eta_prediction

This is the **main project folder**. It contains all the components and files required for the ETA Provider Prediction project, including raw data, model artifacts, logs, src, and serving scripts. Below is a breakdown of the subfolders:

---

1. artifact

- This folder contains **pretrained models** and other important files needed for model inference. Each .pkl file represents a serialized object that can be loaded later for serving.
  - gradient_boosting_model.pkl: The gradient boosting model saved after training.
  - random_forest_model.pkl: The random forest model saved after training.
  - Kmeans_model.pkl: The kmeans used for clustering before classification.

---

2. data

- The folder responsible for **storing raw and processed datasets**.
- **Subfolder:**
  - raw: This subfolder stores the initial raw data, in this case, the rides_data.pq file, which is a parquet file containing ride data used for training and evaluation.

---

3. exploratory_data_analysis

- This folder contains any scripts related to **exploratory data analysis (EDA)**. The EDA step is crucial to understanding the structure, distribution, and correlations within the data.

---

4. logs

- This folder stores logs and other tracking information, including **model weights for Neural Network Classifier**.
- **Subfolder:**
  - model_weights/model1: This is where model weights are stored, specifically the model1.h5 file.

---

5. report

- This folder is intended for any **documentation or reporting** on the project. Reports include classification results of 3 different algorithm.

---

6. serving

- This folder is used for **model deployment** or serving the trained model. It contains all the necessary files to make the model accessible via API.
- **Files:**
  - api.py: The script defining the API endpoint (using FastAPI) to serve the model for predictions.
  - model_service.py: A service or class that handles loading the model and running inference on incoming requests.
  - preprocessing.py: This file is used to preprocess the incoming data to ensure it's in the same format as the training data before passing it to the model.
- **Subfolder:**
  - tests: Contains the test files to ensure that the API and preprocessing scripts are functioning correctly.
    - test_api.py: This is likely a test script for ensuring the API works as expected.

---

7. src

This folder represents the **source code** for the project. It houses all the code related to data processing, model training, and pipeline orchestration. This modular approach allows for clear separation between various components of the project, ensuring that each responsibility is isolated within its own file.

✓ **data**

- This folder contains all the **data preprocessing scripts**.
- **Files:**
  - __init__.py: This file is typically empty but makes the folder a Python package, allowing you to import from it.
  - data_processor.py: This file will contain all the logic related to **data preprocessing**. It includes functions to clean, process, and transform raw data into a format suitable for model training, including handling missing values, encoding, normalization, and feature engineering.

✓ **models**

- This folder contains all the **model training scripts**. Each model type is separated into its own file, adhering to the **Single Responsibility Principle** from SOLID design principles.
- **Files:**
  - __init__.py: Makes this folder a package.
  - gradient_boosting_trainer.py: This script handles **training of Gradient Boosting** models. It contains the model initialization, training process, and saving of the trained model.
  - model_trainer.py: This file is the **base class or abstract class** that provides a common interface for all model trainers. It defines shared methods and attributes that each model-specific trainer can inherit from.
  - neural_network_trainer.py: This file handles the **training of the neural network** (e.g., using TensorFlow). It contains the model architecture, hyperparameters, and training process for neural networks.
  - random_forest_trainer.py: This file is responsible for **training a Random Forest model**. It contains the model definition, training procedure, and any relevant hyperparameters.
  - save_training_weights.py: This file manages **saving model weights** during training. It handles checkpointing of models, useful in the case of neural networks.
  - xgboost_trainer.py: This script handles **training of XGBoost model**. It contains the initialization of the model, training routine, and saving process.

✓ **pipeline**

- This folder contains scripts responsible for **orchestrating the entire process** from data loading, model training, and evaluation. It centralizes all the steps required to complete the training process.
- **Files:**
  - __init__.py: A file to make this folder a Python package.
  - train_pipeline.py: This script is responsible for defining the **end-to-end pipeline** for training models. It likely combines steps like data preprocessing, model training, and evaluation. This ensures that you can run all steps sequentially without needing to call each part individually.
  - main.py: This is likely the **entry point** of the project. Running this file will execute the entire pipeline, kicking off all necessary steps for preprocessing, training, and saving models. This is the main file you run to trigger the workflow.

Additional Considerations:

- **Consistency**: The folder structure is modular and consistent, which makes it easier to maintain and navigate. Each component (e.g., data, models, logs, and serving) has its own folder, ensuring separation of concerns.
- **Scalability**: This setup is scalable for future enhancements, like adding more models or expanding the API for additional functionalities.

This structured approach ensures a clean separation of different stages of the machine learning pipeline—data collection, model training, logging, evaluation, and serving—while adhering to clean code and design principles.

