End-to-End Process Explanation:

**1. Overview:**

This Flask API serves as an endpoint for making predictions using a pre-trained machine learning model. I have included the pretrained machine learning model. The model predicts a binary classification and takes input in the form of JSON data. The input data undergoes a series of preprocessing steps, including feature engineering, mean imputation, and creating dummy variables for categorical features. The predictions and relevant information are then returned as a JSON response.

**2. Code Structure and Modules:**

**2.1. api.py:**

**Loading the Model:**

The joblib library is used to load the pre-trained model (`trained\_model.joblib`) into memory.

**Flask App Initialization:**

An instance of the Flask application is created, and the JSON encoder is set to the default Flask JSON encoder.

**Prediction Route:**

The main route `/predict` is defined to handle POST requests. The route performs the following steps:

- Retrieves JSON data from the request.

- Validates the input data format.

- Processes the input data through custom functions for feature engineering, mean imputation, and creating dummy variables.

- Uses the loaded model to make predictions.

- Prepares a JSON response containing class probabilities, predicted classes, and input variables.

**Error Handling:**

Validation errors and other exceptions are caught, logged, and appropriate error responses are returned with status codes.

**Flask App Execution:**

The Flask app is set to run on port 1313 when the script is executed.

**2.2. custom\_functions.py:**

**Feature Engineering:**

- The feature\_engineering function processes the input data, handling currency symbols, percentages, and missing values in categorical columns.

**Mean Imputation:**

- The mean\_imputation function uses SimpleImputer and StandardScaler from scikit-learn to handle missing values and standardize the numerical data.

**Create Dummy Variables:**

- The create\_dummies function creates dummy variables for categorical columns using one-hot encoding. It considers predefined categories for specific columns.

**3. Choices Made:**

**3.1. Data Preprocessing:**

- Currency symbols and percentages are removed from relevant columns during feature engineering.

- Missing values in categorical columns are replaced with the label "Unknown."

- Mean imputation is used for numerical features.

**3.2. Flask API:**

- Flask is chosen for its simplicity and flexibility in building web APIs.

- Error handling is implemented to provide meaningful responses for validation errors and unexpected exceptions.

**4. Optimizations for Scalability:**

**4.1. Input Validation:**

- The API validates the input data format to ensure it is either a JSON list or object. This helps prevent invalid requests.

**4.2. Model Loading:**

- The model is loaded only once during the initialization of the Flask app, reducing redundant loading for each prediction request.

**4.3. Error Handling:**

- Proper error responses are provided for different scenarios, aiding in debugging and making the API more user-friendly.

**4.4. Asynchronous Processing:**

- For scalability, asynchronous processing (e.g., using Flask-RESTful or Flask-Asyncio) can be considered to handle multiple requests concurrently.

**4.5. Containerization:**

- Containerization tools like Docker can be used for packaging the API and its dependencies, ensuring consistent deployment across different environments. I tried the containerization and I am almost finished with it but need a little more time to finish it.

**4.6. Load Balancing:**

- Deploying the Flask app behind a load balancer can distribute incoming traffic across multiple instances, improving scalability and reliability.

**5. Conclusion:**

The Flask API, coupled with preprocessing functions and proper error handling, provides a robust and scalable solution for making predictions using a pre-trained model. Consideration of input validation, model loading, and error responses contributes to the overall reliability of the API. To further enhance scalability, additional optimizations like asynchronous processing and containerization can be explored based on specific deployment requirements.

**How to run the code:**

The requirements.txt file shows the different libraries and their versions used. The pre-trained model is also included with the Zip file. After installing the required libraries, open a command prompt/terminal and run the api.py file using the command **“python api.py”**. In a different terminal, use the command **“curl --request POST --url http://localhost:1313/predict --header "content-type: application/json" --data <the data in json format>”**.