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. Key elements of the implementation include:

- Dynamic Input Handling: No hard-coding of feature variables, with robust handling of missing data.
- Preprocessing: The preprocessing.py module ensured feature transformations aligned with the model's training process.
- Model Integration: The model_utils.py module handles loading the model and encoders, with structured exception handling to avoid failures due to missing artifacts.

During inference, preprocessing transformations were applied, including encoding categorical variables and managing missing features by adding appropriate defaults. The prediction logic followed a 0.75 classification threshold, and the output was returned as a structured JSON response containing business outcomes, probabilities, and original feature inputs.

Validation of API

To ensure proper functionality, unit tests were implemented to check the API's behavior under different input conditions. These included scenarios such as:

- Missing data: Tests were created to check how the API handles missing feature values.
- Incorrect content types: Ensured the API can handle and respond to wrong content type requests properly.
- Valid requests: Verifying that the API returns expected results for correctly formatted input.

These tests revealed an issue where the API returned an unexpected error when the JSON body was missing, which was identified and debugged.

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.

File References:

model_utils.py > model_utils.txt

preprocessing.py > preprocessing.txt

app.py > app.txt

model.pkl > model.txt

encoders.pkl > encoders.txt

mod_vars.pkl > mod_vars.txt

unit_test.py > unit_test.txt

DockerFile > DockerFile.txt

run_api.sh > run_api.txt