Write Up

Through my eyes and personal interpretation, the core of this exercise appeared to be focused on the usage of the already written code to generate a trained model, to use said model within another application which would then extract and transform any incoming data into a proper form understandable and useable by said model, and then to return these predicted results back to whomever is sending the data in the first place. In addition, the prediction application had to make use of an API and should be available under the form of a Docker Image and eventually container to maximize ease of use for the client-side users.

In terms of the coding carried out on my end, most of the underlying logic for the prediction application originated from the original model training python program, as to be able to use said model the inputs had to be in the same order and transformed in the same fashion as those originally used to train the model in the first place, requiring an input and transformation pipeline for incoming JSON of an identical nature. To achieve this, the SimpleImputer and StandardScalar used to transform said new data also had to be the same from the original model's training, requiring them to be saved and imported into the new prediction application. Of note was that depending upon the number of rows passed in and what column variable values might have been left blank, not all dummy variables might be properly generated during the transformation process, requiring the setting up of default columns with default values for those variable columns the prediction model needed.

The API of the application itself is simplistic, simply taking in JSON data either on its lonesome or in the form of a list, passing it through the prediction application and then returning those results meeting the probability threshold as described and demanded by the client via synchronous means. With further development the implementation of batching, both on the API's end and within the actual prediction application's code, would aid in handling much more important and frequent loads than in the current problem's context, but otherwise works quickly and effectively even when passing in a batch of 10,000 records at once.

The Docker Container which runs the prediction program and API makes use of uvicorn to get an actual server for the API to run on up and going.

Note than as is visible within the design of the 'requirements.txt' file, the python libraries included where specifically chosen so that both the model training and prediction environments are completely identical to one another, as well as to avoid any potential issue of serialization and non-compatible dependencies.

Test were carried out manually for the most part at all stages of development, so a more effective means would be to write up another small python program to automate this process through the usage of assertion, comparing what certain outputs given certain inputs are in comparison to what they should be, as given identical inputs into identical models, the results should stay the same.

From a CICD standpoint used Travis was utilized as I was already better accustomed and comfortable with it from past self-development and training, but long-term checking out and making use of GitHub Actions would make more sense.

If I were more knowledgeable and had more time, I would have also looked into creating a local Kubernetes cluster using kind to test Kubernetes based automation.

Finally, code was copy pasted from jupyter-notebooks used for development and testing into the main.py application once confirmed functional. There are more elegant ways to avoid this code duplication but I did not get to it due to time constraints.