CPE393 SPECIAL TOPIC III :

MACHINE LEARNING OPERATIONS

Final Report

Gunn Wangwichit	65070503406
Parit Leelasetawong	65070503422
Suchanat Ratanarueangrong	65070503440
Phakhapol Maneesopa	65070503459
Phuri Yamuthai	65070503461

Submitted to

Dr. Aye Hninn Khine & Dr. Santitham Prom-on

SEMESTER 2/2024 Department of Computer Engineering

King Mongkut’s University of Technology Thonburi

Table of Contents

Abstract

In an era of clean energy and electric vehicles playing a key role in sustainable development, the ability to accurately forecast the distance of electric vehicles is essential for consumers, manufacturers, and governments. This project aims to develop a machine learning model to forecast the maximum distance of electric vehicles, based on vehicle specifications such as make, model, year of manufacture, battery size, and drive system, through the application of a comprehensive Machine Learning Operations Process.

The dataset used is from Kaggle (Electric Vehicle Population Data), which collects data on electric vehicles registered in Washington State, USA, covering both technical characteristics and actual use. This project has developed models in both regression for form input and classification from vehicle photos to display average distance from real databases.

In terms of system architecture, a structure has been designed to support real-time data transmission via Apache Kafka, using AWS RDS as the main database and using Apache Airflow to manage workflows for automatic model training and running. Models and workflows are tracked and managed via MLflow, which enables efficient version storage and deployment of models. The entire system is managed via Docker Containers for flexible deployment and easy scalability in the future.

The user interface is developed with Streamlit, supports both data entry and image upload, and is connected to the backend system via Ngrok for external access. Users can immediately see the prediction results on a user-friendly screen, in the form of text, tables, and graphs. The project results demonstrate the feasibility of applying the MLOps approach to real-world problems that require accuracy, speed, and scalability to support informed decision-making for future EV users and manufacturers.

Introduction

The global shift towards sustainable transportation has positioned electric vehicles (EVs) at the forefront of automotive innovation. As EV adoption accelerates, accurately predicting their driving range becomes paramount for manufacturers, policymakers, and consumers alike. Driving range, influenced by factors such as battery capacity, drivetrain configuration, and efficiency metrics, directly impacts consumer confidence and the broader acceptance of electric mobility. This project endeavors to optimize the prediction of EV driving range by leveraging machine learning techniques. By analyzing key vehicle specifications, we aim to develop a predictive model that estimates the maximum driving distance of EVs with high accuracy. Such a model addresses range anxiety among potential EV users and provides valuable insights for manufacturers in designing and developing future electric vehicles.

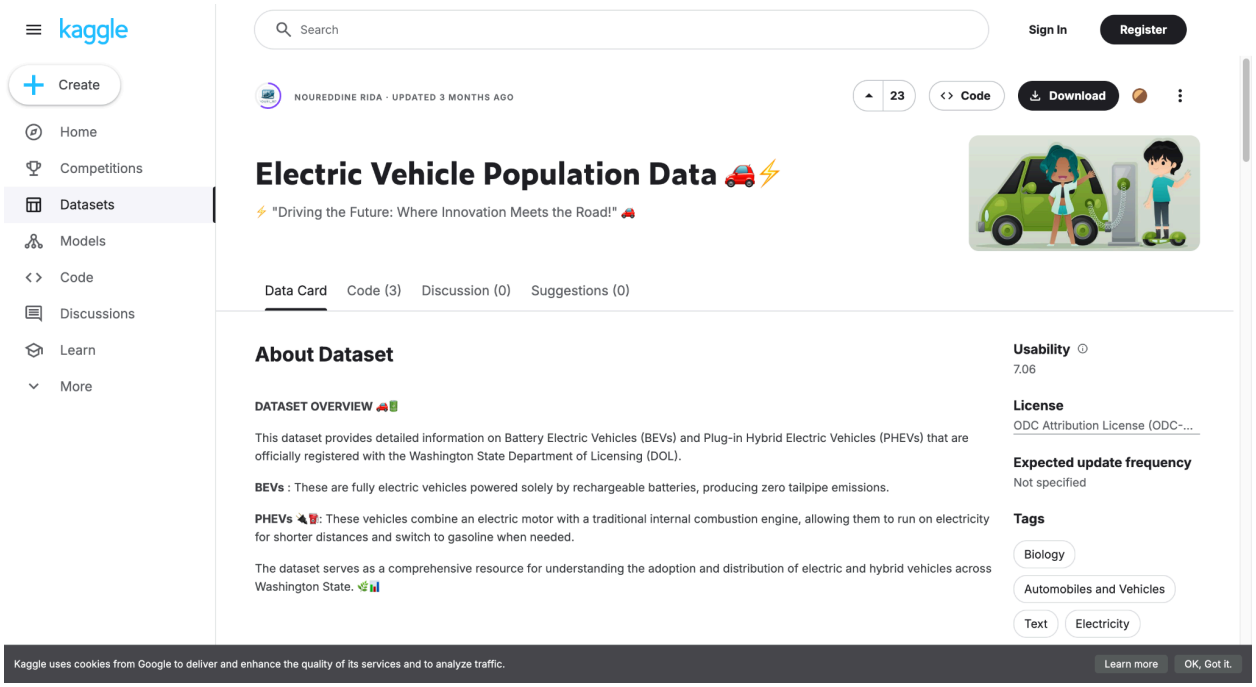
Our analysis utilizes the Electric Vehicle Population dataset from Kaggle, which encompasses detailed information on registered EVs, including make, model, year, battery capacity, and geographical distribution. This dataset serves as a robust foundation for our predictive modeling efforts, allowing us to capture a wide spectrum of vehicle characteristics and usage patterns.

To facilitate a comprehensive and scalable analysis, we have implemented a data pipeline using Apache Kafka for real-time data streaming and Apache Airflow for workflow orchestration. Our machine learning models are developed and tracked using MLflow, ensuring reproducibility and efficient model management. The entire system is containerized using Docker, promoting ease of deployment and scalability.

By integrating advanced machine learning methodologies with a robust data engineering framework, this project aims to contribute to the broader efforts of promoting sustainable transportation through accurate and reliable EV range predictions.

Our Group Github: https://github.com/nutkung1/FinalProjectCPE393

Dataset



In this project, our group used a dataset from the Kaggle website.

🔗 **https://www.kaggle.com/datasets/noureddineridanr96/electric-vehicle-population-data**

This dataset is called Electric Vehicle Population Data, which collects details of electric vehicles registered in Washington State, USA, and is used in research and analysis related to the development of clean energy vehicle technology.

We have used this dataset to create a machine learning model in both the fields of predicting electric vehicle mileage (Regression) and classifying vehicles from photographs (Classification) to simulate real-world usage scenarios of consumers and vehicle manufacturers in the future.

This dataset is suitable for training the model, with a large amount of data and a variety of vehicle types, models, years of manufacture, and usage characteristics in each area, which allows the model to learn and develop efficiently.

For model development, work systems, and management using the MLOps approach, our group has collected them in the GitHub Repository, which can be accessed at:

🔗**https://github.com/nutkung1/FinalProjectCPE393.git**

Objective

1. Problem Statement

This project aims to predict the maximum electric vehicle (EV) driving range based on key factors such as battery capacity, drivetrain type, and other vehicle specifications using machine learning. By analyzing real-world EV population data, the goal is to build a model that can provide accurate range predictions, helping users and developers understand vehicle performance more effectively.

2. Motivation and Impact

As electric vehicles become more common, knowing how far a vehicle can travel on a single charge is critical for users, manufacturers, and policymakers. Accurate range prediction helps reduce range anxiety, supports infrastructure planning (e.g., charging stations), and can guide consumers in choosing suitable EV models. This project also contributes to promoting sustainable transportation and accelerating the transition to cleaner energy solutions.

3. Stakeholders

Key stakeholders include:

- EV manufacturers, who can use insights to improve vehicle designs
- Consumers, who benefit from better information for purchase decisions
- Policymakers and city planners, who need data to support EV adoption strategies
- Researchers and developers, working on EV technology and predictive systems

4. Scope and Constraint

Scope

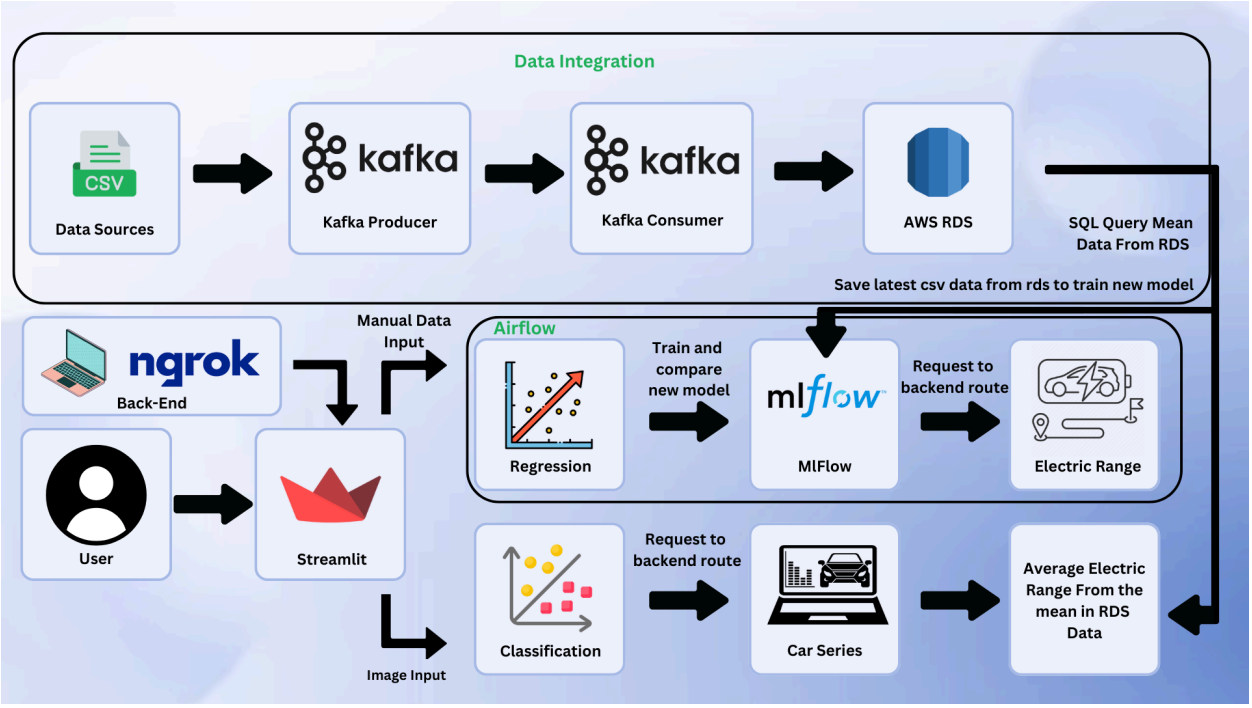
The project uses the Electric Vehicle Population dataset from Kaggle, which includes features like vehicle make, model, year, battery capacity, drivetrain type, electric range, and location data. The focus is on supervised learning techniques to predict electric range.

Constraints

The dataset is limited to registered vehicles and may not reflect all current models or technologies. Some missing or inconsistent data required preprocessing. Additionally, real-world range can vary based on driving behavior and conditions, which are not captured in this dataset.

Methodology

1. System Overview



The system is designed to predict and analyze electric vehicle (EV) ranges using both image-based classification and manual data input regression, while integrating real-time data ingestion through Kafka and AWS RDS. The architecture can be divided into two main sections:

1. Data Integration Layer (Top Section)

This layer is responsible for real-time ingestion and storage of electric vehicle data.

Components & Workflow:

- **CSV Data Sources**:
  Raw EV data is collected from external structured sources (e.g., datasets containing vehicle attributes and electric range information).

- **Kafka Producer**:
  Reads the CSV files and streams data into a Kafka topic.

- **Kafka Consumer**:
  Listens to the Kafka topic and receives real-time data, which is then forwarded to the database.

- **AWS RDS**:
  Acts as the main storage for vehicle data. It stores attributes such as make, model, year, and electric range, and is used for querying statistics.

- **SQL Query**:
  SQL commands are used to extract statistical data (e.g., mean electric range) from AWS RDS for use in the classification module.

2. Application & User Interface Layer (Bottom Section)

This layer includes the Streamlit-based front-end and backend services that handle user interactions, predictions, and visualizations.

User Interaction:

1. **Streamlit Interface**:
   Acts as the main GUI for users to interact with the system via:

   a. **Image Input** for Classification

   b. **Manual Input Forms** for regression

2. **Ngrok (Back-End Exposure)**:
   Exposes the local back-end server to the internet securely so the Streamlit UI can access classification and regression routes.

Our Website: https://finalprojectcpe393.streamlit.app/

Regression Pipeline: Manual Input

1. **User Inputs Vehicle Details**:
   The user manually enters details like make, model, year, battery size, etc.
2. **Getting the latest data from RDS**:
   The backend automatically gets the latest from AWS RDS to train a new model.
3. **MLFlow model comparison**:
   The MLFlow automatically compares the model and saves the log, also saving the best regression model for future use.
4. **Regression Model Triggered via Airflow**:
   Data is sent through Apache Airflow, which manages and schedules the inference pipeline.
5. **Request to Backend**:
   The backend uses a trained regression model to predict the electric range.
6. **Output Electric Range**:
   The predicted electric range is shown in the UI.

Classification Pipeline: Image-Based Input

1. **User Uploads Image**:
   An image of a car is uploaded through the Streamlit interface.
2. **Classification Module**:
   Identifies the car model from the image using a trained machine learning model.
3. **Request to Backend Route**:
   The identified model is sent to the backend.
4. **Car Series Lookup**:
   The backend queries AWS RDS for that car model and retrieves statistical range data.
5. **Display Electric Range Summary**:
   The UI displays **minimum**, **maximum**, and **average** electric range based on existing entries in RDS.

Real-time Data ingestion to AWS RDS

1. Apache Kafka

Apache Kafka is an open-source distributed event streaming platform designed to handle high throughput and real-time data feeds efficiently. It enables ingestion, storage, and processing of continuous data streams, making it suitable for systems that require real-time analytics, monitoring, or messaging.

Key Components of Kafka:

- **Producer:** Sends data (messages) to Kafka topics.
- **Broker:** The server that stores data and manages requests from producers and consumers.
- **Consumer:** Reads data from Kafka topics for processing.
- **Topic:** Logical channels that categorize and organize messages.
- **Zookeeper:** Manages and coordinates Kafka brokers, ensuring cluster stability and synchronization.

Benefits of Kafka:

- Supports real-time data streaming with low latency.
- Easily scalable horizontally to handle large data volumes.
- Highly reliable and fault-tolerant for large-scale distributed systems.

In this project, Kafka is used as a proof of concept (PoC) to simulate real-time data flow. Kafka and Zookeeper services were deployed using Docker containers to simplify setup and ensure environment consistency. The producer and consumer applications also ran inside Docker containers for seamless integration and testing.

- The Producer simulates streaming electric vehicle (EV) data such as vehicle model, battery capacity, and range by sending JSON messages to a Kafka topic.
- The Consumer subscribes to the Kafka topic, consumes messages in real-time, and forwards the data for storage into PostgreSQL on AWS RDS.

