_body += ",".join([str(n) for n in sample[1:-1]]) + "|"
request_body = request_body[:-1]

# create sagemaker client using boto3
client = boto3.client('sagemaker-runtime')

# Specify endpoint and content_type
endpoint_name = "endpoint_from_deployed_model_in_step_6"
content_type = "text/csv"

# Make call to endpoint
response = client.invoke_endpoint(
    EndpointName=endpoint_name,
    ContentType=content_type,
    Body=request_body
    )

# Print out expected and returned labels
print(f"Expected {', '.join([n[-1] for n in deploy_test])}")
print("Returned:")
print(response['Body'].read())
```

aws_sklearn_test_endpoint hosted with ♡ by GitHub view raw

If the expected response matches what was actually returned by the model, you have successfully trained and deployed a working Scikit-learn model on AWS!

Step 8: Clean Up Resources

In order to avoid any AWS charges, be sure to clean up your resources. You can quickly terminate your model's endpoint by uncommenting the final line in Step 6 and running it.

While this will delete the endpoint, to totally purge your AWS account of all of the resources you used in this tutorial complete the following.

1. Open the Sagemaker Console
2. In the side-bar menu under **Inferences,** delete the resources created on the *Models*, *Endpoint*, and *Endpoint configurations* tabs.
3. Open the S3 Console and delete the bucket we created.
4. Open the IAM Console and delete the role we created.
5. Open the Cloudwatch Console and delete all of the /aws/sagemaker logs.

Review

Overall, while the learning curve is a bit high, once you are able to navigate your way around the AWS services and understand the Sagemaker principals, AWS SKLearn Estimators become an incredible tool to quickly deploy Scikit-learn models.

Benefits:

- Very little configuration needed, but lots available
- Plethora of documentation
- Can deploy model with just a few lines of code

Drawbacks

- The large initial learning curve
- Several AWS services to understand and work with
- Documentation can be difficult to find

Other Blog Posts in this Series:

- *Intro*