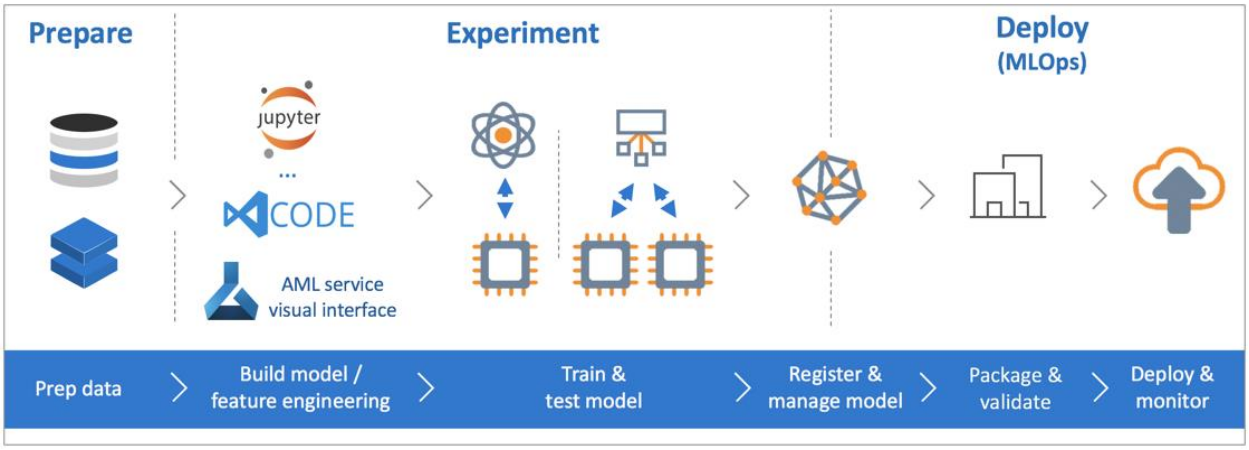
TRAIN AND DEPLOY MACHINE LEARNING MODELS USING THE AZURE ML SERVICE

POSTED BY JASON CANTRELL

Microsoft's cloud-based, scalable Azure Machine Learning (ML) service speeds development and deployment of data science projects. In this demo, we'll use the Azure Machine Learning SDK for Python to train a simple machine learning model and deploy it to the cloud service to then get predictions from our model.



Before beginning, you'll need an Azure subscription (create one free here). Then you'll need to create an Azure Machine Learning service workspace resource in your tenant. Once you spin up the workspace, you will also need to create a Notebook VM, then open a Jupyter notebook to code in. Each of the code segments below are blocks of code inside of the notebook I created for this demo.

We will cover three major sections in our walkthrough: 1) setting up the development environment, 2) training the predictive model, and 3) deploying the model as an Azure ML service – each step has several sub-steps.

Set Up the Development Environment

We'll begin by setting up our environment and getting it ready to train an experiment. There are five essential steps to getting our environment ready for an experiment:

1. Initialize the Workspace
2. Initialize an Experiment
3. Create a Datastore
4. Create or Attach a Compute Resource
5. Environment Set Up

Initialize the Workspace

This step imports the base Azure ML packages into our Python code and connects us to the workspace we created in the Azure portal. The print lines are, of course, just a check to make sure that we've imported and are executing the code properly. The important lines of code in this section are the importing of the azureml.core package, which contains all of our SDK functions, and the initializing of the Workspace object.

```
# base packages to work with AMLS
import azureml.core
from azureml.core import Workspace

# check core SDK version number
print("Azure ML SDK Version: ", azureml.core.VERSION)

# load workspace configuration from the config.json file in the current folder.
ws = Workspace.from_config()
print(ws.name, ws.location, ws.resource_group, ws.location, sep='\t')
```

view rawInitializeWorkspace.py hosted with ♡ by GitHub

Initialize an Experiment

Next, we'll initialize our Experiment. Let's give it the name that we'll save it as. Then we'll create an Experiment object that links the workspace to the experiment for us to use later.

```
# Name your experiment here.
experiment_name = 'framework'

from azureml.core import Experiment
exp = Experiment(workspace=ws, name=experiment_name)
```