Kafka Producer (Core Part) :

```python
def connect_kafka():
    max_retries = 20
    for attempt in range(max_retries):
        try:
            producer = KafkaProducer(
                bootstrap_servers=bootstrap_servers,
                value_serializer=lambda v: json.dumps(v).encode('utf-8')
            )
            return producer
        except NoBrokersAvailable:
            print(f"Retry {attempt + 1}/{max_retries}: Kafka broker not available, retrying in 10
seconds...")
            time.sleep(10)
    raise Exception("Failed to connect to Kafka after retries")

# Connect to Kafka
producer = connect_kafka()

# Read CSV file
df_train = pd.read_csv('train20.csv')
# Function to send data to Kafka
def send_to_kafka(df, topic):
    for _, row in df.iterrows():
        data = row.to_dict()
        producer.send(topic, value=data)
        print(f"Sent to {topic}: {data}")
        time.sleep(1)  # Simulate delay

# Send train and test data
send_to_kafka(df_train, 'simulate_data_train')

# Close producer
producer.flush()
producer.close()
```

The Kafka Producer reads EV data from a CSV file and sends each row to the simulate_data_train topic on Kafka. It uses JSON to serialize the data and includes a retry mechanism to ensure connection to the Kafka broker. A delay is added between messages to simulate real-time streaming. This setup acts as a proof of concept to mimic live data ingestion for testing and development.

Kafka Consumer (Core Part) :

```python
# Kafka Consumer
consumer = KafkaConsumer(
    'simulate_data_train',
    bootstrap_servers=bootstrap_servers,
    group_id='ev_consumer_group',
    value_deserializer=lambda m: json.loads(m.decode('utf-8')),
    auto_offset_reset='earliest',
    enable_auto_commit=False,
    max_poll_records=100
)

# Consume messages and insert into the appropriate table
for msg in consumer:
    data = msg.value
    print(f"Received from topic {msg.topic}: {data}")
    if msg.topic == 'simulate_data_train':
        create_table('simulate_data_train')
        insert_ev_data(data, 'simulate_data_train')
```

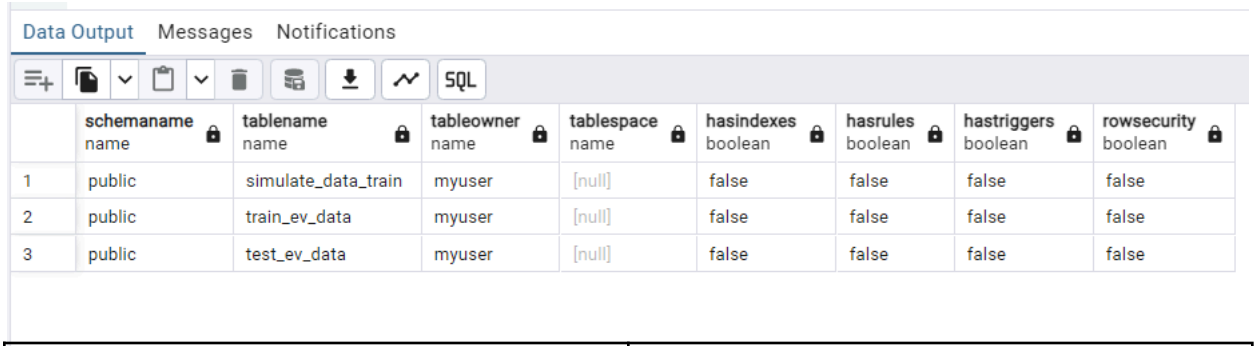
The Kafka Consumer listens to the simulate_data_train topic and receives EV data in real time. Each message is deserialized from JSON and inserted into a PostgreSQL database on AWS RDS. It uses the psycopg2 library to connect to the database. This process enables continuous data ingestion into a relational database, forming a core part of the real-time data pipeline.

2. AWS RDS

Amazon Relational Database Service (RDS) is a fully managed cloud-based relational database that automates administrative tasks such as maintenance, backups, and scaling. In this project, we use PostgreSQL on AWS RDS to store real-time electric vehicle (EV) data streamed from Apache Kafka. RDS serves as the final destination for ingested data, providing persistent storage for downstream applications.

Table Overview:

We designed three main tables, each serving a specific purpose:


Data Output Messages Notifications

SQL

	schemaname name	tablename name	tableowner name	tablespace name	hasindexes boolean	hasrules boolean	hastriggers boolean	rowsecurity boolean
1	public	simulate_data_train	myuser	[null]	false	false	false	false
2	public	train_ev_data	myuser	[null]	false	false	false	false
3	public	test_ev_data	myuser	[null]	false	false	false	false

Table Name	Purpose
simulate_data_train	Receives real-time data streamed from Kafka
train_ev_data	Used by the ML team for training models
test_ev_data	Used by the ML team for testing models

The simulate_data_train table acts as a raw data buffer, while the train_ev_data and test_ev_data tables are designed for machine learning workflows, including model training and evaluation.

Data Flow and Ingestion Process:

The data ingestion process starts with EV telemetry data produced and streamed to Kafka topics by the Kafka Producer. The Kafka Consumer subscribes to these topics, consumes messages in real time, and inserts them into the simulate_data_train table on PostgreSQL on AWS RDS. This pipeline uses Python's psycopg2 library to connect to the PostgreSQL RDS instance. Each incoming Kafka message is deserialized and transformed into a dictionary that matches the database schema before being inserted into the table. This process enables near real-time persistence of streaming data, allowing downstream applications to access fresh data continuously.

Producer and Consumer Logs:

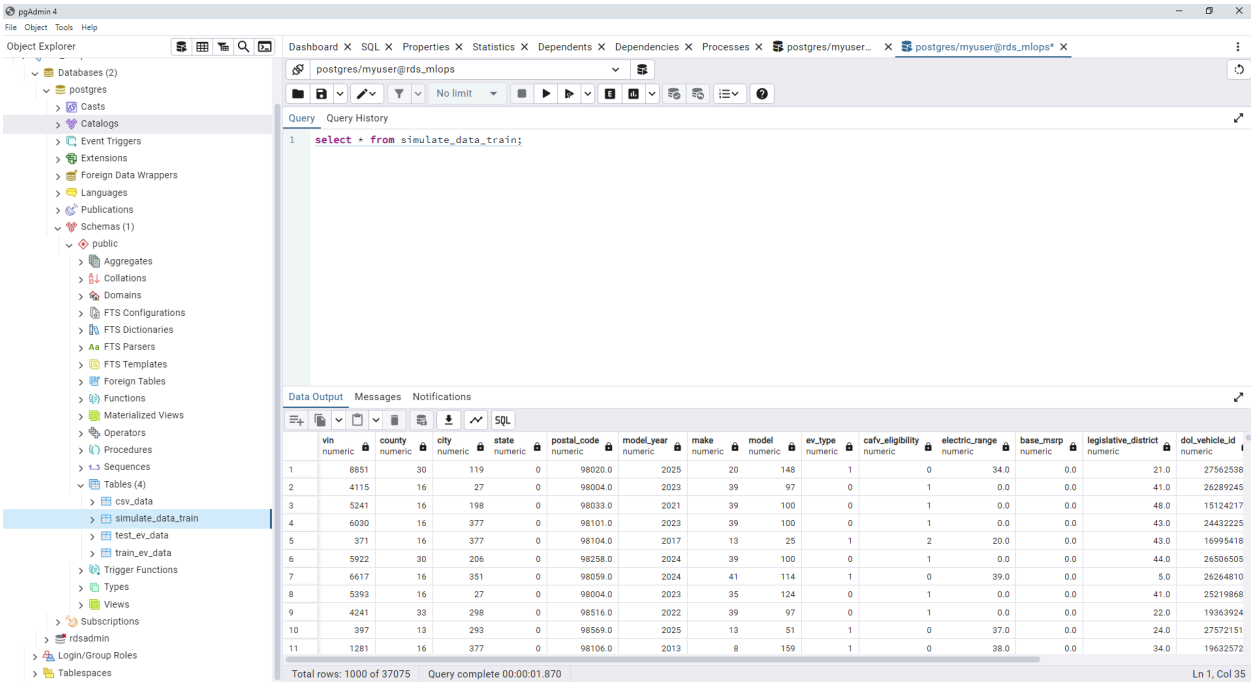
Throughout the streaming process, both the Kafka Producer and Consumer generate logs that provide visibility into the data pipeline's operation:

- **Producer Logs:** These logs show each data record sent to the Kafka topic, including the message content and timestamps. They help confirm that data is correctly serialized and published to Kafka in real time.

- **Consumer Logs:** These logs capture the messages consumed from Kafka topics, including the details of each record being processed and inserted into PostgreSQL on AWS RDS. They verify that the consumer successfully receives messages and writes them to the database.

The logs serve as an essential tool for monitoring and debugging the pipeline. Screenshots of the logs demonstrate the continuous flow of data from the producer through Kafka to the consumer and finally into the RDS tables.

Results :



The real-time data ingestion pipeline developed in this project has shown successful end-to-end functionality. The Kafka Producer continuously streamed data to the Kafka topic, with logs confirming that each message was properly formatted and sent without errors. On the other side, the Kafka Consumer received these messages in real time, as verified by the consumer logs showing successful message deserialization and processing. Once consumed, the data was inserted into the simulate_data_train table. This validates that the pipeline can reliably transfer streaming data into a cloud-hosted relational database. During the testing period, both the producer and consumer operated smoothly without interruptions, showing the stability of this proof-of-concept system. This confirms that real-time data ingestion using Apache Kafka and PostgreSQL on AWS RDS is not only feasible but can provide a strong foundation for scalable analytics and ML workflows.

Sources: https://github.com/nutkung1/FinalProjectCPE393/tree/main/kafka

Exploratory Data Analysis

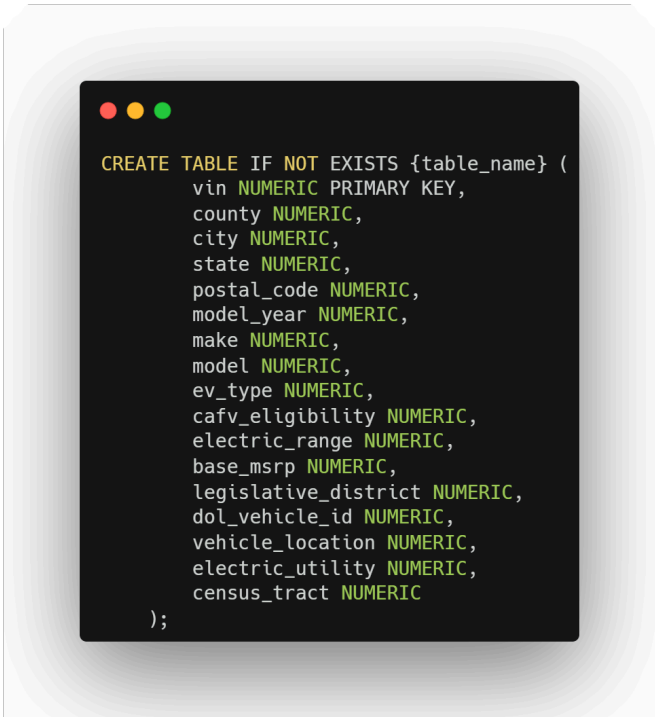
1. Data Cleaning

Data cleaning was performed to improve the quality and consistency of the dataset. The process began with identifying and handling missing values. Columns containing null values were reviewed, and rows with any missing data were removed to ensure that the dataset included only complete and reliable information. In addition to handling missing values, duplicate records were checked and removed to avoid redundancy and ensure that each entry represented a unique observation. A review of data types across all features was also conducted to confirm that each column was appropriately formatted.

By the end of this phase, the dataset had no missing or duplicate values, and all features were properly structured, ensuring that the data was ready for further analysis and model development.

2. Data Wrangling & Transformation

Following the cleaning process, the dataset is transformed to make it suitable for machine learning applications. In particular, all categorical variables such as vehicle make, model, and drivetrain were converted into a numerical format using label encoding. This step is necessary because most machine learning algorithms require numerical input to process and learn from the data. Each unique category in a categorical column was assigned a specific numerical code, enabling the model to interpret these variables as distinct values. After transformation, the dataset was reviewed to ensure the encoding was applied correctly and that no information was lost in the process. These



```
CREATE TABLE IF NOT EXISTS {table_name} (
        vin NUMERIC PRIMARY KEY,
        county NUMERIC,
        city NUMERIC,
        state NUMERIC,
        postal_code NUMERIC,
        model_year NUMERIC,
        make NUMERIC,
        model NUMERIC,
        ev_type NUMERIC,
        cafv_eligibility NUMERIC,
        electric_range NUMERIC,
        base_msrp NUMERIC,
        legislative_district NUMERIC,
        dol_vehicle_id NUMERIC,
        vehicle_location NUMERIC,
        electric_utility NUMERIC,
        census_tract NUMERIC
    );
```

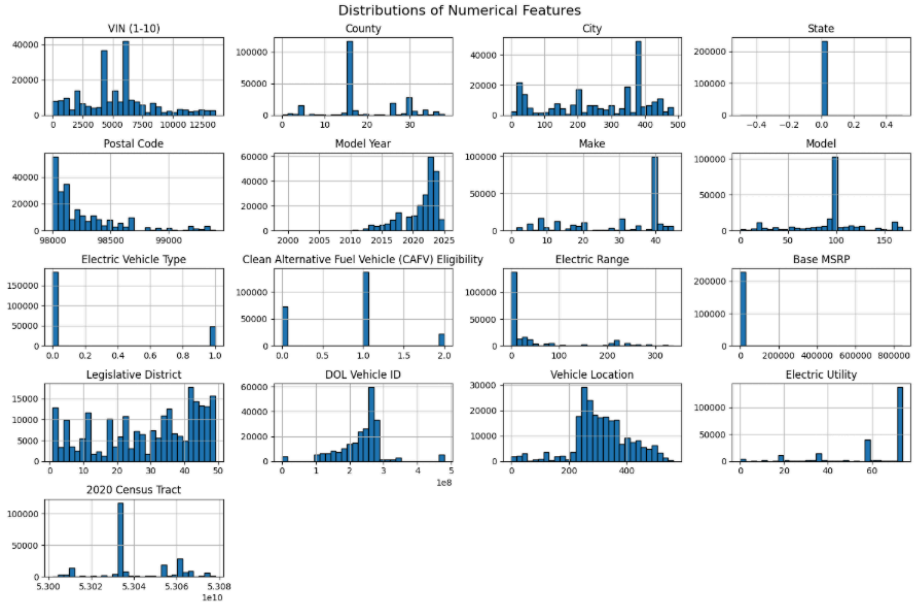
transformations were essential for preparing the dataset for predictive modeling and ensuring that all relevant features could be effectively used in algorithm training.

3. Data Visualization

Each team member contributed to the data visualization process by focusing on different aspects of the dataset. These visualizations helped us explore patterns and trends critical to our predictive modeling task. Below are the most significant visualizations created by each member:

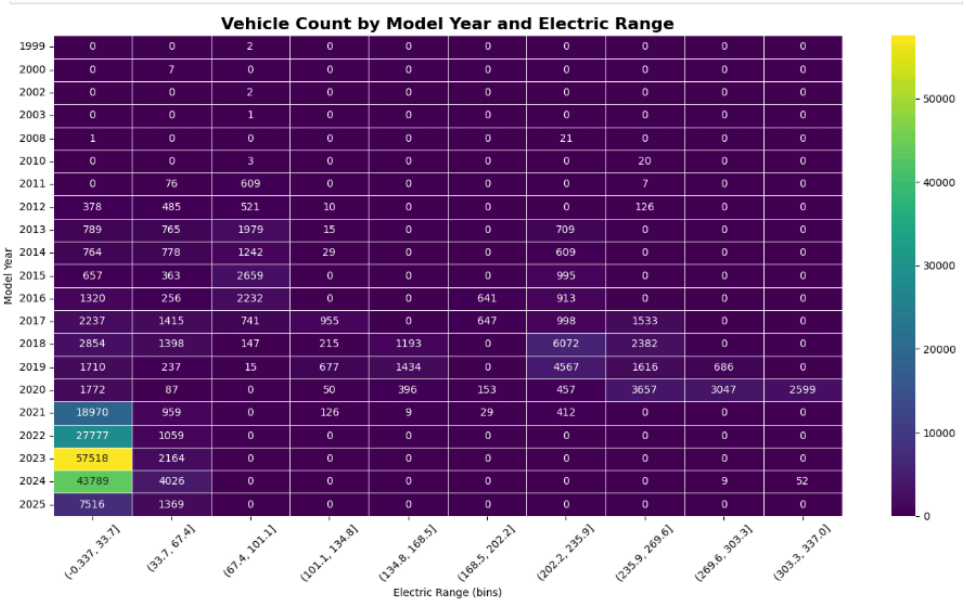
Correlation Heatmap

A heatmap was created to show the correlation between numerical features such as battery capacity, electric range, and model year. This helped identify which variables had strong relationships and informed the selection of features for the machine learning model.



