Udacity: Machine Learning Engineer

Contents

1 Software Engineering Fundamentals

1.1 Software Engineering Practices

Modular Code: putting functions into separate files to be imported into workspace.

Refactoring: restructuring your code to improve its internal structure, without changing its external functionality. This means cleaning and modularizing your program after it is working.

Optimization: we want to write efficient code, so this can be either fast execution or taking up less space in memory. We also want to *vectorize* our code for speed and amount of coding used.

```python
for book in recent_books:
  if book in coding_books:
    recent_coding_books.append(book) # 16.63 sec

recent_coding_books = np.intersect1d(recent_books, coding_books) # 0.035 sec
recent_coding_books = set(recent_books).intersection(coding_books) # 0.0097 sec

for cost in gift_costs:
  if cost < 25:
    total_price += cost * 1.08  # 5.55 sec

total_price = np.sum(gift_costs[gift_costs < 25] * 1.08) # 0.084 sec

```

Git Branches: to switch to a branch in a repository you use *git checkout (branchname)*.
To create and switch to a new branch you use *git checkout -b (newbranch)*.
When in main branch, merge another branch by using *git merge -no-ff (branchname)*.

Previous Code: to see previous commits use *git log*.
Using the commit message number, open the code using a new branch *git checkout (commit#)*.

Unit Testing: pytest is a tool we can use to make sure our function is outputting correctly. We can create a test file starting with *test_* and we get a . if we pass and an F if we fail.

Test Driven Deployment: writing tests before you write the code that's being tested. Your test would fail at first, and you'll know you've finished implementing a task when this test passes.

Virtual Environment: when creating packages, you want to run a program in a virtual environment so the install does not mess up with original Python installation. Doing this means all packages installed in the virtual environment only exist within it, and will be removed after closing it.
To create the virtual environment we run *python -m venv (env_name)*
To move to the virtual environment we run *source (env_name)/bin/activate*

Creating Packages: in the main directory, we have a *setup.py* file and a package folder containing all necessary files. The *setup.py* contains information about the package using *setup* from *setuptools*. In the package folder, we have our Python modules and the *__init__.py* file. When in the main directory, use *pip install .* to install package into environment (on a personal machine use virtual environment above).
To update a package after changes are made, run *pip install - - upgrade*

Uploading Packages: within the package directory including all Python modules you also need a *README.md*, *license.txt*, and *setup.cfg* files. Click here to see how to set these up.
To create the distribution package, run *python setup.py sdist*
To upload to test.pypi run *twine upload - -repository-url https://test.pypi.org/legacy/ dist/**
To install from test.pypi run *pip install –index-url https://test.pypi.org/simple/ (packagename)*
To upload to pypi run *twine upload dist/**
To install package from pypi you now run *pip install (packagename)*

2 Machine Learning in Production

2.1 Introduction to Deployment

Workflow: explore/process data, modeling, and deployment. Click here for AWS workflow.

Production Environment: the application that customers use to receive predictions from the deployed model.

Endpoint: the interface to the model which allows the application to send user data to the model and receives predictions back from the model about that data. Think of this as the python program being the application and a function calling the model being the endpoint.

Application: a web or software application that enables the application users to use the model to retrieve predictions.

Container: a standardized collection/bundle of software that is to be used for the specific purpose of running an application. This is used to create the computational environments for the model and application. A common container is *Docker*.

- **Layers** (bottom to top): infrastructure (data center), operating system, container engine (Docker), libraries/binaries, and application.
- **Script**: the instructions used to create a container (dockerfiles).

Deployment: versioning (indicate a models current version), monitoring (model continues to meet performance metrics), update (when model fails to meet performance use new data to update), routing (test model performance compared to other variants), and predictions (*on-demand* used for customers vs *batch* used in business).