Problem Definition

The project requirement was to select the most accurate ETA provider from a set of options. To address this, we formulated the task as a **classification problem**. For each ride, we calculated the error between the actual arrival time (ATA) and the ETA provided by each provider. A new column, accurate_provider, was introduced to indicate the provider with the least error, serving as the target class for the classification task.

Data Preprocessing

1. **Handling Missing Values**:
    - Missing values (NaNs) in the ETA columns for providers were handled by replacing them with a large value, ensuring no rows were dropped and the model would treat these cases appropriately.
2. **Feature Engineering**:
    - **Datetime Features**: From the accept_event_timestamp, we extracted the following features:
        - accept_hour
        - accept_day_of_week
        - accept_month
        - is_weekend (a binary feature indicating if the ride occurred on a weekend).
    - **Error Calculation**: For each provider (A, B, C, D), we calculated the absolute error between the ETA and the ATA, transforming the problem into a classification task where the provider with the lowest error is the target.
    - **Clustering**: To enhance feature representation, we applied **KMeans clustering** on the data, adding a cluster feature to indicate the predicted cluster for each ride.

- **Scaling**: We applied **StandardScaler** to transform the data, ensuring consistent feature scaling for better model performance.

Model Training

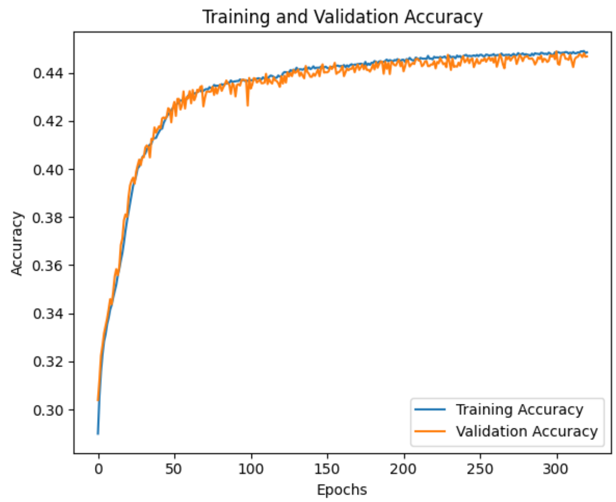
We implemented and trained three different algorithms:

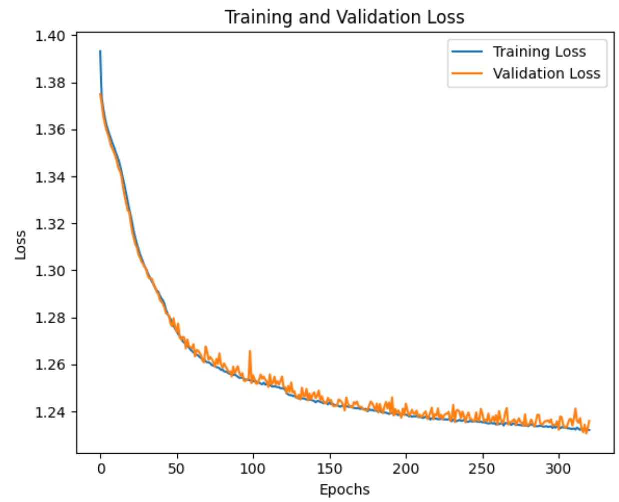
1. **Neural Network**:
   - Architecture:
     - Two dense hidden layers with 20 and 40 units, respectively, using **ReLU activation**.
     - Dense output layer with **softmax activation** for multi-class classification.
   - Optimized for classification by minimizing **sparse_categorical_cross-entropy** loss.
2. **Gradient Boosting**:
   - Parameters:
     - n_estimators = 200
     - max_depth = 6
     - random_state = 42
3. **Random Forest**:
   - Parameters:
     - n_estimators = 200
     - max_depth = 6
     - random_state = 42
4. **KNN:**
   - Parameters:
     - n_neighbors = 10