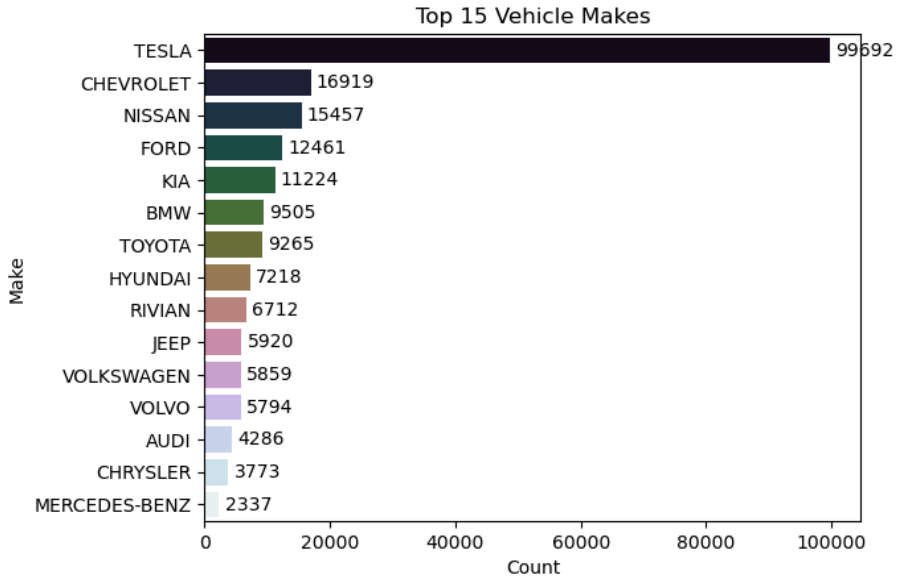
Histogram of Battery Capacity

Explored the distribution of battery capacities among vehicles in the dataset. This helped identify the most common battery sizes and how they vary across models.



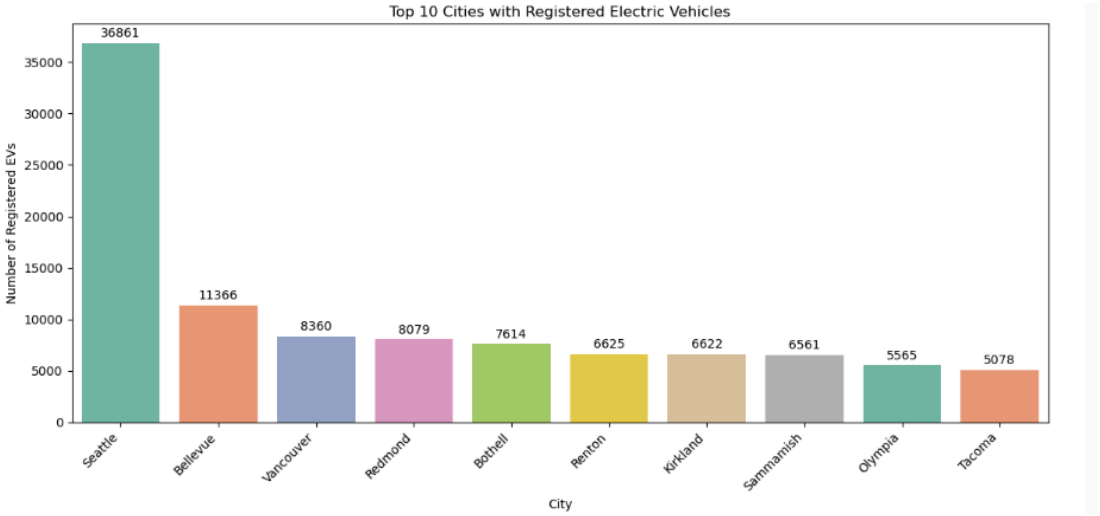
Bar Chart of Vehicle Makes

A bar chart was used to visualize the frequency of different electric vehicle makes. This helped identify the most common manufacturers in the dataset and gave insight into brand distribution.



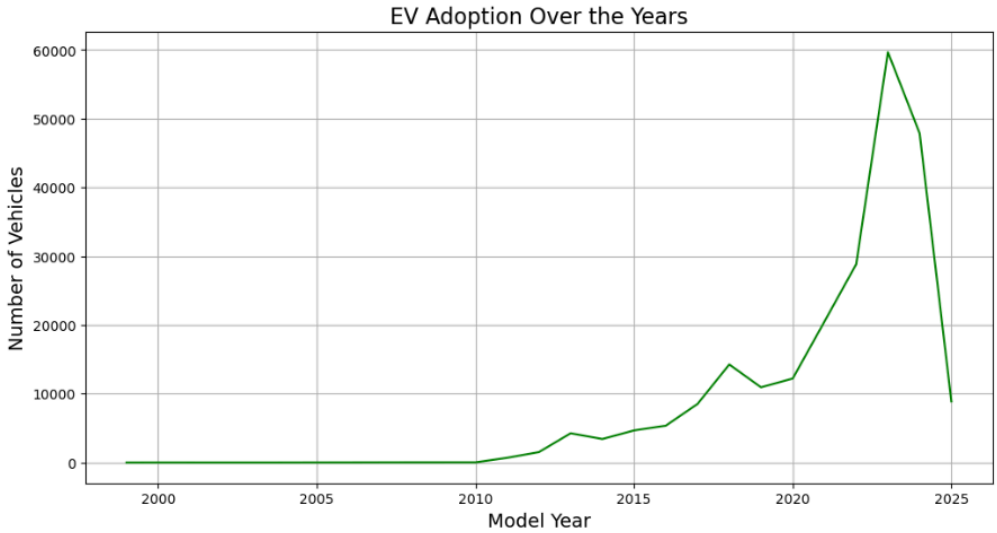
Geographical Distribution Map

A map was developed to display the registration locations of electric vehicles. This visualization showed which regions had higher EV adoption and allowed us to observe spatial trends.



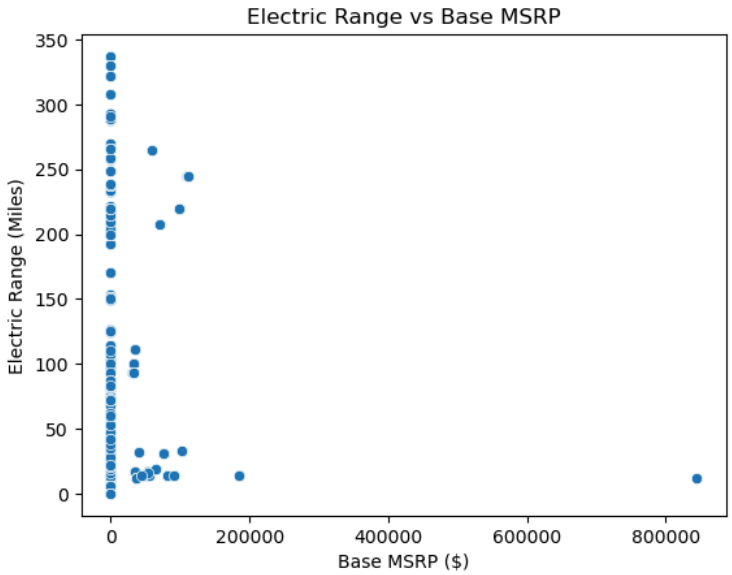
Line Chart of Yearly Registrations

A line chart was used to show how the number of electric vehicle registrations changed over time. This highlighted growth trends and helped connect the data to broader developments in technology and policy.



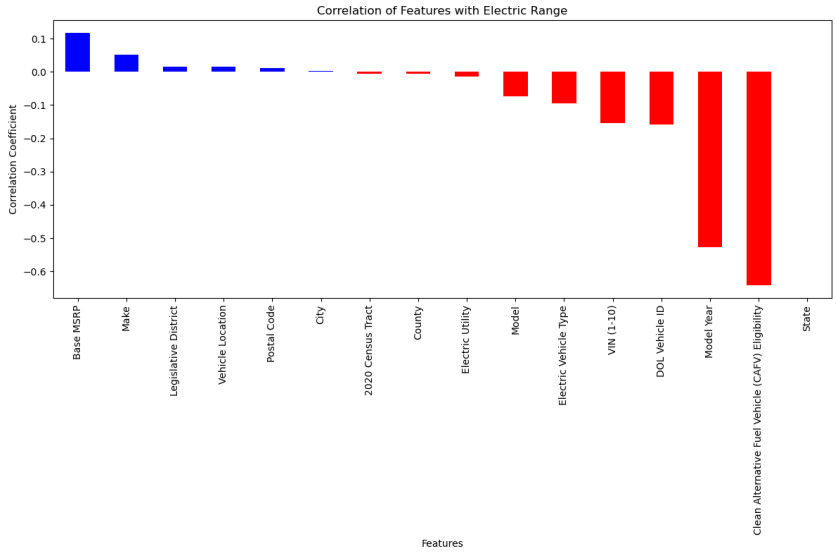
Boxplot of Electric Range by Model

A boxplot compared the electric range of different vehicle models. This was important for understanding how range varies depending on vehicle type and was directly related to our prediction goal.



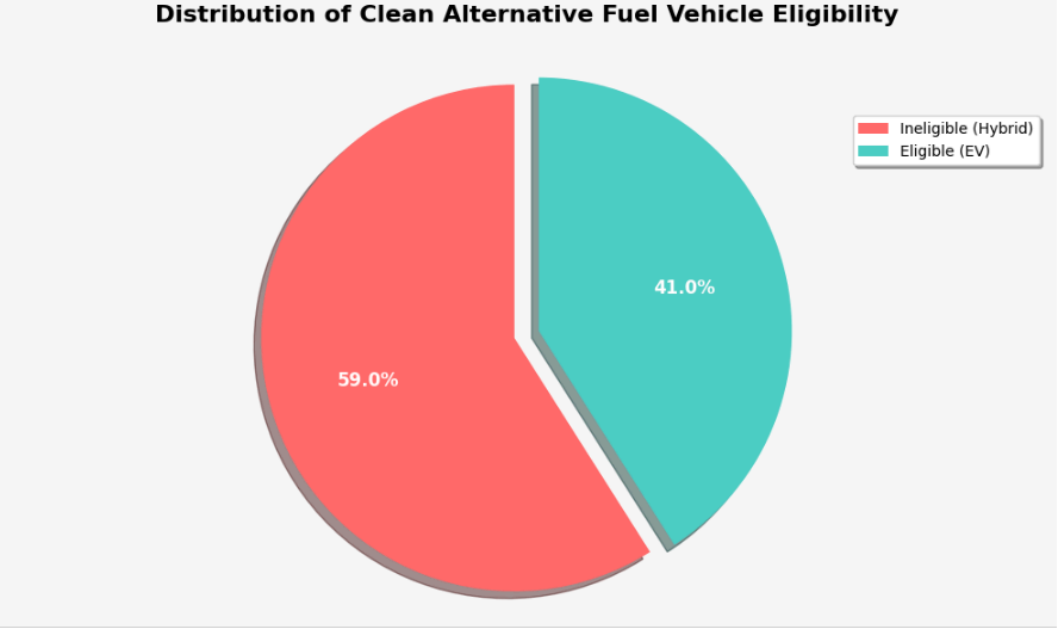
Feature Correlation with Electric Range

This bar chart shows how different features are correlated with electric range. Some features, like model year and CAFV eligibility, have a positive impact, while others like vehicle type and weight, show a negative correlation.



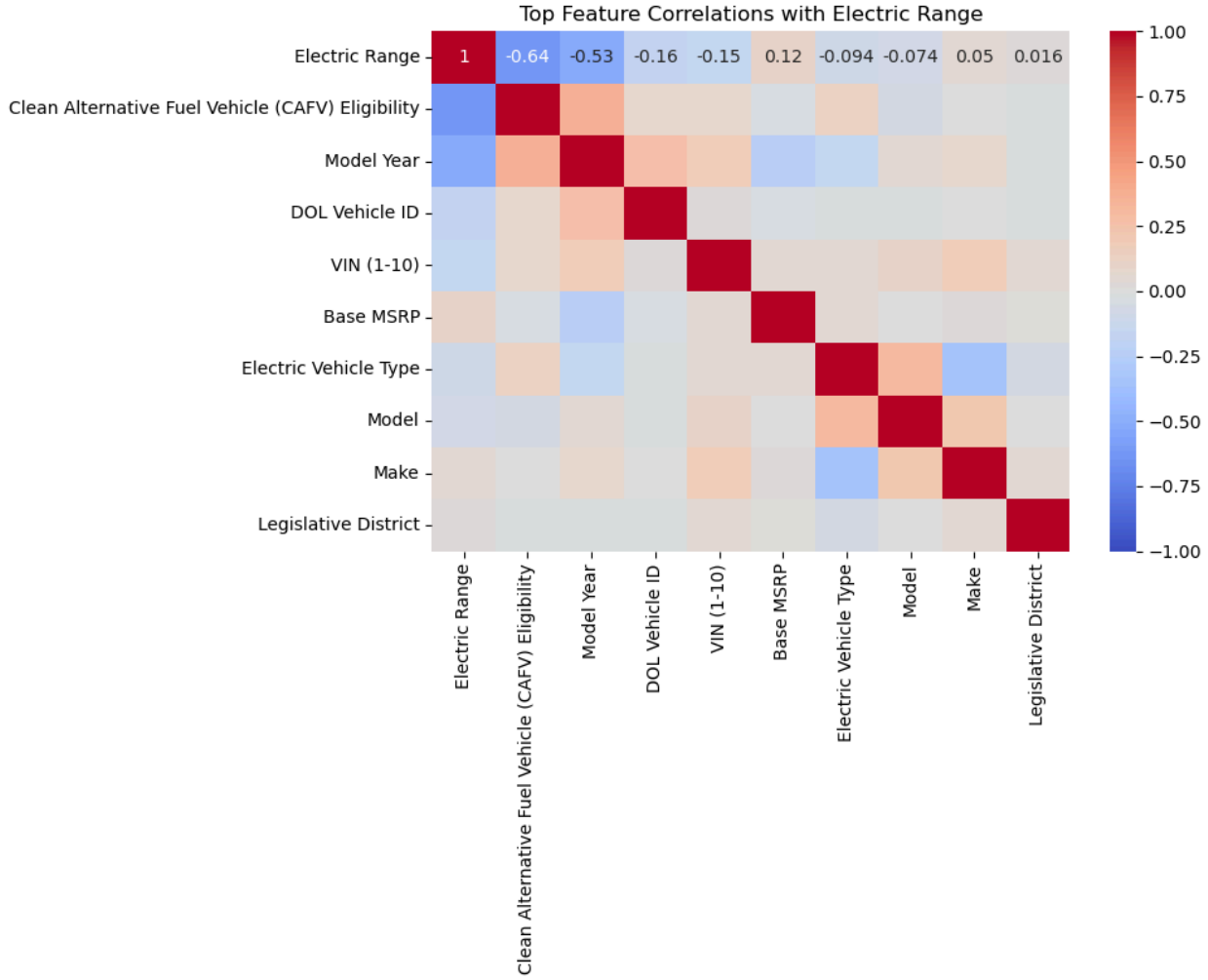
CAFV Eligibility Distribution

The pie chart displays the proportion of vehicles eligible for the Clean Alternative Fuel Vehicle (CAFV) program. Around 41% are eligible, and 59% are not, giving insight into environmental compliance.