2.2 Building a Model with SageMaker

NOTE: refer to BH - XGBoost (Batch Transform) - Low Level for details about API.

```python
import sagemaker
from sagemaker import get_execution_role
from sagemaker.amazon.amazon_estimator import get_image_uri
from sagemaker.predictor import csv_serializer

```

Session (doc): a special object that allows you to do things like manage data in S3 and create and train any models. We use this throughout the notebook workspace in SageMaker.

```python
session = sagemaker.Session() # create session object to be used throughout workspace

```

Role: the IAM role created when you create a notebook, this defines how data that your notebook uses/creates will be stored. We will use this later for training.

```python
role = get_execution_role() # create IAM role object to be used throughout workspace

```

S3 Bucket: When a training job is constructed using SageMaker, a container is executed which performs the training operation and accesses data in S3. This means that we need to upload the data we want to use to S3. When we perform a batch transform job, SageMaker expects the input data to be stored on S3. We upload data to S3 using the session object after saving the DataFrame to a csv file.

```
prefix = 'boston-xgboost-HL'

test_location = session.upload_data(os.path.join(data_dir, 'test.csv'),
                                    key_prefix=prefix)
val_location = session.upload_data(os.path.join(data_dir, 'validation.csv'),
                                   key_prefix=prefix)
train_location = session.upload_data(os.path.join(data_dir, 'train.csv'),
                                     key_prefix=prefix)

```

Estimators (doc): an object that specifies some details about how a model will be trained. It gives you the ability to create and deploy a model. It requires a container which can be obtained from the session object. We can also set hyperparameters on this estimator object.

```
# Construct the image name for the training container.
container = get_image_uri(session.boto_region_name, 'xgboost')

xgb = sagemaker.estimator.Estimator(container, # Image name of the training container
        role, # The IAM role to use
        train_instance_count=1, # The number of instances to use for training
        train_instance_type='ml.m4.xlarge', # Type of instance to use for training
        output_path='s3://{}/{}/output'.format(session.default_bucket(), prefix),
        # output_path is where to save the output (the model artifacts)
        sagemaker_session=session) # The current SageMaker session

xgb.set_hyperparameters(...)   # set hyperparameters here

```

Input: we specify where in S3 and the type of the data that we will feed into our estimator object.

```
# A wrapper around the location of our train and validation data, to make sure that
# SageMaker knows our data is in csv format.
s3_input_train = sagemaker.s3_input(s3_data=train_location, content_type='csv')
s3_input_validation = sagemaker.s3_input(s3_data=val_location, content_type='csv')

xgb.fit({'train': s3_input_train, 'validation': s3_input_validation})

```

Transformer (doc): used to create a transform job and evaluate a trained model. We specify the location of the test data and the format it is in. We can then use the *wait()* method to see the progress on testing and when we can resume coding.

```
xgb_transformer = xgb.transformer(instance_count = 1, instance_type = 'ml.m4.xlarge')
xgb_transformer.transform(test_location, content_type='text/csv', split_type='Line')
xgb_transformer.wait()

```

Predictions: predictions are stored on S3, but we can download them into a specific directory.

```
!aws s3 cp --recursive $xgb_transformer.output_path $data_dir
Y_pred = pd.read_csv(os.path.join(data_dir, 'test.csv.out'), header=None)

```

2.3 Deploying and Using a Model

NOTE: refer to BH - XGBoost (Deploy) - Low Level for details about API.

Deploying: using the high level API we can use the *deploy* method to create an endpoint for our model. Note that this creates a compute instances and needs to be shutdown when not in use.

Predicting: Now that we have deployed our endpoint, we can send the testing data to it and get back the inference results. When using the created endpoint it is important to know that we are limited in the amount of information we can send in each call (note here that the data is small enough to send in one file, but for larger files we need to break it up and send in chunks).

```
# Deploy model and created endpoint for predicting
xgb_predictor = xgb.deploy(initial_instance_count=1, instance_type='ml.m4.xlarge')

# We need to tell the endpoint what format the data we are sending is in
xgb_predictor.content_type = 'text/csv'
xgb_predictor.serializer = csv_serializer
Y_pred = xgb_predictor.predict(X_test.values).decode('utf-8')

# Y_pred is currently a comma delimited string and so we would like to break it up
# as a numpy array.
Y_pred = np.fromstring(Y_pred, sep=',')

xgb_predictor.delete_endpoint() # shut down endpoint after no longer in use

```

2.3.1 Deploying to a Web App

NOTE: refer to IMDB - XGBoost - Deploy for more details about deployment code.

Overview: Only authenticated users can access the SageMaker API, so we will create a new endpoint which does not require authentication and which acts as a proxy for the SageMaker endpoint. We can do this through the *Lambda* and *API Gateway* services, which works in the following steps: a user submits data and our app sends it to the API Gateway endpoint, the API sends this to the Lambda function, Lambda then processes and sends the data to our model, our model makes inference and sends the prediction back through, and finally the data is displayed to the user.

