- Summary of process to deploy Fast API:
    - The data preprocessing was transformed into a scikit-learn pipeline for simplicity. This makes it easy to save a .pkl object of the fully trained pipeline containing the mean imputation, scaling, one hot encoder, and the top 25 feature selection. There are custom functions and classes that were created for this that are in helper_functions.py.
    - The production model that was approved by the business partner was replicated as part of a training script model_train.py. This script was used to test the methods and classes that are used by the Fast API implementation. After comparing the summary output of the final trained model to what was in the attached notebook, everything seems good to deploy. The production model was also saved as a .pkl object for loading within the Fast API.
    - Deployment of the Fast API is done using Docker. This API can be extensible and scalable with the use of an automated load balancer e.g. Docker Swarm, AWS ECS, Nginx, or Traefik. Within the main.py module, the data preprocessing pipeline and production classification model are loaded into memory. For simplicity, all of the data processing, probability estimation, classification, and post processing are handled with the predict method defined as part of the POST request.

- Consider how your API might handle a large number of calls (thousands per minute). What additional steps/tech could you add to your deployment in order to make it scalable for enterprise level production?
    - Use an automated load balancer to distribute requests across multiple instances of the API. With Docker Swarm or AWS ECS, the number of instances concurrently running the Fast API will scale up and down automatically based on the incoming load. Autoscaling will effectively minimize the cost of cloud resources while maintaining reliability of the API.
    - It would be helpful to add error handlers to the API for evaluating input data. Currently, the assumption is that all data given to the API is reliable and as expected. This might become a problem if covariate drift occurs. For example, if new categories appear in one of the feature columns, the feature processing class could result in an error.
    - Reduce the time it takes to deploy new models by creating a general framework where models are hosted on AWS, GCP, or Azure. Continuing with the example of AWS ECS and Docker Swarm for handling load balancing and scaling, we can add a few additional elements:
        - Create a central repository where all model and preprocessing artifacts are versioned and stored for posterity.
        - Create a post deployment monitoring solution that tracks the performance of the model over time to ensure reliability.

        - Using an orchestration tool for scheduling automatic retraining of models on a fixed cadence to avoid problems with data drift, label shift, or concept drift.

- Notes on improving the classification model:
    - Model L1 penalty optimized using cross validation prior to feature selection.
    - Use median imputation of floating point features instead of mean as the median is more robust to outliers.
    - Consider using pairwise interaction features.
    - Imputation and standard scaling were fitted on the training split similar to what was done in the example notebook. Fitting the transformers on the entire training data set might make a difference in out of sample performance. The production model is exactly what was in the attached notebook.