Feature Correlation Heatmap

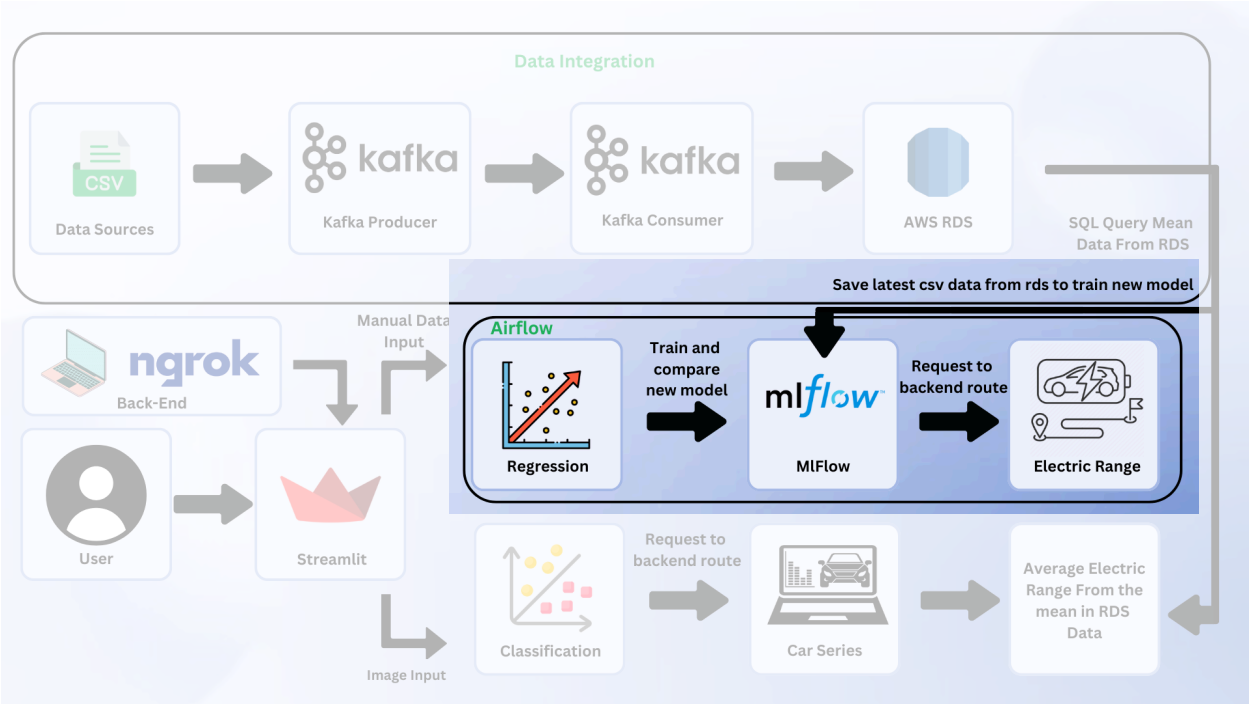
The heatmap highlights the relationships between key features. Electric range is most correlated with CAFV eligibility and model year, supporting the findings from the bar chart.



Sources: https://github.com/nutkung1/FinalProjectCPE393/tree/main/notebooks

Machine Learning Pipeline

1. Apache Airflow and MLflow



In this system, Airflow and MLflow are used together to automate the retraining of the electric range prediction model based on the latest vehicle data streamed through Kafka and stored in AWS RDS.

1. Automated Retraining Workflow

1. Scheduled Trigger via Airflow (Every 50 Minutes)

- Airflow is configured with a DAG (Directed Acyclic Graph) that runs every 50 minutes.
- This DAG initiates the retraining process by:
    - Querying AWS RDS to extract the most recent vehicle data.
    - Saving the data as a new CSV file used as training input.

2. Data Retrieval and Model Training

- The extracted dataset is passed into a regression pipeline, which:
  - Cleans and processes the data.
  - Trains a new model using the latest input features (e.g., vehicle make, model, year, battery specs).

3. Model Tracking and Comparison via MLflow

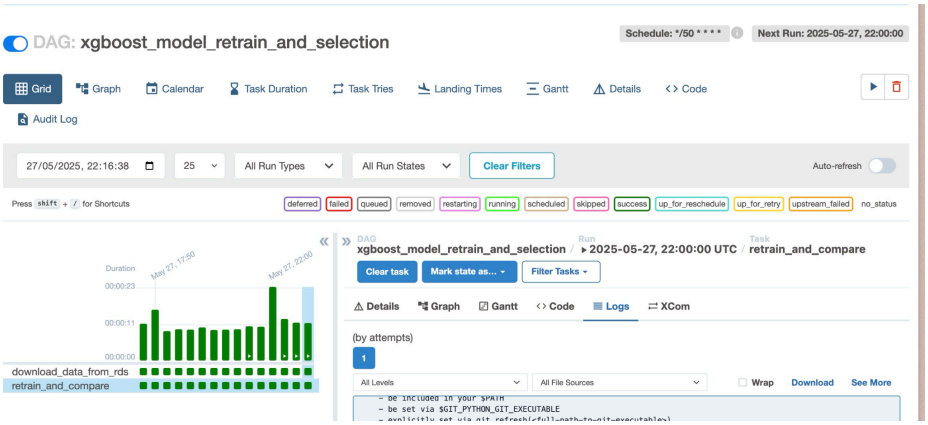
- The newly trained model is logged and versioned using MLflow, which tracks:
  - Parameters
  - Metrics
  - Artifacts
- MLflow automatically compares the new model's performance with the existing one.
- If the new model performs better, it is promoted for inference.

4. Updated Model Used for Predictions

- Once approved, the latest model is exposed through the backend route.
- When users submit vehicle information through the Streamlit UI, the new model is used to predict the electric range accurately based on current trends and data.

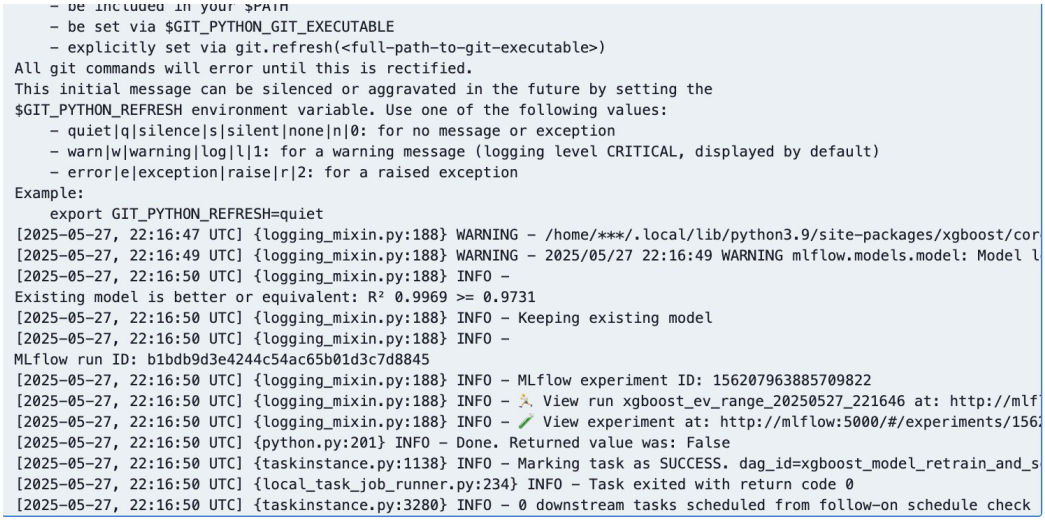
2. Data Flow Integration

- Kafka continuously streams new vehicle data into AWS RDS.
- Airflow accesses RDS, converts the data into CSV, and passes it to MLflow for retraining.
- The system closes the loop by ensuring that the model evolves, reflecting the most recent patterns in EV range data.

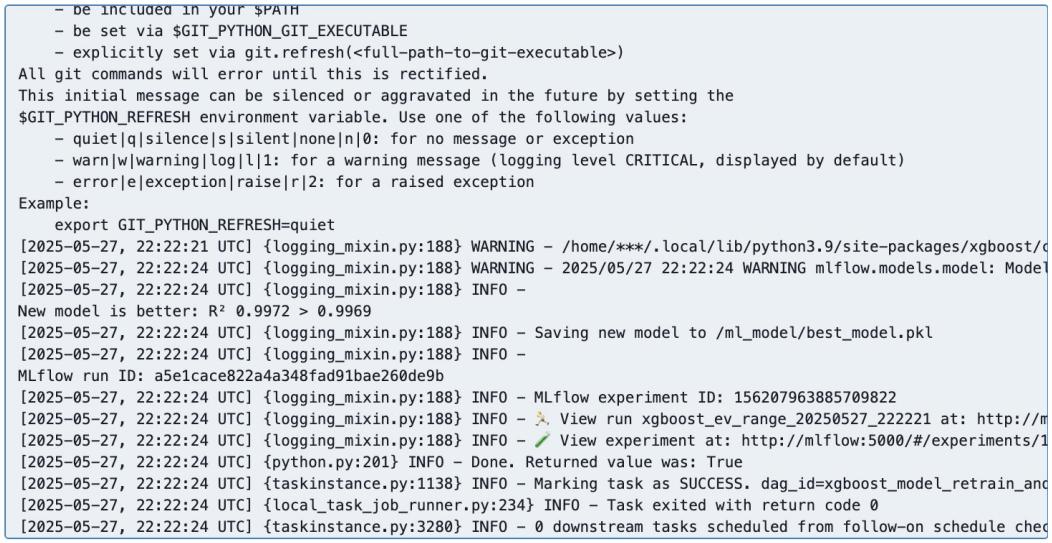


Airflow schedule runs tasks successfully, Our airflow consists of 2 jobs:

1. **download_data_from_rds:** download data from AWS RDS and save as csv file.
2. **retrain_and_compare:** use CSV data saved from AWS RDS and retrain the regression model with XGBoost, then compare the old model with R2, RMSE, MAE, etc.

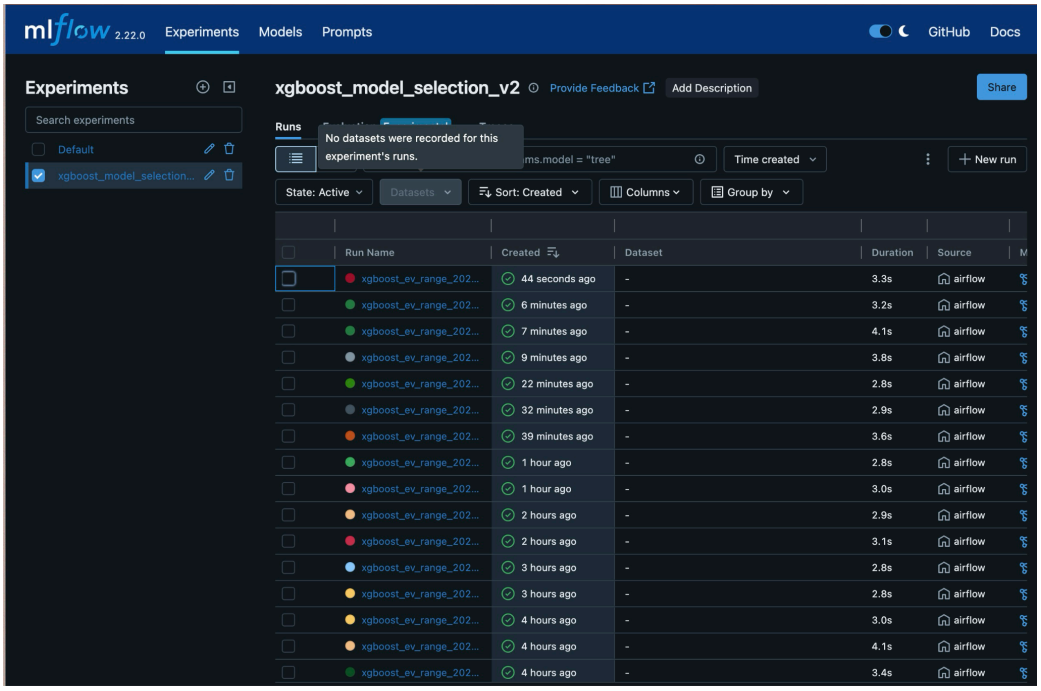


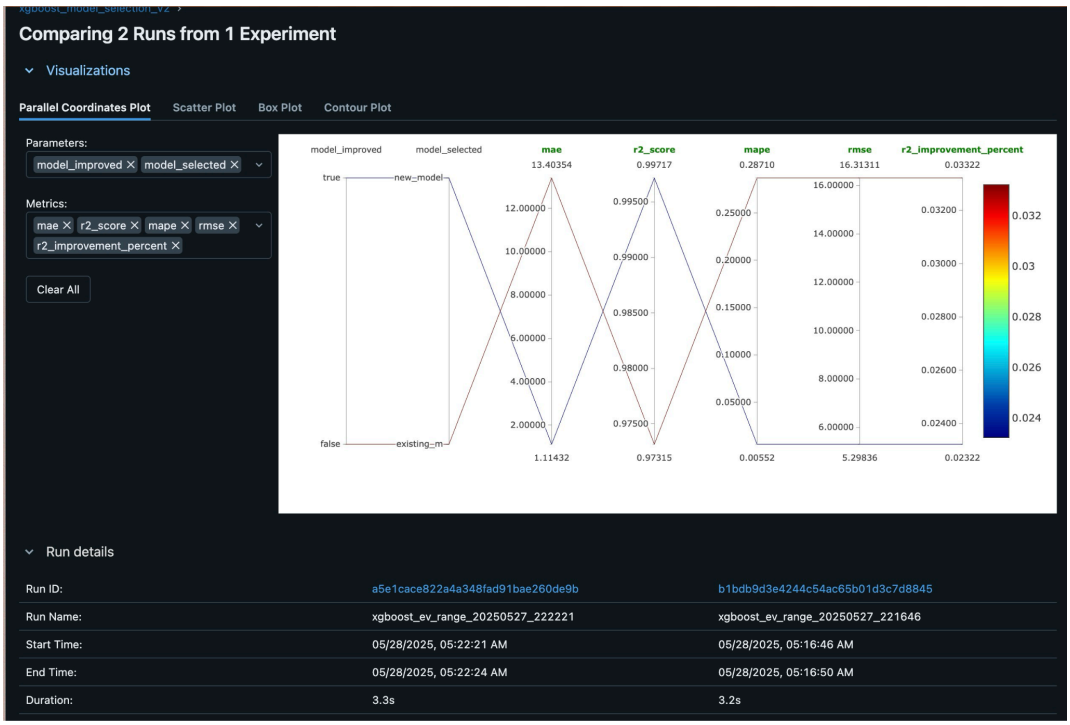
In case of the old model better, we will keep the old model to operate.



In case of a new model, better, we will replace the old regression model with a new regression model to operate.

Sources: https://github.com/nutkung1/FinalProjectCPE393/tree/main/airflow





MLFlow logs of the model have been trained and compared in runs and experiments.

