**ML Pipeline Implementation - Project Summary**

**Initial Setup**

For this project, I implemented an end-to-end machine learning pipeline using Flask, ensuring modularity and real-time monitoring. Initially, I used ChatGPT to generate code fragments necessary for the pipeline. After setting up the virtual environment and installing dependencies via the requirements.txt file, some conflicts were encountered when running the code. The issue was that the output was not displayed as expected. Instead, only a minimal message indicating that the "Credit Card Fraud Detection API successfully running" was visible without additional details. After troubleshooting and tweaking the implementation, I successfully integrated the API with a custom Front-End UI, making the system fully functional.

**Execution**

To execute the pipeline, the following steps were performed displayed in **Figure 1**:

The **run.py** file now handles both model hosting and API execution, reducing manual intervention.

A screenshot of a computer program

Description automatically generated

Figure 1: Execution Pipeline

**Training Process**

The model training process is implemented in **train\_model.py** seen in **Figure 2**. In this process, the cleaned dataset is first loaded and then split into features and target values. After scaling the features appropriately, a **LightGBM** model is trained using a carefully chosen configuration aimed at improving performance and efficiency. Finally, both the trained model and the scaler are saved for later use by the API. This process ensures that the model deployment is based on an optimized and robust training routine.

A screen shot of a computer program

Description automatically generated

Figure 2: Training Process with LightGBM

**Major Changes and Enhancements**

During the project, significant changes were made based on suggestions and performance observations:

1. **Model Upgrade:**

The initial approach was revised to switch to a **LightGBM** model, which offered improved efficiency and processing speed compared to the earlier algorithm.

1. **Front-End UI Improvements:**

The user interface was enhanced to improve **output readability and better visualize predictions**. These improvements solved the initial display issues, allowing users to see real-time responses clearly.

**Simulated Monitoring and Bottleneck Analysis**

Through simulated requests, the API performance was continuously monitored, allowing for the identification of occasional response-time spikes. This real-time monitoring was instrumental in informing further system optimizations.

**Figure 3** explains the server response time for requests made to the Credit Card Fraud Detection Model API.

* It imports necessary libraries such as requests, numpy, and matplotlib for handling HTTP requests, generating random test data, and plotting results, respectively.
* The script generates 20 random input samples, sends them to the API endpoint, and records the response time for each request.
* After executing the requests, it plots the response times against the request numbers, providing a visual representation of the server's performance.
* This script is essential for evaluating the efficiency and responsiveness of the API under multiple request scenarios.

A screen shot of a computer program

Description automatically generated

Figure 3: Client.py

**Client-Server Separation and Real-Time Monitoring**

The final design follows a well-structured separation of concerns:

* The **server (server.py)** manages model training, inference, and backend execution.