Response: After accessing the SageMaker runtime, we can invoke the endpoint to create and HTML response that contains the predicted response in the 'Body' section.

```
import boto3

# Create the endpoint backup (Lambda does not have access to this)
xgb_predictor = xgb.deploy(initial_instance_count=1, instance_type='ml.m4.xlarge')

# Get handle for SageMaker runtime (API for Lambda to use)
runtime = boto3.Session().client('sagemaker-runtime')

response = runtime.invoke_endpoint(EndpointName = xgb_predictor.endpoint,
                                   ContentType = 'text/csv', # Data format
                                   Body = ...) # The data to be predicted on

response = response['Body'].read().decode('utf-8')

```

Lambda: a service which uses a Python code file and executes whenever a chosen trigger occurs. When it is executed it will receive the data, perform any sort of processing that is required, send the data to the SageMaker endpoint we've created and then finally return the result.

API Gateway: a service that allows you to create HTTP endpoints (url addresses) which are connected to other AWS services. One of the benefits to this is that you get to decide what credentials, if any, are required to access these endpoints (in our case it is open to the public).

NOTE: refer to IMDB - XGBoost - Deploy for details about how to set up Lambda and Gateway.

2.4 Hyperparameter Tuning

NOTE: refer to IMDB - XGBoost (Hyperparameter) - Low Level for details about low level API.

Base Model: we create a model similar to how we have done previously, which will be our *base model* that SageMaker will tune for us. We also set hyperparameters for the base model similar to how we have done before. *(See 2.2 for how to set up base model).*

Tune Object: We create a hyperparameter tuning object where we pass a model, the measurement metric, the objective, how many models to fit, and ranges for our hyperparameters we want to tune. We then fit this object and wait for training to finish in order to find the best model.

```
from sagemaker.tuner import IntegerParameter, ContinuousParameter,
  HyperparameterTuner

xgb_hyperparameter_tuner = HyperparameterTuner(estimator = xgb, # Base model
      objective_metric_name = ’validation:rmse’, # Metric used to compare models
      objective_type = ’Minimize’, # Minimize or maximize the metric
      max_jobs = 20, # The total number of models to train
      max_parallel_jobs = 3, # The number of models to train in parallel
      hyperparameter_ranges = { # Hyperparameters ranges to be tuned
        ’max_depth’: IntegerParameter(3, 12),
        ’eta’      : ContinuousParameter(0.05, 0.5),
        ’min_child_weight’: IntegerParameter(2, 8),
        ’subsample’: ContinuousParameter(0.5, 0.9),
        ’gamma’: ContinuousParameter(0, 10),
      })

s3_input_train = sagemaker.s3_input(s3_data=train_location, content_type=’csv’)
s3_input_validation = sagemaker.s3_input(s3_data=val_location, content_type=’csv’)

xgb_hyperparameter_tuner.fit({’train’: s3_input_train,
                            ’validation’: s3_input_validation})
xgb_hyperparameter_tuner.wait()


```

Best Model: after tuning, we can then access the best fit model and attach it to an estimator object which we can use for predicting. We will create a transform object from this model, which we then pass our testing data to.

```
# Attach the best model to an Estimator to create a new model
xgb_attached = sagemaker.estimator.Estimator.attach(xgb_hyperparameter_tuner.
                                                    best_training_job())

# Create transformer object using the hypertuned model
xgb_transformer = xgb_attached.transformer(instance_count = 1,
                                           instance_type = ’ml.m4.xlarge’)
xgb_transformer.transform(test_location, content_type=’text/csv’, split_type=’Line’)
xgb_transformer.wait()

```

2.5 Updating a Model

2.5.1 Multiple Models

NOTE: refer to BH - Updating an Endpoint for details about setting up models.

A/B Testing: in some cases we may create two separate models for the same problem and want to see which performs better. By using the low level endpoint API we can set it up so that data is split and sent to both models.