Sources:

https://github.com/nutkung1/FinalProjectCPE393/tree/main/mlruns/335654393148451540

2. Feature Engineering

In the feature engineering phase, several preprocessing steps were applied to prepare the dataset for both models, namely XGBoost and ANN (Artificial Neural Network). Irrelevant or redundant columns such as Legislative District, Vehicle Location, Postal Code, City, 2020 Census Tract, County, and Electric Utility were dropped to reduce noise and simplify the model. Records with an Electric Range value of zero were excluded, as these do not contribute useful information to predicting electric vehicle range. The target variable was defined as Electric Range, while the remaining features were used as predictors. Any missing values in the numerical features were imputed using the mean of each column to ensure a complete dataset. Finally, the data was split into training and testing sets using an 80/20 split to enable model training and performance evaluation.

3. Model Selection

a. XGBoost Model

i. For this regression problem, I employed the XGBoost (Extreme Gradient Boosting) model, which is well known for its performance, speed, and ability to handle complex data structures. XGBoost is a powerful ensemble learning algorithm based on gradient boosting that builds decision trees sequentially, each one learning from the errors of the previous trees. It supports regularization and handles missing values internally, making it a robust choice for structured data.

4. **Hyperparameter Tuning**
    a. XGBoost Model
        i. To enhance model performance, hyperparameter tuning was performed using Bayesian optimization with BayesSearchCV from the skopt library. A range of hyperparameters for the gradient boosting regressor was defined, including learning_rate, max_depth, subsample, colsample_bytree, reg_lambda, reg_alpha, and n_estimators. The search space included both continuous (Real) and discrete (Integer) parameters, sampled from uniform distributions within specified bounds. The optimization objective was to minimize the negative mean absolute error using 10-fold cross-validation, ensuring robust performance estimation. A total of 120 iterations were run, evaluating one parameter configuration per iteration. Gaussian Process (GP) was used as the surrogate model for the optimization process, allowing for an efficient exploration of the hyperparameter space. The tuning process was set to be reproducible with a fixed random seed and was configured to avoid overfitting by disabling refitting and training score evaluation. This session took us several hours (10+ hours to be precise) to find the best hyperparameters.
        ii. For the records, BayesSearchCV is an advanced hyperparameter optimization method that uses Bayesian optimization to efficiently search for the best combination of model parameters. Unlike traditional grid search or random search, which try out many combinations without learning from previous results, Bayesian optimization builds a probabilistic model of the objective function (e.g., validation error) and uses it to choose the next set of parameters to test. This approach balances exploration (trying new areas of the parameter space) with exploitation (focusing on promising regions), making it significantly more efficient, especially when the parameter space is large or the model is computationally expensive to train. In this project, BayesSearchCV was used to fine-tune the hyperparameters of a regression model, improving prediction accuracy while reducing the computational cost of tuning.

b. ANN (MLP)

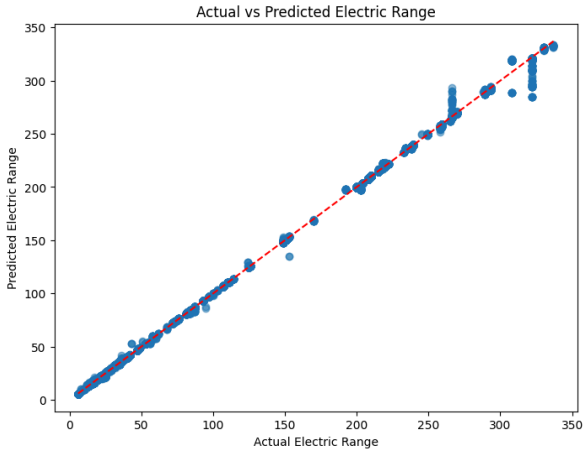
i. We explored a neural network approach using an MLP with residual connections for electric range prediction. This architecture incorporated skip connections inspired by ResNet to help with gradient flow during training.

```
def residual_block(x, units):
    shortcut = x
    x = layers.Dense(units, activation='relu')(x)
    x = layers.Dense(units)(x)
    x = layers.Add()([shortcut, x])
    x = layers.Activation('relu')(x)
    return x
```

While this model achieved reasonable performance (visible in the scatter plot of actual vs. predicted values), we ultimately selected XGBoost for our production implementation due to:

1. Superior accuracy: XGBoost consistently produced lower MAE and RMSE metrics
2. Faster training: XGBoost required significantly fewer computational resources
3. Better interpretability: Feature importance measures provided clearer insights
4. Simpler deployment: The trained model had a smaller footprint and fewer dependencies

This exploration highlights our model selection process, where we evaluated multiple approaches before selecting the most effective solution for our specific requirements.



5. **Model Training and Evaluation**
    a. XGBoost Model
        i. For this regression task, we trained two versions of the XGBoost model: a baseline (vanilla) model with default or minimally specified parameters, and an optimized version using hyperparameter tuning. The baseline model served as a reference point, while the tuned model aimed to improve prediction accuracy. Hyperparameter tuning was performed using BayesSearchCV, optimizing parameters such as learning_rate, max_depth, n_estimators, subsample, and regularization terms (reg_alpha and reg_lambda). The best model was configured with tree_method="hist" for faster histogram-based training and device="cuda" to leverage GPU acceleration. After we finished training, we used pickle to save the model for deployment.
        ii. The performance of both models was evaluated using key regression metrics. The tuned model significantly outperformed the baseline. It achieved a Mean Absolute Error (MAE) of 0.0738, a Mean Squared Error (MSE) of 0.5132, a Root Mean Squared Error (RMSE) of 0.7164, and a Mean Absolute Percentage Error (MAPE) of 0.10%. In contrast, the baseline model yielded an MAE of 0.1327 and an MSE of 0.4167. These results demonstrate that hyperparameter optimization led to more accurate and reliable predictions of electric vehicle range.
6. **Model Deployment**
    a. Containerization Strategy
        i. Microservice architecture with a separate model server and frontend components
        ii. Docker multi-container deployment utilizing lightweight Python base images
        iii. Environment isolation through container-specific configurations
        iv. Resource allocation optimization for inference workloads
    b. Model Serving Infrastructure
        i. FastAPI implementation for high-performance, asynchronous API endpoints
        ii. Hugging Face model integration for computer vision capabilities
        iii. Separate endpoints for classification and regression services
        iv. Swagger/OpenAPI documentation for API discovery and testing
    c. Orchestration and Scalability
        i. Docker Compose implementation for multi-service coordination
        ii. Integration with Airflow for scheduled model retraining
        iii. Health check endpoints for load balancer compatibility
        iv. Stateless design for horizontal scaling capabilities

d. Deployment Automation
    i. Automated Docker image builds triggered by code changes
    ii. Version tagging for release management
    iii. Runtime configuration through environment variables
    iv. Zero-downtime deployment strategy
e. Infrastructure as Code
    i. Declarative configuration for all infrastructure components
    ii. Version-controlled deployment specifications
    iii. Environmental parity between development and production
    iv. Reproducible deployment process

7. Model Monitoring

a. Health Monitoring
    i. API-based health check implementation
    ii. Component-level status reporting
    iii. Readiness and liveness probes
    iv. Dependency availability monitoring
b. Performance Metrics
    i. Inference latency tracking
    ii. Throughput measurement methodology
    iii. Resource utilization monitoring
    iv. Scaling trigger implementation
c. Model Quality Monitoring
    i. Prediction Drift Detection Approach
    ii. Confidence score tracking
    iii. Model accuracy evaluation methodology
    iv. Outlier detection implementation
d. Operational Considerations
    i. Logging and Observability
    ii. Structured logging implementation
    iii. Error tracking and classification
    iv. Request/response logging for audit trails
    v. Debugging capabilities in production
e. Security Implementation
    i. Input validation and sanitization
    ii. Rate limiting and request throttling
    iii. Authentication and authorization framework
    iv. Sensitive data handling practices
f. Disaster Recovery
    i. Backup and restore procedures
    ii. Model versioning and rollback capabilities

   iii. High availability configuration
   iv. Failure recovery automation
g. Feedback Loops
   i. Automated retraining trigger mechanisms
   ii. Performance threshold monitoring
   iii. A/B testing infrastructure
   iv. User feedback integration
h. Model Versioning Strategy
   i. Model metadata tracking
   ii. Version compatibility management
   iii. Canary deployment approach
   iv. Gradual rollout methodology

8. Classification Model

a. Objective

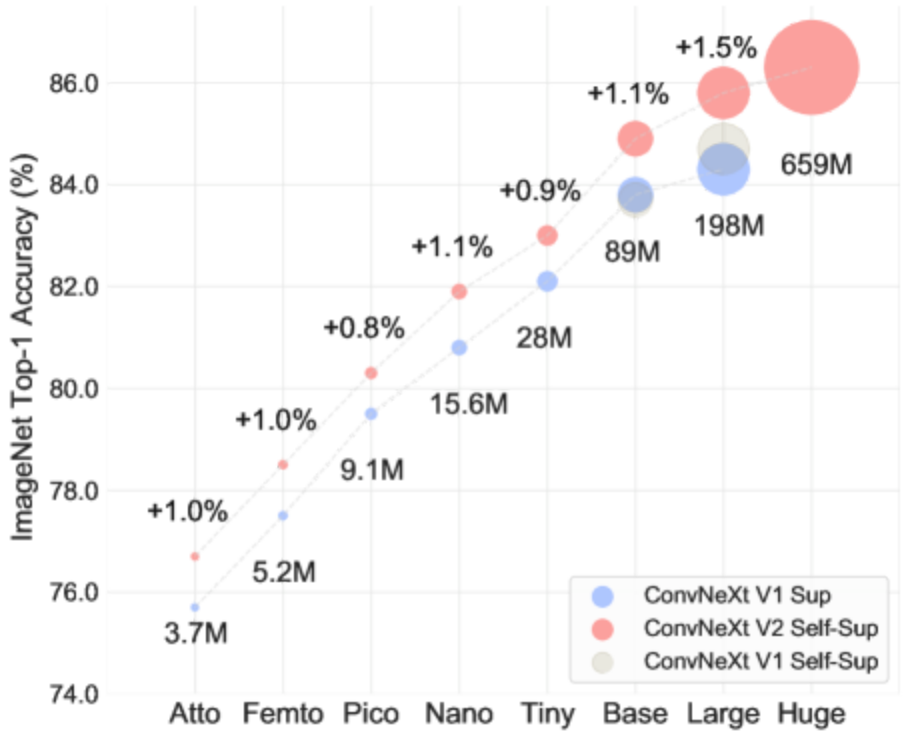
i. This part aims to develop an accurate car image classification model capable of identifying different car makes and models from images. This model serves as a component in a larger MLOps pipeline for electric vehicle analysis, enabling automated identification of vehicle types from photographs.

b. Model Selection

i. For this project, the ConvNeXTv2 Tiny model from Facebook (facebook/convnextv2-tiny-1k-224) was chosen as the base architecture for the car classification task. The decision to use this model was influenced by both its technical strengths and practical considerations related to data availability.

1. Efficiency: ConvNeXTv2 Tiny provides an excellent trade-off between computational cost and predictive performance, making it suitable for projects with limited resources.
2. Modern Design: It incorporates cutting-edge architectural features such as depthwise separable convolutions and layer normalization, which contribute to better model performance and training stability.
3. Transfer Learning: Pre-trained on the large-scale ImageNet dataset (with 1,000 classes), the model is highly effective at extracting general visual features. This makes it an ideal candidate for transfer learning, especially when task-specific data is limited.
4. Data Constraints: One of the major challenges in this project was the **lack of a large, labeled dataset specific to electric vehicles (EVs).** Training a model from scratch would require a significant amount of high-quality, labeled data, which is not readily available. Fine-tuning a robust pre-trained model like ConvNeXTv2 allows for effective adaptation to the EV classification task with far less data and training time.
5. Seamless Integration: The model integrates easily with the Hugging Face Transformers ecosystem, streamlining the fine-tuning and deployment process.

To adapt the model to the specific classification task, the original classification head was replaced with a custom head matching the number of target classes in the dataset. This approach leverages the rich feature representations learned from ImageNet while focusing the model on the domain-specific task of EV classification.



c. Train & Test Dataset

i. We utilized the Hugging Face datasets library to efficiently manage and preprocess the dataset, organizing the images into a structured folder format suitable for image classification tasks. Due to the absence of publicly available datasets specifically focused on electric vehicle (EV) models, we curated our own dataset by scraping images from the internet as part of a proof of concept (POC). The dataset consists of 160 images, with 20 images for each of the 8 EV car classes: CHEVROLET_BOLT, KIA_NIRO, NISSAN_LEAF, RIVIAN_R1T, TESLA_3, TESLA_S, TESLA_X, and TESLA_Y. For evaluation, we reserved 5 images per class as the test set to assess model performance. This custom dataset allowed us to validate the feasibility of fine-tuning a pre-trained model for EV car classification despite the limited data availability.

d. Data Augmentation and Normalization

i. To enhance the robustness of the model and improve generalization, a series of data augmentation and preprocessing steps were applied using the torchvision.transforms module. The image transformation pipeline includes:

1. RandomResizedCrop: Randomly crops and resizes the image to a target size defined by image_processor.size["shortest_edge"], encouraging the model to learn features from various scales and image regions.

2. RandomHorizontalFlip: Randomly flips images horizontally during training to simulate real-world variations in image orientation.
3. ToTensor: Converts images from PIL format to PyTorch tensors and scales pixel values to the [0, 1] range.
4. Normalize: Applies normalization using the mean and standard deviation values from the image_processor, typically precomputed from the ImageNet dataset. This ensures the input distribution aligns with that of the pre-trained ConvNeXTv2 model.

A custom function, train_transforms() was defined to apply these transformations to each training example. Specifically, it converts images to RGB (to ensure consistency across channels), applies the composed transformations, and stores the processed images under the pixel_values key. Normalization is a critical step, as it standardizes the input data, helping the model converge faster and more effectively during training.

e. Fine-tuning Model

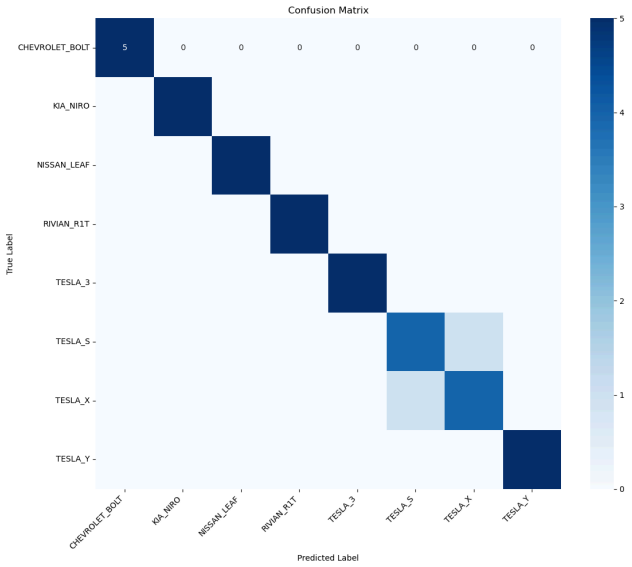
i. The fine-tuning process was conducted using a pre-trained ConvNeXTv2 Tiny model (facebook/convnextv2-tiny-1k-224) adapted for image classification tasks through the Hugging Face transformers library. The model was initialized with custom id2label and label2id mappings to align with the EV car classes and loaded with ignore_mismatched_sizes=True to allow for replacement of the classification head. A custom data loading pipeline was created using PyTorch's DataLoader, with a collate_fn to stack the transformed image tensors and their corresponding labels. Training was performed on a GPU (if available) using the AdamW optimizer with a learning rate of 5e-5. The model was trained over 10 epochs, and each batch was moved to the appropriate device. After each forward pass, the loss and logits were computed, followed by backpropagation and an optimizer step. Metrics such as accuracy were tracked throughout training to monitor performance. This fine-tuning approach allowed the pre-trained model to adapt its learned visual representations to the custom EV classification task, even with a limited dataset, effectively demonstrating the power and practicality of transfer learning. After that, we saved the model into Huggingface for deployment.

f. Evaluation Metrics

i. To assess the performance of the fine-tuned ConvNeXTv2 Tiny model, we evaluated it on a held-out test dataset consisting of 40 images, 5 samples for each of the 8 electric vehicle classes. The evaluation process involved generating predictions using the trained model and comparing them against the true labels. We calculated key classification metrics, including accuracy, precision, recall, and F1-score, and visualized the results using a confusion matrix.