view rawInitializeExperiment.py hosted with ♡ by GitHub

Create a Datastore

In this step, we need to create a folder in our workspace in which to write out our code in some later steps. Any custom Python scripts used in Training or Scoring the Model should be added to this folder after it is created. This is also the folder where we'll want to write out any file outputs (Training, Scoring, YAML, etc.)

```
# Directory to write training script.
# Code Directory
import os
script_folder = os.path.join(os.getcwd(), "AzureMLFramework")
os.makedirs(script_folder, exist_ok=True)

#Upload Data
ds = ws.get_default_datastore()
print(ds.datastore_type, ds.account_name, ds.container_name)
#ds.upload(src_dir=data_folder, target_path='mnist', overwrite=True, show_progress=True)
```

view rawCreateDatastore.py hosted with ♡ by GitHub

Create or Attach a Compute Resource

The code below is one of the most basic variations on creating a Compute Cluster that I've found in my research. This version creates a cluster using the VM size and Max Node options from the Azure ML Compute Cluster interface. This section of code is expandable beyond what we're showing here and is where much of Azure ML service's scalability is based. You can choose from dozens of options in the Azure Cloud to spin up various types of clusters. These clusters also come preloaded with your environment, whether it's Anaconda, Spark, or TensorFlow, to name a few. You will need to add any packages that aren't preloaded into the environments, which may take a little trial and error, but you won't need to create a Python Environment from scratch.

Once this code completes, a Compute Cluster will appear under the Compute section in the AzureML Workspace resource in the Azure Portal.

```
# Compute cluster creation.
```

```
from azureml.core.compute import ComputeTarget, AmlCompute
from azureml.core.compute_target import ComputeTargetException

# Choose a name for your CPU cluster
cpu_cluster_name = "AzureMLFramework"

# Verify that cluster does not exist already
try:
    cpu_cluster = ComputeTarget(workspace=ws, name=cpu_cluster_name)
    print('Found existing cluster, use it.')
except ComputeTargetException:
    compute_config = AmlCompute.provisioning_configuration(vm_size='STANDARD_D2_V2',
                                                           max_nodes=4)
    cpu_cluster = ComputeTarget.create(ws, cpu_cluster_name, compute_config)

cpu_cluster.wait_for_completion(show_output=True)
```

view rawComputeClusterCreation.py hosted with ♡ by GitHub

Environment Set Up

In this step, we need to import our Environment package from the azureml.core, if we haven't already done so. We must also initialize the Environment object for later steps. If you're not adding any packages that aren't preinstalled on the Compute Cluster, you can skip the "adding packages" steps. I would, however, advise leaving in the step that saves the YAML file to your workspace, as you may need it for later deployments. Also, be aware that some packages are not registered with Anaconda and may need to be loaded from PIP (the package installer for Python), or some other resource. I've added a Conda and a PIP example in the code below (conda installs and manages packages from the Anaconda repository.)

```
from azureml.core import Environment
from azureml.core.conda_dependencies import CondaDependencies

fwrk = Environment("fwrk")

fwrk.docker.enabled = True
fwrk.python.conda_dependencies = CondaDependencies.create(conda_packages=['scikit-learn',
                                                                          'pandas',
                                                                          'numpy',
                                                                          'seaborn',


'category_encoders',
```

```
                                                    'lightgbm',
                                                    'papermill'
                                                ])
    fwrk.python.conda_dependencies.add_pip_package("inference-schema[numpy-support]")
    fwrk.python.conda_dependencies.save_to_file(".", "fwrk.yml")
```

view rawEnvironmentSetup.py hosted with ♡ by GitHub

Train the Model

This section describes the three steps involved in training and executing an experiment in the Azure ML Service:

