MLE Exercise 26 Report

By Alex O'Neill

Justification

I began by carefully reading over the instructions and jupyter notebook report. Once I read through the report, I began recreating the model in model_builder.py. I went through all the code from the notebook and began refactoring and modularizing code that was used repeatedly. Especially the feature engineering and data prepping sections since I knew this would be needed for the json from the api calls. Feature engineering just fixes two columns of strings and converts them to floats. Data prepping trains and uses both an imputer and standard scaler to fill in NaN values. It also takes the remaining string columns and expands the string values into columns and replaces the string with a value of 1 under the corresponding column. Get variables cleanse the training data and use the same logistic regression as the notebook to find the variables needed for predictions. Build model is used to fit a model with the cleaned training data and variables. The model created is saved as a pickle file so the model won't need to be trained every time the class is created. I then added the api_data_cleaning, predict, and get_bins methods to interface with the flask server. All three return a data frame that works on the data as needed. I decided keeping the data as a dataframe would make it easier to work with and help to maintain consistency across methods. Some room for improvement could be separating those three methods from the model builder and creating a model_interface class. To save on time, I kept them as one class to spend more time debugging and testing.

The server.py file uses a flask server to route and handle calls. First I checked the data to make sure it was the correct format. From there, the data was turned into a dataframe and prepped. After being prepped, the phat predictions were found and added to the dataframe of variables used. Finally, the business logic was made and that filtering was applied to the phat column and saved in another dataframe and added with the last one that was created. To finish the requirements, the columns were reordered alphabetically and then the dataframe was made into a json object to be returned to the user.

To make sure the code was robust, I added a logger and log statements to track what the code was doing during various test calls. I also created two test files to test the server and model files. To make sure the server wouldn't crash, try/except blocks of code were wrapped around all methods and functions to handle any errors. Logs and tests confirmed that the system could handle errors and not bring down the server. In the case of an error, the server responds with a 400 status code as opposed to a 200 code to let the user know there was an error along with a json object that contains an error message.

Lastly, since flask is a development server, I wrapped the server code with a wsgi.py file and used gunicorn to deploy the api. I only used 4 workers, but that could be adjusted depending on need. Once the code was ready and fully tested, it was built with a Dockerfile to make an image that could be used to spin up a server. The shell file runs the docker build command to make a container from the image and makes sure it is listening on port 1313.

Areas to Optimize

There are two main areas I would like to optimize. The first area is how I implemented api_data_cleaning. This method takes the current dataframe of raw training data, appends the api data to it, prepares all the data together, and returns the cleaned api data. I implemented it this way so that the api data with string values could get the other string values needed to create all the columns. This can slow down the service since it always calls the training data and cleans the same couple thousand of rows on top of 1 to thousands of api rows of data. A much better approach that would optimize this method would be to run the same feature engineering and data prepping on just the api data and add any missing columns to the data frame by checking the cleaned dataframe, or even better by checking if any columns are missing from the variables needed for prediction since that would be less columns to check.

The second area I would improve is how the data is loaded for the server when server.py is run. Currently, it instantiates the GLM Model object which spends time calculating the variables, training the imputer and standard scaler, and calculates the bins. A better approach would be to load this data from a json file so it is fast to read the data instead of calculating. Those values could be passed to the object when being created so it wouldn't need to calculate and waste time. As this api scales, starting new instances would all have to load this data and that would impact the first few seconds while being spun up. By loading the data initially, the api can respond to thousands of calls quicker.

Link to my image on docker hub:
https://hub.docker.com/repository/docker/aj96oneill/python-ml-api