The model achieved an impressive overall accuracy of 95%, correctly predicting 38 out of 40 test samples. Most classes achieved perfect scores in precision, recall, and F1-score, demonstrating the model's strong generalization capabilities despite the limited dataset. However, there was a slight drop in performance for the TESLA_S and TESLA_X classes, both achieving 0.80 in precision and recall, indicating potential confusion between visually similar models.

The confusion matrix provided further insight into class-wise performance, highlighting where misclassifications occurred. Additionally, a detailed classification report was generated to summarize per-class performance metrics. These results validate the effectiveness of using a pre-trained model with transfer learning, even in constrained data scenarios, and demonstrate the model's capability to distinguish between various EV models with high accuracy.



Sources: https://github.com/nutkung1/FinalProjectCPE393/tree/main/ml_model

User Interface

We deploy the front end on Streamlit Cloud, linking with the GitHub repository. The user interface (UI) serves as the primary interaction point between the end-user and the system. It is designed to be intuitive, user-friendly, and responsive, providing two core functionalities: classification-based electric range aggregation and regression-based electric range prediction. Both modules aim to assist users in understanding and estimating the electric range of electric vehicles (EVs) through different types of input: image-based classification and form-based prediction.

In the user interface, we use a tool called Streamlit to create a website that allows users to easily interact with our machine learning models through a simple and user-friendly web application.

On our website, we designed it to have two main functions:

1. **Regression**
   Users can enter information about electric vehicles, such as model year, vehicle model, battery capacity, and type of electric vehicle.
   The system will then use a trained regression model to predict the expected driving range (in miles) for that specific EV configuration.
   This helps users, manufacturers, or researchers gain insights into vehicle efficiency and supports data-driven decision-making.

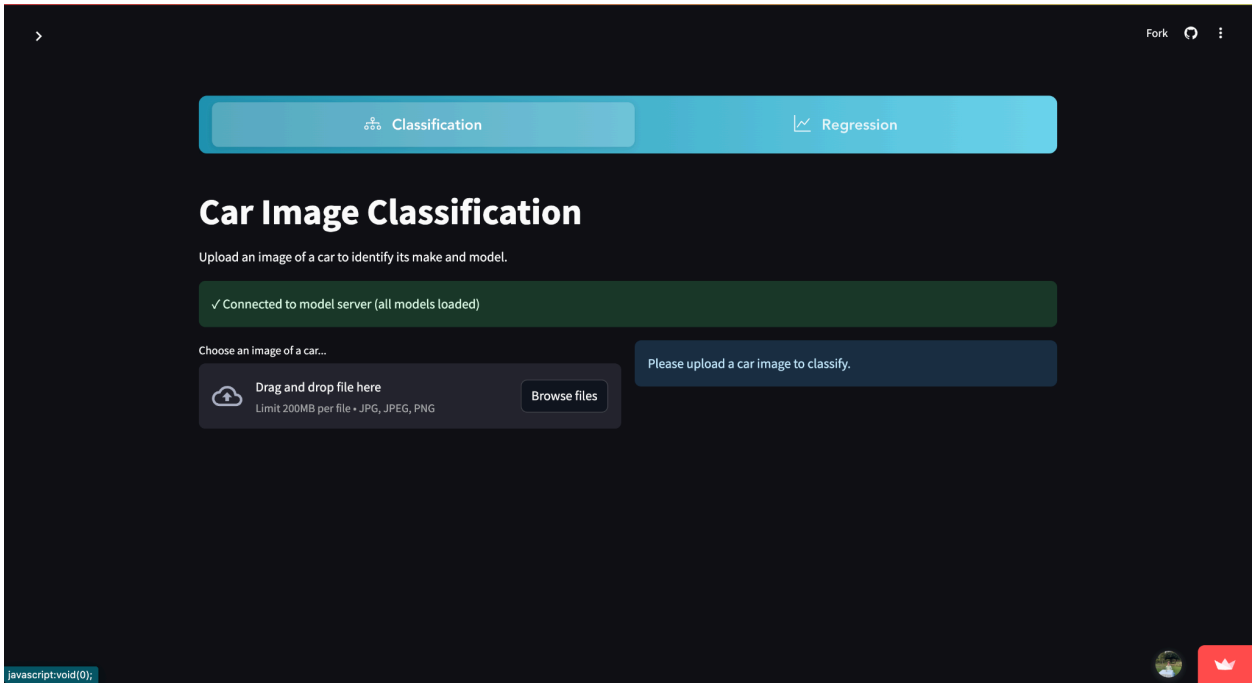
2. **Classification**
   Users can upload a picture of an electric vehicle, and the system will use the pre-trained machine learning classification model to analyze the image and predict the most likely brand or model of the vehicle.
   The system will display the top 3 predicted results along with the confidence scores in both tabular and graphical formats, allowing users to understand the prediction clearly.

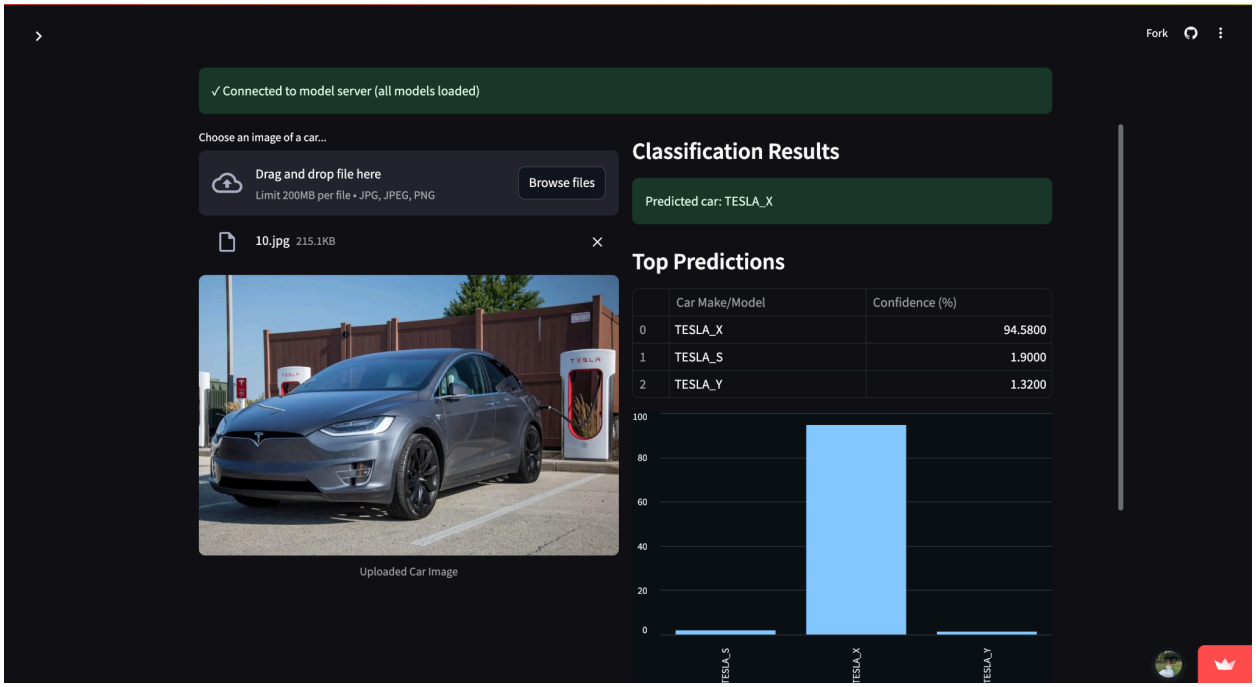
Website: https://finalprojectcpe393.streamlit.app/

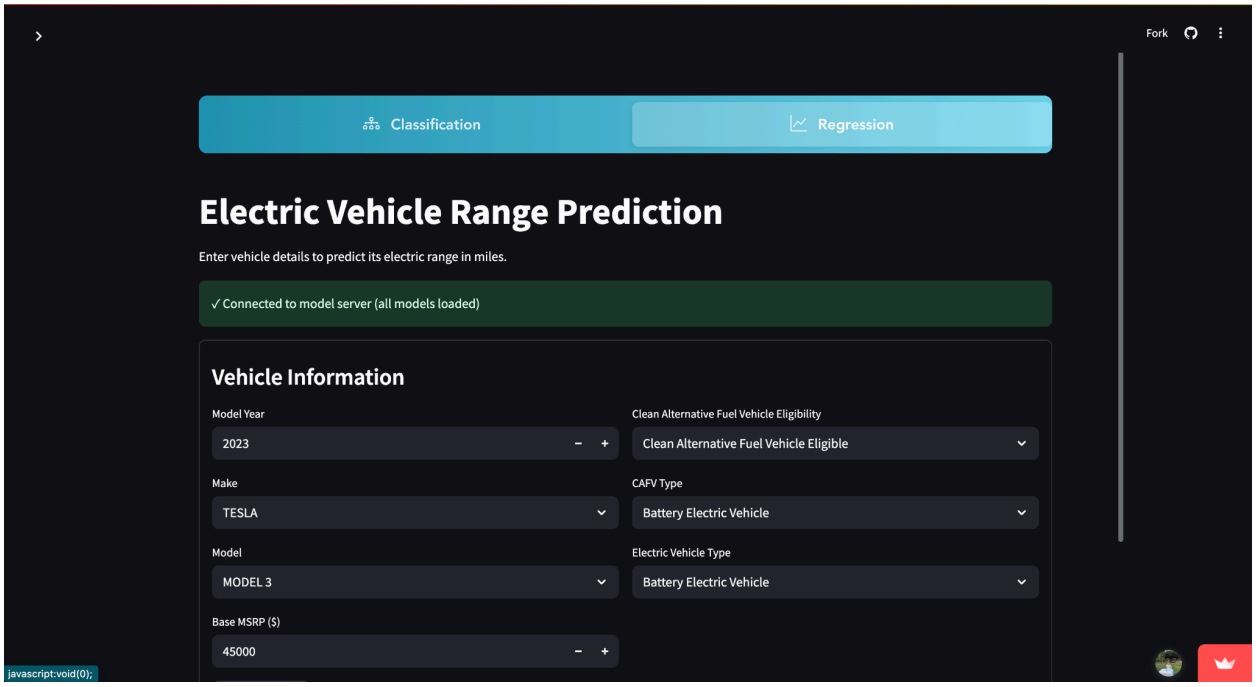
Overall UI Features:

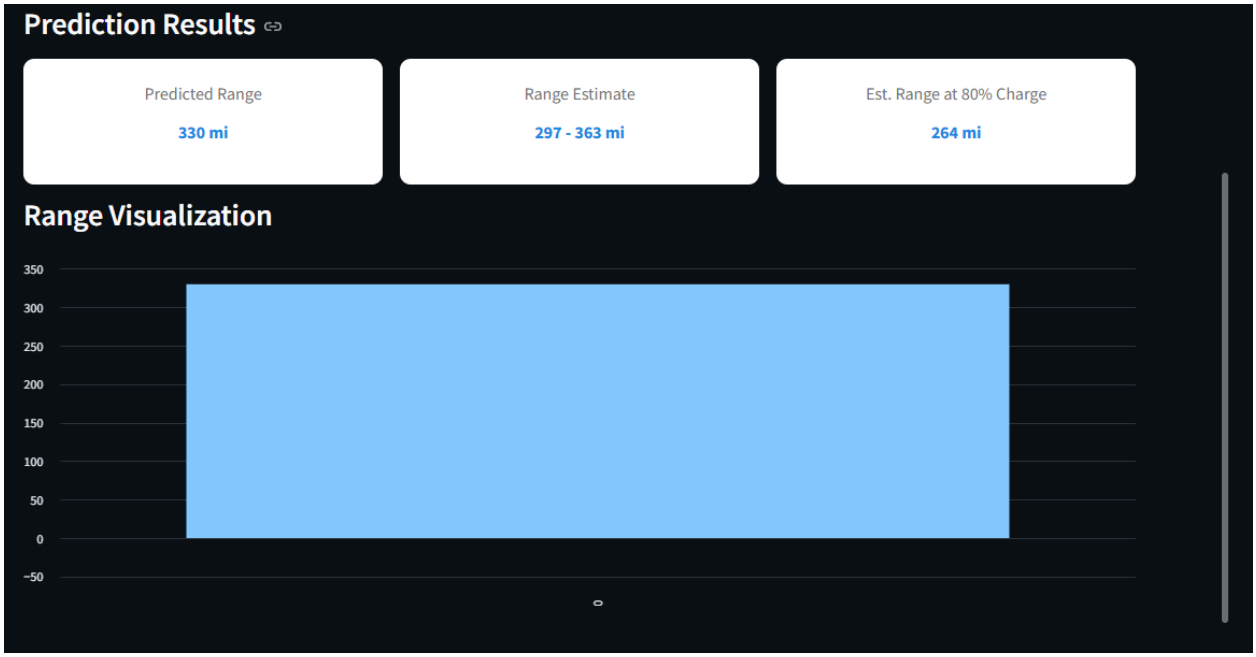
Feature	Description
Responsive Design	Mobile and desktop-friendly interface
Image Upload + Preview	For classification input
Auto-filled Results	For predicted range and database summaries
Form Validation	Ensures all required fields are input correctly for regression
Visualization	Range can be visualized as a bar chart or gauge meter for better UX