Results

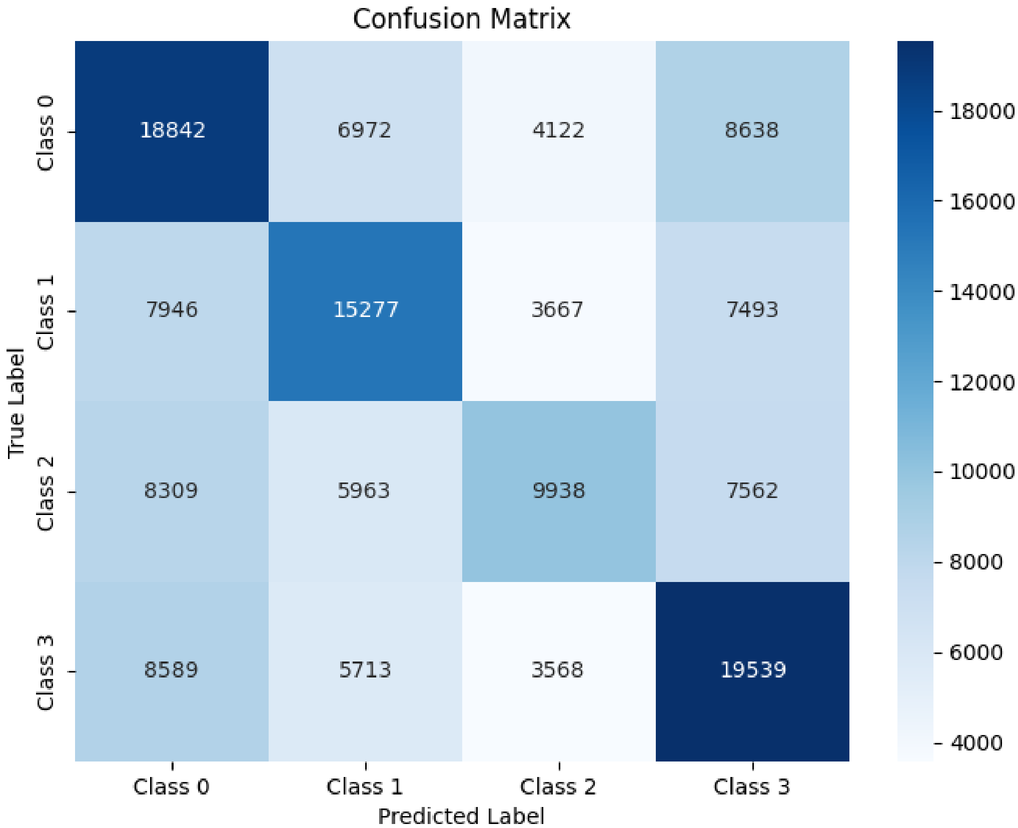
The performance of each model was evaluated based on the precision, recall, and F1-score for each class (provider A, B, C, D). Below are the key results of the trained models:

✓ ***Neural Network Results:***





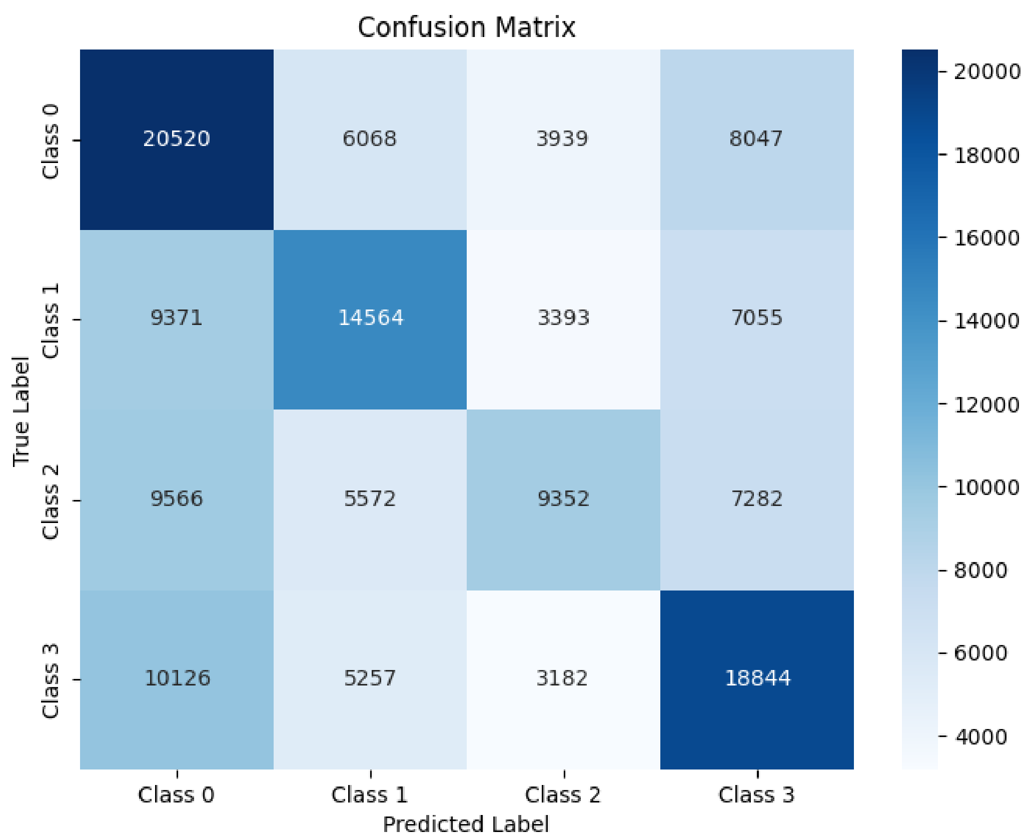
Class	*Precision*	*Recall*	*F1-score*
Class 0 *(provider_A)*	0.43	0.49	0.46
Class 1 *(provider_B)*	0.45	0.44	0.45
Class 2 *(provider_C)*	0.47	0.31	0.37
Class 3 *(provider_D)*	0.45	0.52	0.48