Endpoint Setup: SageMaker provide functionality for this by letting us pass multiple models in the ProductionVariants when creating an endpoint. The amount of data sent to each model is determined by the weight we assign it (assigned weight divided by total weight of all models).

```
# Give our endpoint configuration a name which should be unique
combined_endpoint_config_name = "boston-combined-endpoint-config-" +
                                strftime("%Y-%m-%d-%H-%M-%S", gmtime())

# We then ask SageMaker to construct the endpoint configuration
combined_endpoint_config_info = session.sagemaker_client.create_endpoint_config(
    EndpointConfigName = combined_endpoint_config_name,
    ProductionVariants = [
        { # First we include the linear model
          "InstanceType": "ml.m4.xlarge",
          "InitialVariantWeight": 1, # 1/2 of total data
          "InitialInstanceCount": 1,
          "ModelName": linear_model_name,
          "VariantName": "Linear-Model"
        }, { # Second we include the xgb model
          "InstanceType": "ml.m4.xlarge",
          "InitialVariantWeight": 1, # 1/2 of total data
          "InitialInstanceCount": 1,
          "ModelName": xgb_model_name,
          "VariantName": "XGB-Model"
        }])

# Again, we need a unique name for our endpoint
endpoint_name = "boston-update-endpoint-" + strftime("%Y-%m-%d-%H-%M-%S", gmtime())

# Then we can deploy our endpoint
endpoint_info = session.sagemaker_client.create_endpoint(
    EndpointName = endpoint_name,
    EndpointConfigName = combined_endpoint_config_name)

endpoint_dec = session.wait_for_endpoint(endpoint_name)

```

Updating Endpoint: After running and A/B we determine which model best suits our problem, and we only want to send data to one model. We can update the endpoint without having to shut it down by using the SageMaker function *update*, which starts a new endpoint in the background and replaces the existing endpoint once the new one is running. This means no down time.

```
session.sagemaker_client.update_endpoint(
    EndpointName=endpoint_name, # endpoint with both models
    EndpointConfigName=linear_endpoint_config_name) # endpoint from linear model

endpoint_dec = session.wait_for_endpoint(endpoint_name) # wait for endpoint to update

```

2.5.2 Data Distribution Change

NOTE: refer to IMDB - XGBoost - Updating a Model for more details.

Change in Data: sometimes the distribution of the data submitted to our model can change and in-turn, our model starts to perform worse at predicting. We can import new data, upload to S3, and use the transform object to gauge how our current model is performing on the new data. Explore the data (in this case, compare 5000 most common words in each dataset to each other and find the difference).

Change the Model: if we believe that something in our data has changed, we can create and train a new model to replace the existing one. We create new training and validation sets, save them, and upload to S3 to be used in the model. We create a new estimator and set the hyperparameters as well. We then fit the model to the new data, and use a transform object to test on the new data. We can also test the new model on the old test data to see how it performs. *Note that if we had deployed our XGBoost model like we did in the Web App notebook then we would need to implement this vocabulary change in the Lambda function as well.*

Update the Model: we have a new model that we'd like to use instead of one that is already deployed. Furthermore, we are assuming that the model that is already deployed is being used in some sort of application. As a result, what we want to do is update the existing endpoint so that it uses our new model. Using the low level approach, we create a new endpoint configuration that we will update the existing one with.

```
# Create unique name for the endpoint
new_xgb_endpoint_config_name = "sentiment-update-xgboost-endpoint-config-" +
                               strftime("%Y-%m-%d-%H-%M-%S", gmtime())

# Set up the new endpoint configuration with the updated model
new_xgb_endpoint_config_info = session.sagemaker_client.create_endpoint_config(
    EndpointConfigName = new_xgb_endpoint_config_name,
    ProductionVariants = [{
      "InstanceType": "ml.m4.xlarge",
      "InitialVariantWeight": 1,
      "InitialInstanceCount": 1,
      "ModelName": new_xgb_transformer.model_name,
      "VariantName": "XGB-Model"
    }])

# Update the endpoint
session.sagemaker_client.update_endpoint(EndpointName=xgb_predictor.endpoint,
    EndpointConfigName=new_xgb_endpoint_config_name)

session.wait_for_endpoint(xgb_predictor.endpoint)

```

3 ML Case Studies

3.1 Population Segmentation (Unsupervised)

NOTE: refer to Pop_Segmentation in the case_studies folder for the code.

Population Segmentation: segmenting the population based on shared demographic traits. We will use PCA to reduce the dimension of our data (from 34 to 7) and KMeans to group the population into clusters based on this.

PCA: attempts to reduce the number of features within a dataset while retaining the "principal components", which are defined as *weighted* combinations of existing features that:

1. Are uncorrelated with one another, so you can treat them as independent features.
2. Account for the largest possible variability in the data!

