Model Deployment and API Implementation

The model deployment process involved developing a Flask-based API to serve predictions, implementing robust preprocessing and validation mechanisms, and ensuring a seamless execution pipeline using Docker. The project followed a structured approach, emphasizing modularity, maintainability, and scalability.

API Development and Model Integration

The API was designed to handle inference requests efficiently while maintaining flexibility in input handling. The implementation adhered to best practices, such as dynamically processing feature variables rather than hard-coding them and ensuring robustness in handling missing data. The model_utils.py module was responsible for loading the model and encoders, preventing failures due to missing artifacts through structured exception handling. Additionally, input data preprocessing was encapsulated in preprocessing.py, ensuring feature transformations aligned with the model's training process.

During inference, the API applied preprocessing transformations, including encoding categorical variables and handling missing features by adding them with appropriate defaults. The prediction logic followed the predefined threshold of 0.75 for classification, as specified in the instructions. Output formatting was structured to ensure clarity, encapsulating business outcomes, probabilities, and original feature inputs in a structured JSON response.

To validate API behavior, unit tests were implemented to check various input scenarios, such as missing data, incorrect content types, and valid requests. These tests helped identify an issue where the API returned an unexpected error message when the JSON body was missing, which was then debugged and refined.

Dockerization and Deployment

To containerize the application for a consistent execution environment, a Dockerfile was created to package the necessary dependencies, model artifacts, and API logic. The shell script facilitated running the container, ensuring that the application launched correctly within the Dockerized environment. This approach eliminated system dependency inconsistencies, making the API easily deployable across different environments.

Once the container was running, inference requests were successfully tested, verifying that the model processed inputs correctly and returned structured predictions. Logging was integrated to capture API requests and processing steps, aiding in debugging and monitoring.

This deployment process demonstrated best practices in structuring API logic, handling model artifacts dynamically, ensuring proper data validation, and encapsulating the application within a Docker container for scalability. The modularized design enables future enhancements, such as integrating additional preprocessing steps, expanding model support, or optimizing deployment strategies.