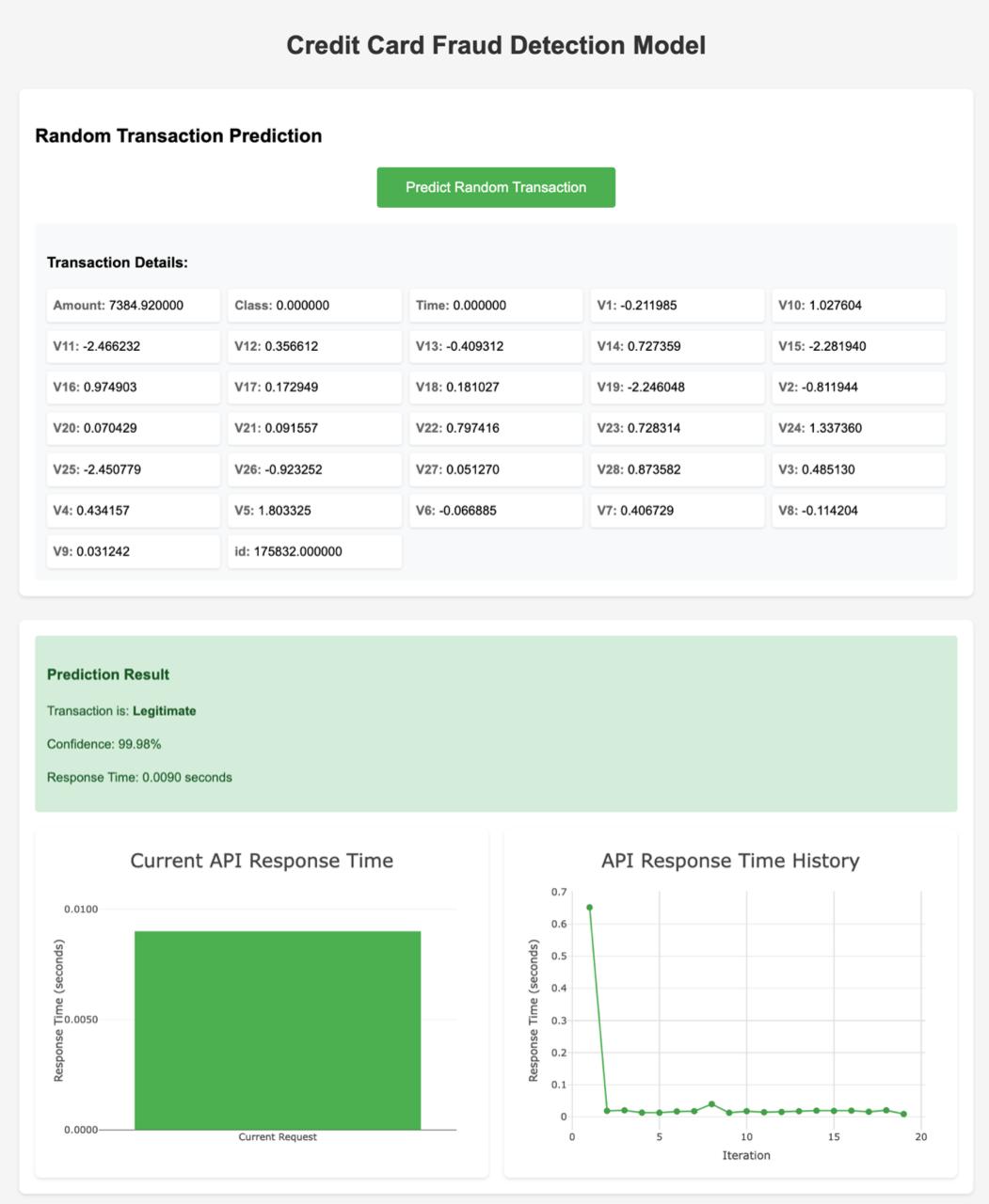


Figure 4: Credit Card Fraud Detection Model Interface

The web interface visualized in **Figure 4** has a "**Predict Random Transaction**" button that triggers the prediction of a randomly selected transaction's legitimacy. It displays detailed transaction information, including various features and the prediction result, which indicates whether the transaction is **"Legitimate" or "Fraudulent"** along with a confidence percentage. Additionally, the interface includes graphs illustrating the current API response time and historical response times, providing insights into the model's performance.

* The **client (client.py)** sends API requests and processes responses.

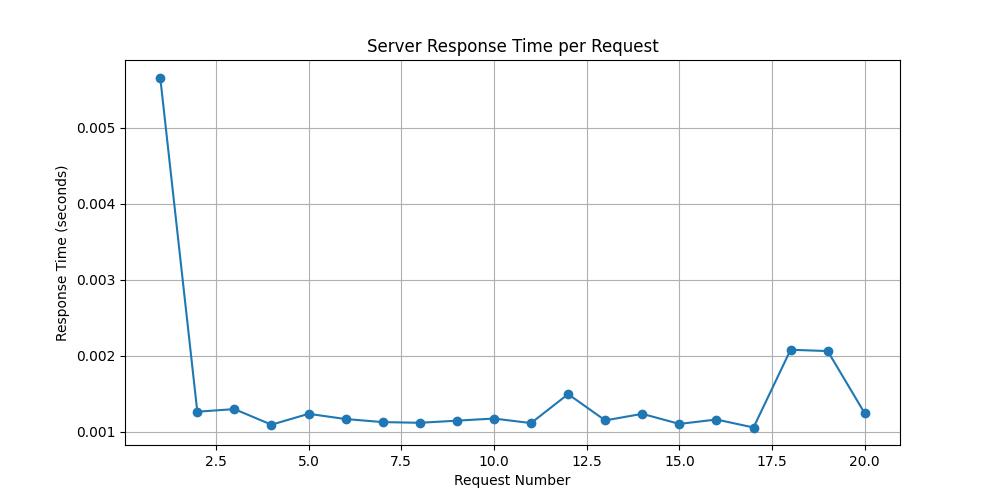


Figure 5: Server Response Time Per Request

**Figure 5** presents a line graph illustrating the server response time for each request made to the Credit Card Fraud Detection Model API. The X-axis represents the request number, while the Y-axis indicates the response time in seconds. The graph shows a significant drop in response time after the first request, stabilizing around 0.001 seconds for subsequent requests, indicating efficient performance of the server. This visualization is crucial for monitoring the API's responsiveness and ensuring optimal user experience, reflecting the underlying code's effectiveness in handling multiple requests efficiently.

This clear separation also supports real-time monitoring of the system, enabling prompt identification and resolution of any performance bottlenecks.

**Conclusion**

This lab successfully demonstrated an end-to-end machine learning pipeline powered by a Flask-based API. Despite early challenges with output display, refining the Front-End UI and upgrading to a LightGBM model resulted in a streamlined and efficient system. The enhanced orchestration process, now managed via run.py instead of separate server executions, further simplified deployment. Future improvements could include advanced response-time optimizations, integration of an interactive dashboard for better model interpretability, and deployment on a scalable cloud platform.

**GitHub Repo URL:** <https://github.com/Krypton0626/ml-pipeline>

**References:**

OpenAI. (2024). *ChatGPT (April 2024 version)*. <https://openai.com>