So, depending on how many components we want to produce, the first one will be responsible for the largest variability on our data and the second component for the second-most variability, and so on. The idea is that components that cause a larger variance will help us to better differentiate between data points and (therefore) better separate data into clusters.

SageMaker PCA: we can reduce dimensionality with the built-in SageMaker model for PCA. We first must specify an IAM role and an S3 bucket to save model attributes. Click here for the SageMaker documentation for the parameters that need to be passed to the model. An important note is that the *num_components* parameter is just one less than the total number of features, and we just choose how many features to keep later on.

Formatting Training Data: to prepare the data for a built-in model, we must convert the DataFrame to a NumPy array, then convert that into a RecordSet format. This is required for all built in models, and allows SageMaker to perform faster with large datasets.

Heatmap: since we can't visualize 7-dimensional data, we can use a heatmap to see which components carry the most weights for each cluster. This is one way to visualize results from PCA and clustering. We can also go backwards and see which original features makes up the components by plotting or creating a DataFrame.

Follow this link for how to clean up after finishing a notebook.

3.2 Payment Fraud Detection (Supervised)

NOTE: refer to Fraud_Detection in the case_studies folder for the code.

LinearLearner: binary classification, in which a line is separating two classes of data and effectively outputs labels; either 1 for data that falls above the line or 0 for points that fall on or below the line. We will use this to separate valid and fraudulent transactions.

Precision: the number of true positives (truly fraudulent transaction data, in this case) over all positives, and will be the higher when the amount of false positives is low. We optimize based on this if we do not want any valid transactions to be categorized as fraudulent.

Recall: the number of true positives over true positives plus false negatives and will be higher when the number of false negatives is low. We optimize based on this if we want to catch almost all cases of fraud, even if it means a higher number of false positives.

Class Imbalance: even if the model labels all of the data as valid, we will have a high accuracy (over 99%) with only around 0.017% of the data is fraudulent. We can balance classes by assigning a higher weight to the positive (fraudulent) examples, which SageMaker allows us to do.

Model tuning: this is when we optimize a model based on a specific metric (accuracy, recall, precision, etc.). SageMaker provides us a number of ways to automatically do this. For the LinearLearner model, we can specify the model selection criteria as well as the target value.

3.3 Custom Models (PyTorch)

NOTE: refer to Moon_Classification in the case_studies folder for the code.

Training Script: in the *train.py* file, we import the model created in *model.py* and set up all code to be executed when fitting the estimator. This includes hyperparameters, loading data, saving the model, and defining any other functions needed for training.

Estimator: we first train an estimator, specifying the training script and any hyperparameters we want for our model. Note that this requires a *train.py* file for an entry point.

PyTorchModel: for PyTorch, creating a model and an endpoint are separate (no *deploy* function). We need to create a PyTorchModel object, which will use our estimators attributes from training and a *predict.py* file. This is what we will use to create an endpoint.

Endpoint: we can then deploy our PyTorchModel object to create an endpoint which we can use to make predictions.

3.4 Time-Series Forecasting (RNN)

NOTE: refer to Energy_Consumption in the case_studies folder for the code.

Forecasting: looking at historical data and trying to predict the future.

Context Length: the amount of data recorded in the past that the algorithm will see as input. The longer the context length the better the model will be a predicting.

Prediction Length: using the context length to predict the pattern found in the context. In general, data closest to the prediction time frame will contain the information that is most influential in defining that prediction.

DeepAR (doc): a supervised learning algorithm for forecasting scalar (one-dimensional) time series using recurrent neural networks (RNN). It looks at all the training time series and tries to identify similarities across them by randomly sampling training examples from them. See in-depth details about training here. This algorithm is best for data that has recurring patterns.

NaN Values: since we are doing long-term forecasting, it is okay to fill *NaN* values with the column mean. But for short-term analysis (hourly) we may just want to drop *NaN* values.

3.5 Plagiarism Detector (Project Notes)

Containment: features that look at a whole body of text and count of occurrences of words in text files. We calculate this using *n-grams* (group of words) and the formula: $\frac{count(ngram_A) \cap count(ngram_S)}{count(ngram_A)}$. If the two texts have no n-grams in common, the containment will be 0, but if all their n-grams intersect then the containment will be 1.

Longest Common Subsequence: the longest sequence (left to right) that appears in both texts (words don't have to be continuous).

4 NLP Fundamentals

5 Convolutional Neural Networks