1. **Windows or Mac**

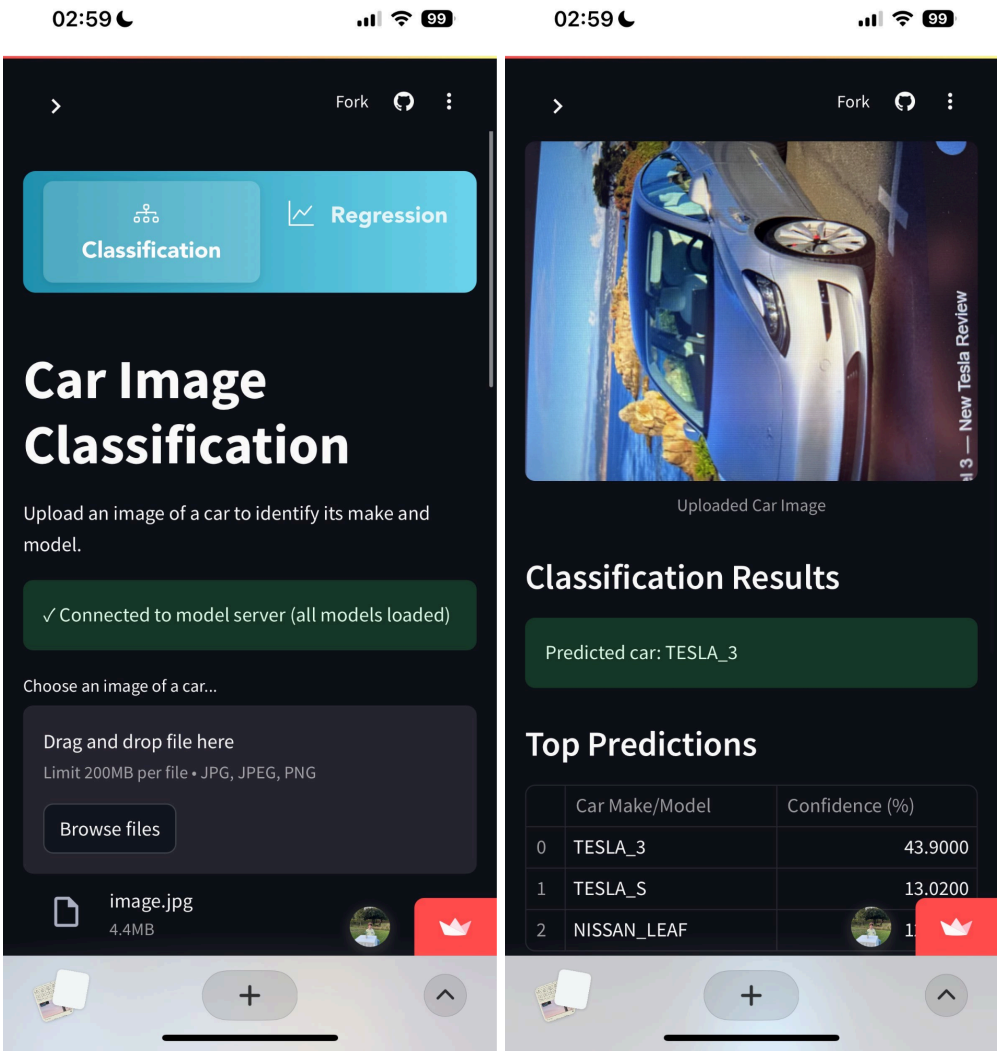


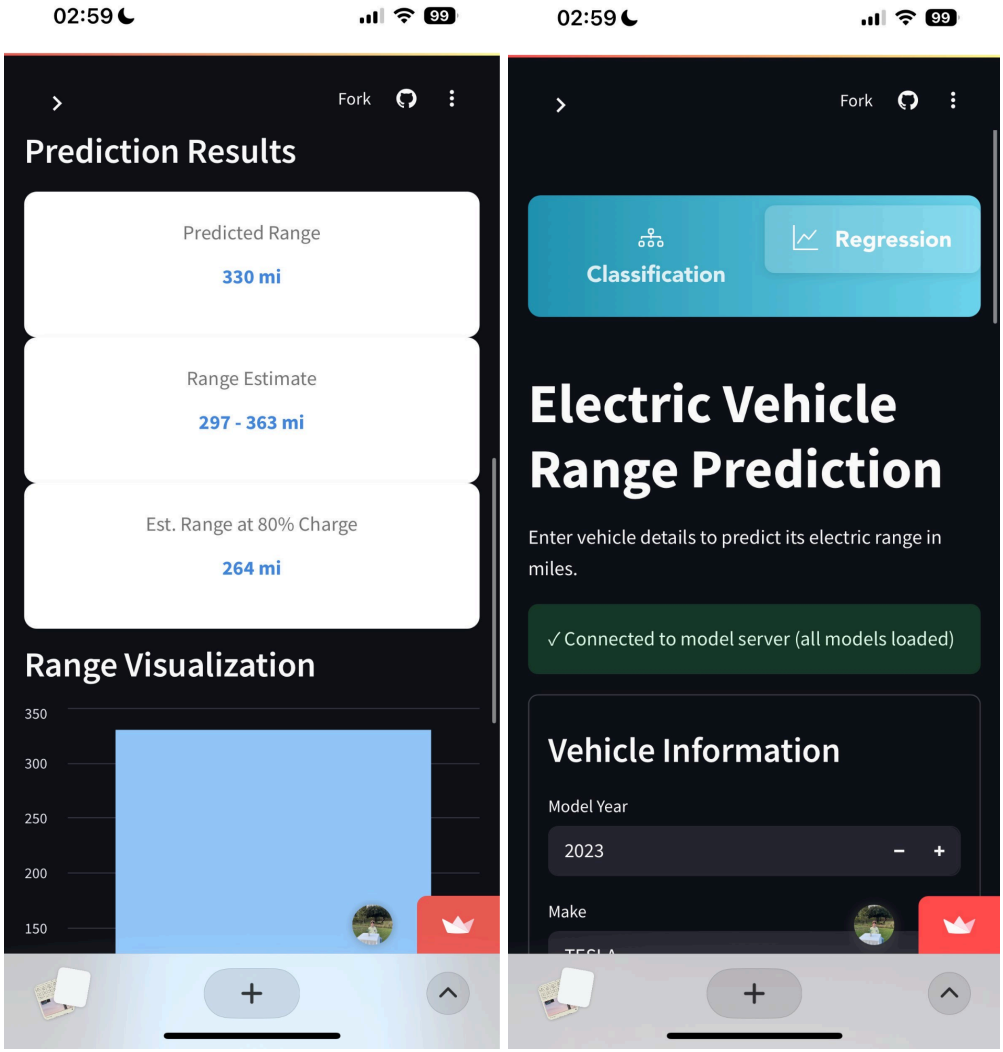


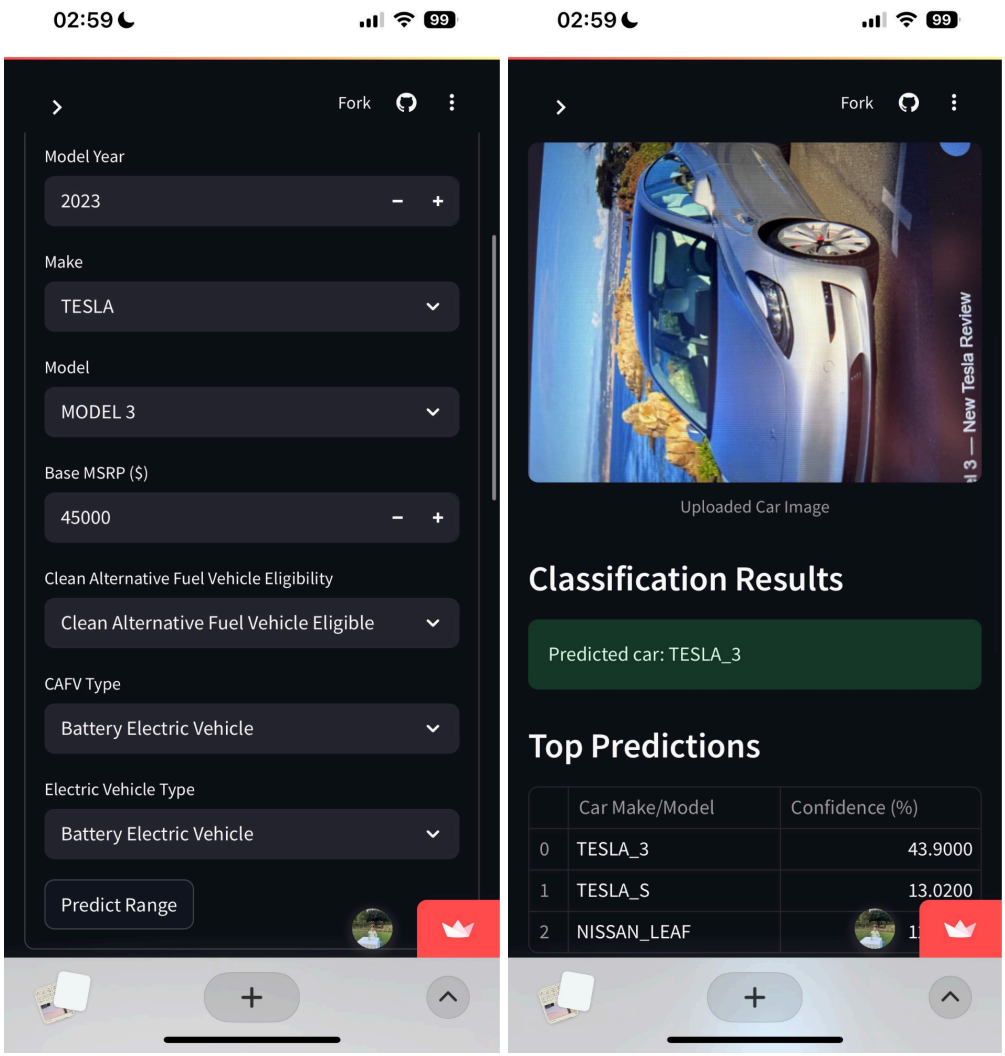

Fork
Classification
Regression
Electric Vehicle Range Prediction
Enter vehicle details to predict its electric range in miles.
✓ Connected to model server (all models loaded)
Vehicle Information
Model Year
2023
Clean Alternative Fuel Vehicle Eligibility
Clean Alternative Fuel Vehicle Eligible
Make
TESLA
CAFV Type
Battery Electric Vehicle
Model
MODEL 3
Electric Vehicle Type
Battery Electric Vehicle
Base MSRP ($)
45000
javascript:void(0);


Prediction Results
Predicted Range
330 mi
Range Estimate
297 - 363 mi
Est. Range at 80% Charge
264 mi
Range Visualization
350
300
250
200
150
100
50
0
−50

2. Phone


02:59
99
Fork
Classification
Regression
Car Image Classification
Upload an image of a car to identify its make and model.
✓ Connected to model server (all models loaded)
Choose an image of a car...
Drag and drop file here
Limit 200MB per file • JPG, JPEG, PNG
Browse files
image.jpg
4.4MB
02:59
99
Fork
Uploaded Car Image
Classification Results
Predicted car: TESLA_3
Top Predictions
Car Make/Model
Confidence (%)
0 TESLA_3 43.9000
1 TESLA_S 13.0200
2 NISSAN_LEAF


02:59
99
Fork
Prediction Results
Predicted Range
330 mi
Range Estimate
297 - 363 mi
Est. Range at 80% Charge
264 mi
Range Visualization
350
300
250
200
150
02:59
99
Fork
Classification
Regression
Electric Vehicle Range Prediction
Enter vehicle details to predict its electric range in miles.
✓ Connected to model server (all models loaded)
Vehicle Information
Model Year
2023
Make


02:59
99
Fork
Model Year
2023
Make
TESLA
Model
MODEL 3
Base MSRP ($)
45000
Clean Alternative Fuel Vehicle Eligibility
Clean Alternative Fuel Vehicle Eligible
CAFV Type
Battery Electric Vehicle
Electric Vehicle Type
Battery Electric Vehicle
Predict Range
02:59
99
Fork
Uploaded Car Image
Classification Results
Predicted car: TESLA_3
Top Predictions
Car Make/Model
Confidence (%)
0 TESLA_3 43.9000
1 TESLA_S 13.0200
2 NISSAN_LEAF

Back End

We use ngrok to expose our port, so the front-end on Streamlit Cloud can communicate with the back-end on the local computer.

Model Serving Infrastructure

The backend system is implemented as a RESTful API service using FastAPI, a modern, high-performance web framework for building APIs with Python. This architecture provides asynchronous request handling capabilities, automatic OpenAPI documentation, and robust error handling mechanisms essential for production machine learning deployments.

The server exposes multiple endpoints designed to handle different prediction tasks while maintaining statelessness to support horizontal scalability.

Lazy Loading Pattern

The system implements a lazy loading pattern for model initialization using FastAPI's event handlers. This approach optimizes resource utilization by deferring model loading until the application startup phase:

```
@app.on_event("startup")
async def load_models():
    global feature_extractor, classification_model, regression_model, ev_data
```

Multi-Model Architecture

The backend maintains three distinct components to support diverse prediction capabilities:

```
classification_model = AutoModelForImageClassification.
from_pretrained(
    "dreamypancake/fine_tune_Car_ConvNeXTv2"
)
```

1. Feature Extraction Model: Leverages the Hugging Face AutoImageProcessor for preprocessing image inputs
2. Classification Model: Utilizes a fine-tuned ConvNeXTv2 architecture for vehicle type identification
3. Regression Model: Employs an XGBoost model for electric range prediction based on vehicle specifications

This separation of concerns allows for independent scaling, versioning, and optimization of each model component.

Data Model Design

The system implements strongly-typed data models using Pydantic to ensure robust input validation and clear API contracts:

```python
class ImageRequest(BaseModel):
    image_base64: str

class PredictionResponse(BaseModel):
    predicted_class: str
    top_predictions: dict
    electric_range_stats: dict = None
```

These models enforce schema validation at runtime, reducing the potential for invalid inputs and ensuring consistent response structures.

Prediction Endpoints

Image Classification Service

The /predict endpoint handles computer vision tasks for electric vehicle identification:

```python
@app.post("/predict", response_model=PredictionResponse)
async def predict_classification(request: ImageRequest):
```

This endpoint processes base64-encoded images, performs feature extraction, executes inference using the classification model, and returns prediction results with confidence scores. The implementation includes:

- Base64 decoding and image preprocessing
- Tensor-based inference with gradient calculation disabled for efficiency
- Softmax normalization for probability interpretation
- Top-k prediction selection for multiple candidate results

Range Prediction Service

The /predict_range endpoint provides electric vehicle range estimation based on multiple vehicle attributes:

```python
@app.post("/predict_range", response_model=RegressionResponse)
async def predict_regression(request: RegressionRequest):
```

This endpoint performs:

- Categorical feature encoding via mapping dictionaries
- Feature alignment with model expectations
- XGBoost inference for range prediction
- Confidence interval calculation for uncertainty estimation

Error Handling and Resilience

The system implements a comprehensive error-handling strategy to ensure robustness:

```python
try:
    # Processing logic
except Exception as e:
    import traceback
    print(traceback.format_exc())
    raise HTTPException(status_code=500, detail=f"Error processing image: {str(e)}")
```

Key resilience features include:

- Pre-flight model availability checks
- Structured exception handling with detailed error messages
- Graceful degradation when auxiliary data is unavailable
- Comprehensive logging for debugging and monitoring

Health Monitoring

The system provides a dedicated health check endpoint that reports on the operational status of critical components:

```python
@app.get("/health")
def health_check():
    return {
        "status": "healthy",
        "classification_model_loaded": classification_model is not None,
        "regression_model_loaded": regression_model is not None,
    }
```

This endpoint enables infrastructure monitoring tools to assess service readiness and perform automated recovery actions when necessary.