Confusion Matrix
Class 0
Class 1
Class 2
Class 3
True Label
18842
6972
4122
8638
7946
15277
3667
7493
8309
5963
9938
7562
8589
5713
3568
19539
Class 0
Class 1
Class 2
Class 3
Predicted Label
18000
16000
14000
12000
10000
8000
6000
4000

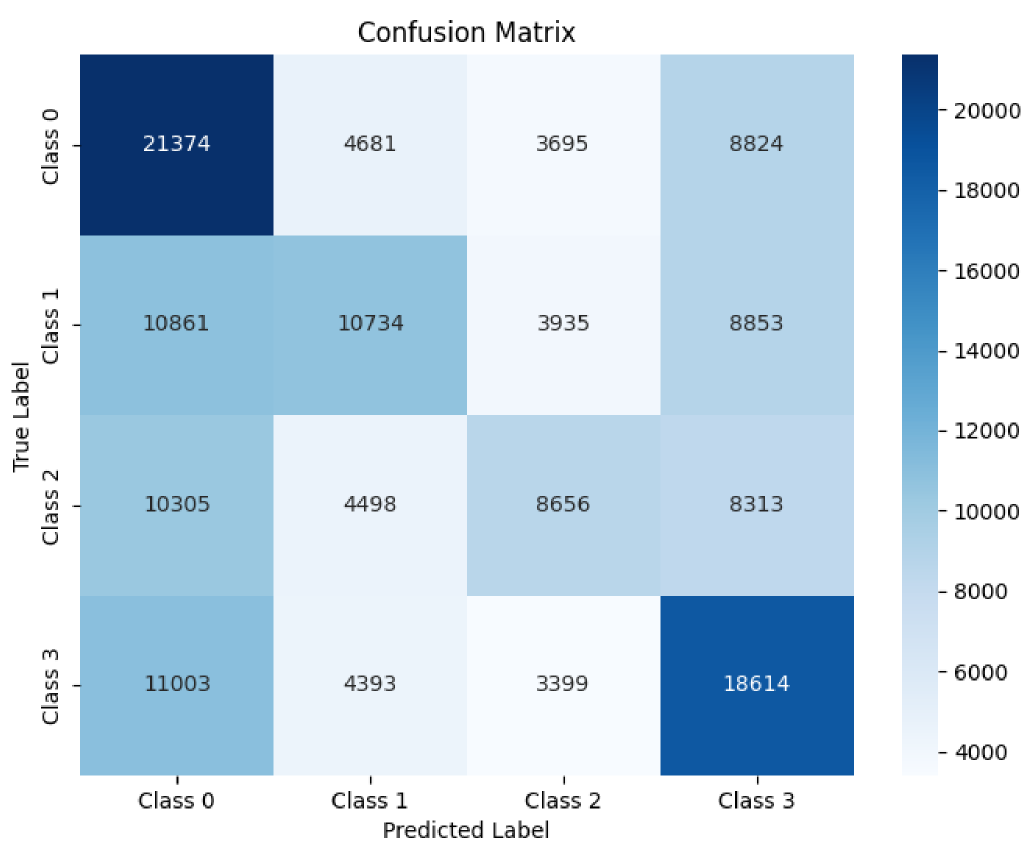
✓ ***Random Forests:***

Class	*Precision*	*Recall*	*F1-score*
Class 0 *(provider_A)*	0.41	0.53	0.47
Class 1 *(provider_B)*	0.46	0.42	0.44
Class 2 *(provider_C)*	0.47	0.29	0.36
Class 3 *(provider_D)*	0.46	0.50	0.48