1. Create the Training Script
2. Submit the Training Job to the Compute Cluster
3. Register the Model

Create the Training Script

The training script is essentially the guts of the Azure ML Service. Every other section in this post is Azure ML SDK code. This section is based almost entirely on the code you are using to produce your AI model. I recommend writing this section outside of your Azure ML service deployment; doing so allows you to develop and test your code without the overhead of developing the Azure ML service deployment at the same time. That being said, for this code to work inside of the Azure ML service, the last few lines that output the pickle file are mandatory. The pickle file is what is used to register the Model inside of the Azure ML service.

```
%%writefile $script_folder/train.py

import argparse
import os
import numpy as np

from sklearn.svm import SVC
from sklearn.externals import joblib
import pickle

from azureml.core import Run

# let user feed in 2 parameters, the location of the data files (from datastore), and the
regularization rate of the logistic regression model
```

```
parser = argparse.ArgumentParser()
parser.add_argument('--data-folder', type=str, dest='data_folder', help='data folder
mounting point')
parser.add_argument('--regularization', type=float, dest='reg', default=0.01,
help='regularization rate')
args = parser.parse_args()

# height, width, shoe size
X = [[181, 80, 44], [177, 70, 43], [160, 60, 38], [154, 54, 37], [166, 65, 40], [190, 90,
47], [175, 64, 39],
     [177, 70, 40], [159, 55, 37], [171, 75, 42], [181, 85, 43]]

Y = ['male', 'male', 'female', 'female', 'male', 'male', 'female', 'female', 'female',
'male', 'male']

clf = SVC()
clf = clf.fit(X, Y)

print('Predicted value:', clf.predict([[190, 70, 43]]))
print('Accuracy', clf.score(X,Y))

print('Export the model to model.pkl')
f = open('fwrk.pkl', 'wb')
pickle.dump(clf, f)
f.close()

print('Import the model from model.pkl')
f2 = open('fwrk.pkl', 'rb')
clf2 = pickle.load(f2)

X_new = [[154, 54, 35]]
print('New Sample:', X_new)
print('Predicted class:', clf2.predict(X_new))

os.makedirs('outputs', exist_ok=True)
# note file saved in the outputs folder is automatically uploaded into experiment record
joblib.dump(value=clf, filename='outputs/fwrk.pkl')
```


Submit the Training Job to the Compute Cluster

In this step, we'll create an object to run our script against the Compute Cluster we created earlier. The object needs the script directory and the training script to initialize. We'll then run configuration functions against the object to set the target Compute Cluster and the Environment we will use in the training. And lastly, we will execute the configured run script job.

When this code section completes, an Experiment will show up in the Azure portal in your Azure ML workspace resource. Also, assuming your Experiment has a pickle file output, that file is created here.

```
from azureml.core import ScriptRunConfig
from azureml.core.runconfig import DEFAULT_CPU_IMAGE

src = ScriptRunConfig(source_directory=script_folder, script='train.py')

# Set compute target to the one created in previous step
src.run_config.target = cpu_cluster.name

# Set environment
src.run_config.environment = fwrk

run = exp.submit(config=src)
run
```

view rawSubmitJobToCluster.py hosted with ♡ by GitHub

And we can watch the execution of the Training with the following code.

```
%%time
# specify show_output to True for a verbose log
run.wait_for_completion(show_output=True)
```

view rawSubmitJobToCluster2.py hosted with ♡ by GitHub

Register the Model

This last step in Training the Model takes the earlier-mentioned pickle file, output in the training script creation and execution, and registers it in the Azure ML service workspace.

Once this code completes, a Model will show up in the Azure portal.

```
# register model
model = run.register_model(model_name='fwrk', model_path='outputs/fwrk.pkl')
print(model.name, model.id, model.version, sep='\t')
```

```
from azureml.core import Workspace
from azureml.core.model import Model
import os
ws = Workspace.from_config()
model=Model(ws, 'fwrk')

model.download(target_dir=os.getcwd(), exist_ok=True)

# verify the downloaded model file
file_path = os.path.join(os.getcwd(), "fwrk.pkl")

os.stat(file_path)
```

view rawRegisterModel.py hosted with ♡ by GitHub

Deploy the Model as an Azure ML Service

This last section describes how to deploy the Model and create a web service that can be used for scoring new data.

1. Create the Scoring Script
2. Deploy the Azure Container Instance
3. Test the Deployed Service

Create the Scoring Script

The scoring script is used to return predictions from your registered model. This script has two mandatory sections and one optional section. The scoring script requires an INIT function and a RUN function to be defined. The INIT function connects the scoring script to the Model we deployed in the previous section. The RUN function executes the predict function from whatever package you used to do your training. For example, in the scikit-learn package the predict function is called predict(). Other packages may use a differently named function to do the same thing. The optional third section defines a schema for the inputs and outputs for scoring. In some software this schema is a requirement. For example, in the webinar BlueGranite did on this subject, we to connect to the Model and get predictions. Power BI dataflows require creation of a schema. As most other requests to this API will not require the schema, it can be skipped.

```
%%writefile score.py
import json
import numpy as np
import os
import pickle
import pandas as pd
from sklearn.externals import joblib
from sklearn.svm import SVC

from inference_schema.schema_decorators import input_schema, output_schema
from inference_schema.parameter_types.numpy_parameter_type import NumpyParameterType
from inference_schema.parameter_types.pandas_parameter_type import PandasParameterType

from azureml.core.model import Model

def init():
    global model
    #model = joblib.load('recommender.pkl')
    model_path = Model.get_model_path('fwrk')
    model = joblib.load(model_path)

input_sample = pd.DataFrame(data=[{
              "input_name_1": 5,         # This is a decimal type sample. Use the data
type that reflects this column in your data
              "input_name_2": 5,    # This is a string type sample. Use the data type
that reflects this column in your data
              "input_name_3": 3           # This is a integer type sample. Use the data
type that reflects this column in your data
           }])

output_sample = np.array([0])              # This is a integer type sample. Use the data
type that reflects the expected result

@input_schema('data', PandasParameterType(input_sample))
@output_schema(NumpyParameterType(output_sample))

def run(data):
    try:
        result = model.predict(data)
        # you can return any datatype as long as it is JSON-serializable
        return result.tolist()
    except Exception as e:
        error = str(e)
```

```
            return error
```

