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.

```
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.

```
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.

```
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.

```
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