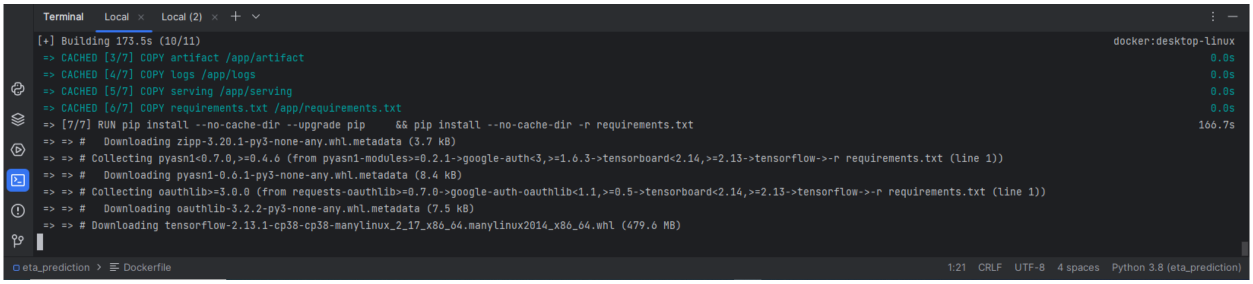
✓ ***Gradient Boosting:***

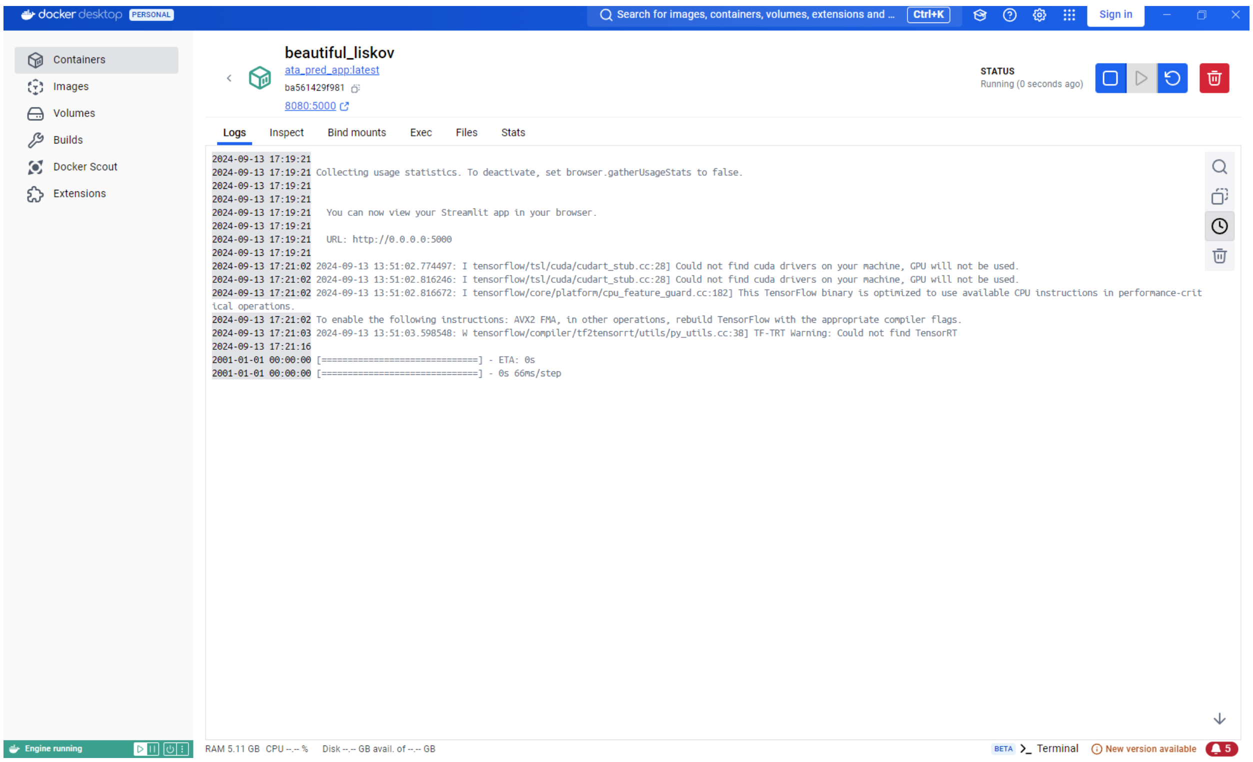
Class	*Precision*	*Recall*	*F1-score*
Class 0 *(provider_A)*	0.40	0.55	0.46
Class 1 *(provider_B)*	0.44	0.31	0.37
Class 2 *(provider_C)*	0.44	0.27	0.34
Class 3 *(provider_D)*	0.42	0.50	0.45



Serving the Models

✓ *Dockerization:*





✓ ***Streamlit UI:***