view rawCreateScoringScript.py hosted with ♡ by GitHub

Deploy in an Azure Container Instance

Now that we have a trained model and the ability to score data input to the model, we need to wrap the scoring script in a container and deploy the Container as a Service (CaaS) to the Azure ML service. In this example, we'll deploy the service as an Azure Container Instance (ACI). This first section of code adds some metadata to the container configuration.

```
from azureml.core.webservice import AciWebservice

aciconfig = AciWebservice.deploy_configuration(cpu_cores=1,
                                               memory_gb=1,
                                               tags={"data": "SVM",  "method" :
"sklearn"},
                                               description='Predict gender with sklearn
SVM')
```

view rawDeployACI.py hosted with ♡ by GitHub

And the second section of code creates the Image and deploys it as a service.

When this section completes, the ACI will show up under Images in the Azure ML service user interface, and the web service will appear under Deployments.

```
%%time
from azureml.core.webservice import Webservice
from azureml.core.image import ContainerImage
from azureml.exceptions import WebserviceException

# configure the image
image_config = ContainerImage.image_configuration(execution_script="score.py",
                                                   runtime="python",
                                                   conda_file="fwrk.yml")


service_name = 'fwrk'

# delete service if it exists
try:
    service = Webservice(ws, name=service_name)
    if service:
        service.delete()
```

```
except WebserviceException as e:
    print()

service = Webservice.deploy_from_model(workspace=ws,
                                       name=service_name,
                                       deployment_config=aciconfig,
                                       models=[model],
                                       image_config=image_config)

service.wait_for_deployment(show_output=True)
```

view rawDeployACI2.py hosted with ♡ by GitHub

Test the Deployed Service

We can test our deployed service, using test data in JSON format, to make sure the web service returns a result.

```
import requests
import json

headers = {'Content-Type':'application/json'}

if service.auth_enabled:
    headers['Authorization'] = 'Bearer '+service.get_keys()[0]

print(headers)

test_sample = json.dumps({'data': [[190, 70, 43]]})

response = requests.post(service.scoring_uri, data=test_sample, headers=headers)
print(response.status_code)
print(response.elapsed)
print(response.json())
```

view rawTestDeployedService.py hosted with ♡ by GitHub

Additional Resources

Think of this tutorial as a basic framework; a starting point to developing your own Azure Machine Learning deployments tailored to your company's needs. We also have a couple of other blog posts at BlueGranite that add to what we've discussed here.

To see a webinar in which I give a more complex demo of Azure ML capabilities, click HERE.

Andy Lathrop, a BlueGranite principal, wrote a blog post describing how we used Azure ML to build a personalized marketing model using the Microsoft Recommenders GitHub code: https://www.blue-granite.com/blog/introduction-to-personalized-marketing-with-azure-machine-learning

And be sure to also explore this blog post, by BlueGranite Senior Consultant David Eldersveld, a Microsoft MVP, on using Azure ML deployments in a Power BI Dataflow: https://www.blue-granite.com/blog/enrich-power-bi-data-with-ai-insights-from-cognitive-services

David's post makes use of both Azure's Cognitive Services and Machine Learning.

The code in this post was developed using the sample code included in the Azure ML service and the Microsoft Recommenders GitHub repository: https://github.com/microsoft/recommenders

And you can find copies of the Jupyter notebooks I used to create this post and the webinar demo here: https://github.com/datascinerd/AzureML-Examples

Want to find out more about how Azure ML can fit into your advanced analytics strategy? Contact us today to learn more.

ABOUT THE AUTHOR

JASON CANTRELL

Jason is a Senior Consultant at BlueGranite with a passion for data science. He has over 10 years of experience delivering Data and Analytics solutions for various employers. Jason also has over 6 years of experience delivering Data and Analytics solutions in a consulting role, which includes technical pre-sales, solution architecture design, development, and implementation.
LinkedIn Twitter