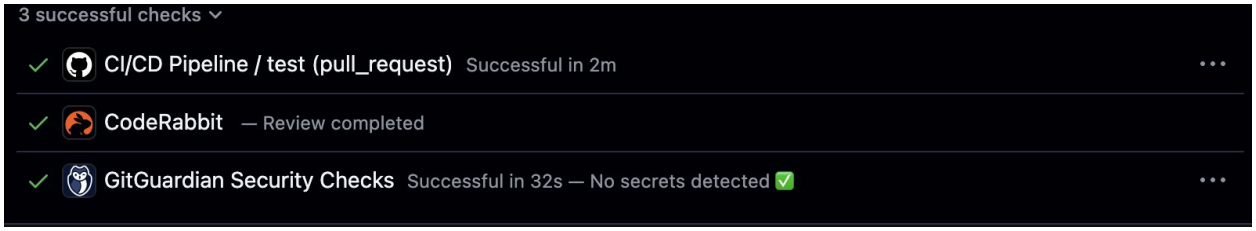
Production Deployment Considerations

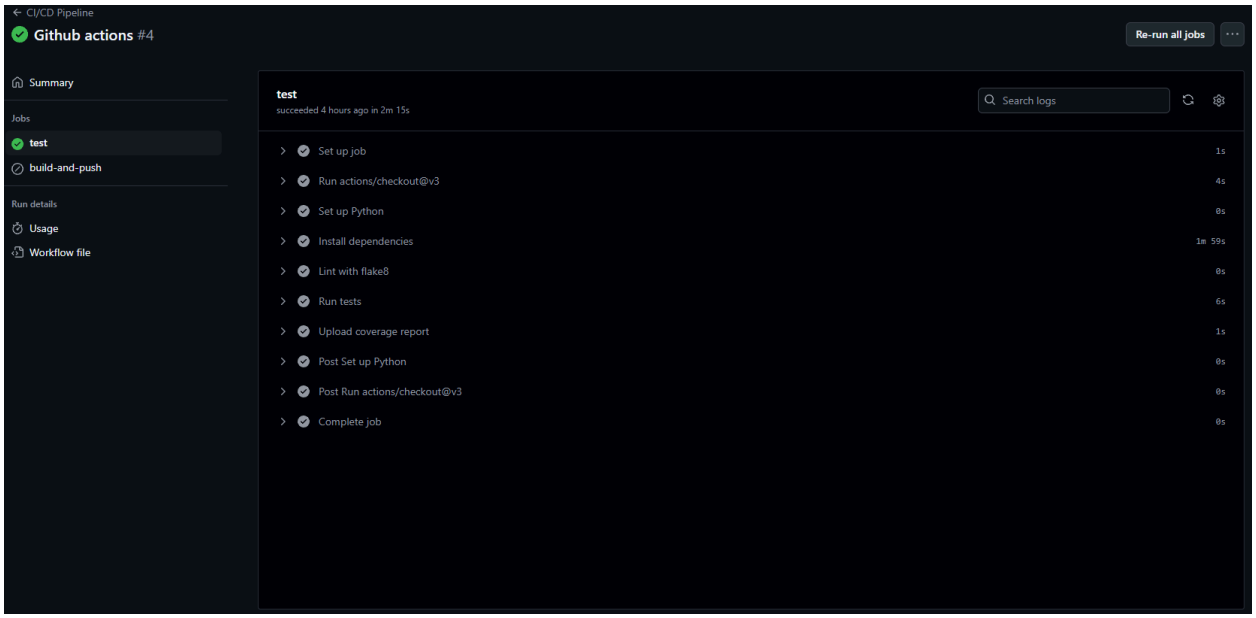
The server is configured for containerized deployment with appropriate host binding and production-mode settings:

```python
if __name__ == "__main__":
    uvicorn.run("model_server:app", host="0.0.0.0", port=8000, reload=False)
```

This configuration supports Docker deployment while disabling the development-only auto-reload feature to ensure stable operation in production environments.

CI/CD





This CI/CD pipeline defines an automated CI/CD pipeline that runs whenever code changes are pushed or pull requests are created.

1. Pipeline Architecture Overview

The implemented continuous integration and continuous deployment (CI/CD) pipeline utilizes GitHub Actions as the primary workflow orchestration platform. This pipeline follows modern MLOps principles by automating testing, building, and deployment processes within a single workflow definition.

2. Workflow Configuration Components

The workflow is defined using YAML syntax in the ci-cd.yml file. It leverages GitHub's event-driven architecture to trigger pipeline execution based on repository events.

3. Testing Environment & Quality Assurance Tools

3.1 Runtime Environment

- actions/checkout@v3: Git repository content retrieval tool
- actions/setup-python@v4: Python environment configuration utility (version 3.10)

3.2 Static Code Analysis

- Flake8: Python linting library that enforces PEP 8 style guide
- Configured to detect syntax errors (E9)
- Identifies undefined name errors (F82)
- Detects breakage of function call arguments (F7)
- Catches syntax errors in string formatting (F63)

3.3 Test Execution Framework

- pytest: Test discovery and execution framework
- pytest-cov: Test coverage measurement extension
- Generates XML coverage reports for further analysis

3.4 Quality Metrics Integration

- codecov/codecov-action@v3: Test coverage analytics platform integration
- Uploads coverage data to the Codecov dashboard
- Configured with failure tolerance to prevent pipeline disruption

4. Container Build & Deployment Infrastructure

4.1 Container Build Tools

- docker/setup-buildx-action@v2: Enhanced Docker build engine
- Provides multi-platform build capabilities
- Offers build caching for improved performance

4.2 Registry Authentication

- docker/login-action@v2: DockerHub authentication utility
- Leverages repository secrets for credential management

4.3 Image Build & Publication

- docker/build-push-action@v4: Container image build orchestrator
- Manages context and builds file specification
- Handles tagging and registry pushing operations
- Creates two distinct application images:
    - ML Model Server (ev-model-server:latest)
    - Frontend Application (ev-app:latest)

Workflow Triggers

1. Activates push to the main branch
2. Activates on pull requests targeting the main branch

Job 1: Test: This job verifies your code quality

- Setup Environment:
    - Uses Ubuntu as the runner
    - Sets up Python 3.10
    - Installs your project dependencies from requirements.txt
    - Installs testing tools
- Code Quality Checks:
    - Runs flake8 to catch syntax errors and undefined variables
    - Execute your tests with coverage reporting
    - Uploads coverage reports to Codecov

Job 2: Build and Push: This job creates and publishes Docker images

- Conditional Execution:
    - Only runs if tests pass
    - Only executes on direct pushes to the main
- Docker Operations:
    - Sets up Docker Buildx for efficient image building
    - Authenticates with DockerHub using your stored credentials
    - Builds and pushes two Docker images:
        - Model server image (from Dockerfile.model)
        - Frontend app image (from Dockerfile.app)
    - Tag images with your DockerHub username

CI

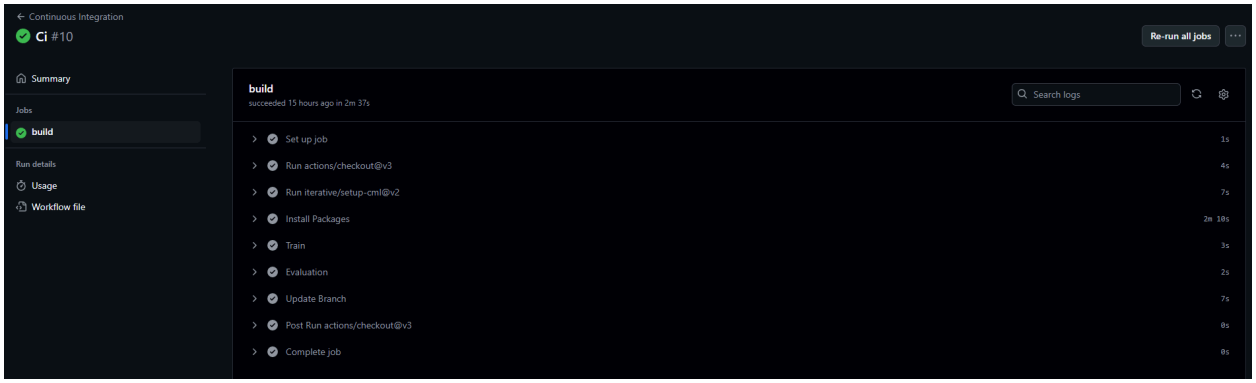
To ensure reproducibility, automation, and continuous feedback in the machine learning workflow, a Continuous Integration (CI) pipeline was implemented using GitHub Actions in combination with CML (Continuous Machine Learning). The pipeline is defined in the ci.yml file and is triggered on three events: pushes to the main branch, pull requests targeting main, and manual workflow dispatches.

Workflow Overview:

1. **Checkout Code**: The pipeline begins by checking out the repository using actions/checkout@v3.
2. **Set Up CML**: It sets up the environment with iterative/setup-cml@v2 to enable integration with CML tools for reporting machine learning metrics directly into pull requests or commits.
3. **Install Dependencies**:
    - The make install command runs the install target from the Makefile, which upgrades pip and installs all required dependencies from requirements.txt.
4. **Train the Model**:
    - The make train step executes the train target, running train.py to train the machine learning model.
5. **Evaluate the Model**:
    - The make eval step evaluates the model and appends the metrics from ./Results/metrics.txt to a markdown file report.md.
    - Using CML, the results are automatically posted as a comment in the pull request, providing immediate visibility of performance metrics to collaborators and reviewers.
6. **Update Results Branch**:
    - The make update-branch step commits the updated report.md to a branch named update. User credentials are securely passed via GitHub Secrets (USER_NAME and USER_EMAIL) to configure Git and push the changes.

Benefits:

- **Automation**: Reduces manual effort by automating training and evaluation.
- **Transparency**: Makes model performance visible and traceable via CML comments.
- **Collaboration**: Facilitates team collaboration by posting results directly in pull requests.
- **Version Control**: Keeps evaluation reports under version control by pushing them to the repository.



Source: https://github.com/nutkung1/FinalProjectCPE393/tree/main/.github/workflows

Docker Containerization

7.1 Containerization Philosophy and Architecture

The MLOps project implements a comprehensive containerization strategy using Docker and Docker Compose to achieve environment consistency, service isolation, and deployment reproducibility. The architecture follows microservices principles, where each component operates independently while maintaining seamless inter-service communication through well-defined interfaces.

The containerized system consists of five primary service layers: the model serving layer hosting the FastAPI backend, the presentation layer running the Streamlit frontend, the orchestration layer managing Airflow workflows, the messaging layer handling Kafka-based data streaming, and the persistence layer providing PostgreSQL database services.

7.2 Model Server Container Design

The model server container encapsulates the FastAPI application responsible for serving machine learning predictions. This container utilizes a Python 3.10 slim base image to minimize resource footprint while maintaining necessary runtime dependencies. The container structure includes the main application code, trained machine learning models stored in the ml_model directory, and the Electric Vehicle dataset required for range predictions.

The container exposes port 8000 for external API access and implements proper layering to optimize build times and image size. Dependencies are installed early in the build process to leverage Docker's layer caching mechanism, while application code and data files are copied in subsequent layers to minimize rebuild times during development iterations.

7.3 Frontend Application Container

The Streamlit frontend container provides the user interface for the electric vehicle analysis system. Similar to the model server, it uses Python 3.10 slim as the base image and follows container best practices for efficient builds and deployments.

This container exposes port 8501, which is Streamlit's default port, and includes only the necessary application files to maintain a lean container image. The separation of frontend and backend containers enables independent scaling and updates of each component, supporting the microservices architecture principles.

7.4 Workflow Orchestration Infrastructure

7.4.2 Custom Airflow Container

The Airflow orchestration system utilizes a custom container built upon the official Apache Airflow 2.8.1 image. This approach allows for the installation of project-specific dependencies required by the DAGs without modifying the base Airflow installation. The custom container ensures that all necessary Python packages for model training, data processing, and external integrations are available to the workflow execution environment.

7.4.1 Multi-Service Orchestration

The Airflow infrastructure consists of multiple interconnected services orchestrated through Docker Compose. The PostgreSQL service provides metadata storage for Airflow's task scheduling and execution history. The system implements health checks to ensure proper service startup sequencing and maintains data persistence through named Docker volumes.

The web server and scheduler services work in tandem to provide the Airflow user interface and task execution capabilities. Both services share the same custom image but serve different roles within the workflow orchestration system.

7.5 Experiment Tracking Integration

MLflow is integrated into the containerized infrastructure to provide experiment tracking and model registry capabilities. The MLflow service connects to the same PostgreSQL backend used by Airflow, creating a unified metadata store for both workflow execution and experiment tracking.

This integration enables seamless tracking of model training experiments initiated through Airflow DAGs while maintaining proper artifact storage for model versioning and deployment management.

7.6 Real-Time Data Streaming Architecture

7.6.1 Kafka Messaging Infrastructure

The system incorporates Apache Kafka for real-time data streaming capabilities, supporting continuous model inference and data pipeline processing. The Kafka infrastructure consists of Zookeeper for cluster coordination and Kafka brokers for message handling.

The messaging system enables asynchronous communication between different system components and supports future scalability requirements for high-throughput data processing scenarios.

7.6.2 Producer and Consumer Services

Custom producer and consumer services are implemented as separate containers to handle data ingestion and processing workflows. These services connect to the Kafka infrastructure and can be scaled independently based on data volume and processing requirements.

The modular design allows for easy addition of new data sources or processing logic without affecting the core application functionality.

7.7 Volume Management and Data Persistence

The containerized system implements strategic volume management to ensure data persistence and enable development workflow efficiency. Named volumes preserve database contents across container restarts, while bind mounts provide access to DAG files, model artifacts, and configuration files.

This approach balances the benefits of container isolation with the practical needs of development and operations, allowing for easy updates to workflow definitions and model artifacts without container rebuilds.

7.8 Network Configuration and Service Discovery

The Docker Compose configuration establishes internal networks that enable secure inter-service communication while selectively exposing necessary ports to the host system. Services communicate using Docker's built-in DNS resolution, eliminating the need for hardcoded IP addresses and supporting dynamic service discovery.

Port exposure is limited to essential services that require external access, following security best practices for containerized applications.

7.9 Benefits and Operational Advantages

This containerization strategy provides several operational advantages, including environmental consistency across development and production, simplified dependency management, enhanced security through service isolation, and improved scalability through independent service scaling.

The infrastructure supports continuous integration and deployment workflows, enabling automated testing and deployment of individual components without affecting the entire system. This modular approach facilitates maintenance, updates, and troubleshooting while ensuring system reliability and performance.

Version Control

In this project, version control is managed using Git and GitHub, which provide a robust framework for tracking changes, collaborating effectively, and ensuring the reproducibility of machine learning workflows. The entire codebase is maintained in a Git repository, enabling structured development through commits, branches, and pull requests.

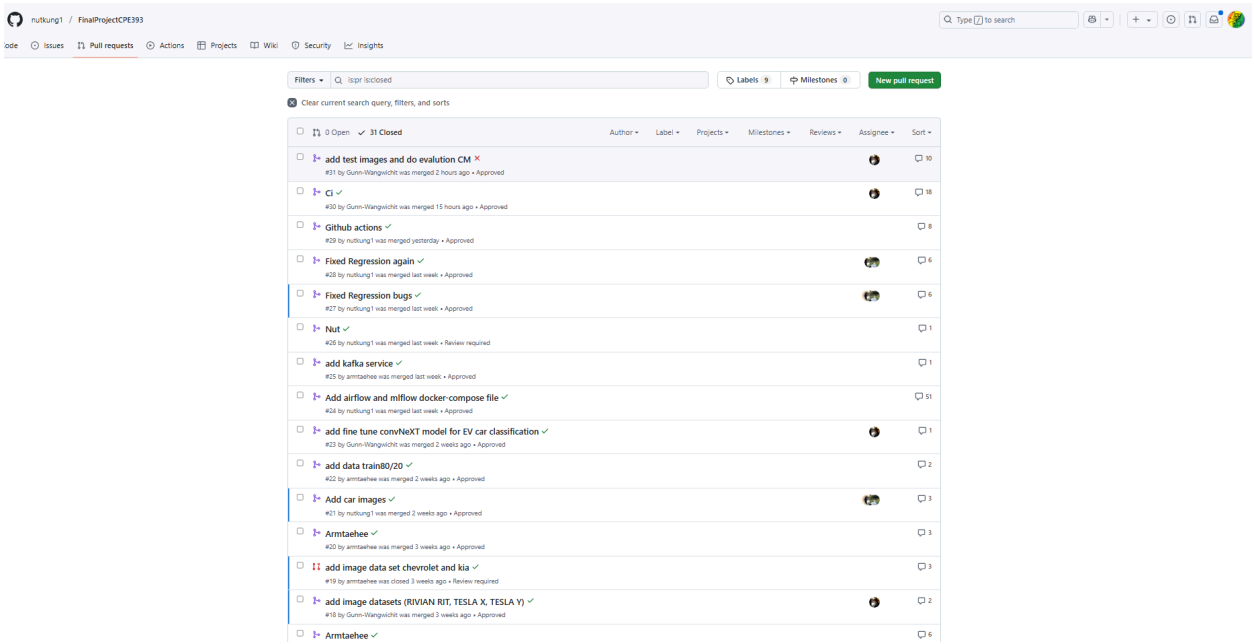
To ensure a collaborative and quality-controlled workflow, the project enforces branch protection rules. Specifically, direct pushes to the main branch are restricted. This means contributors must submit their changes via pull requests, which require review and approval by another team member before they can be merged. This setup adds a layer of validation, helping prevent bugs or untested code from being integrated into the main branch.

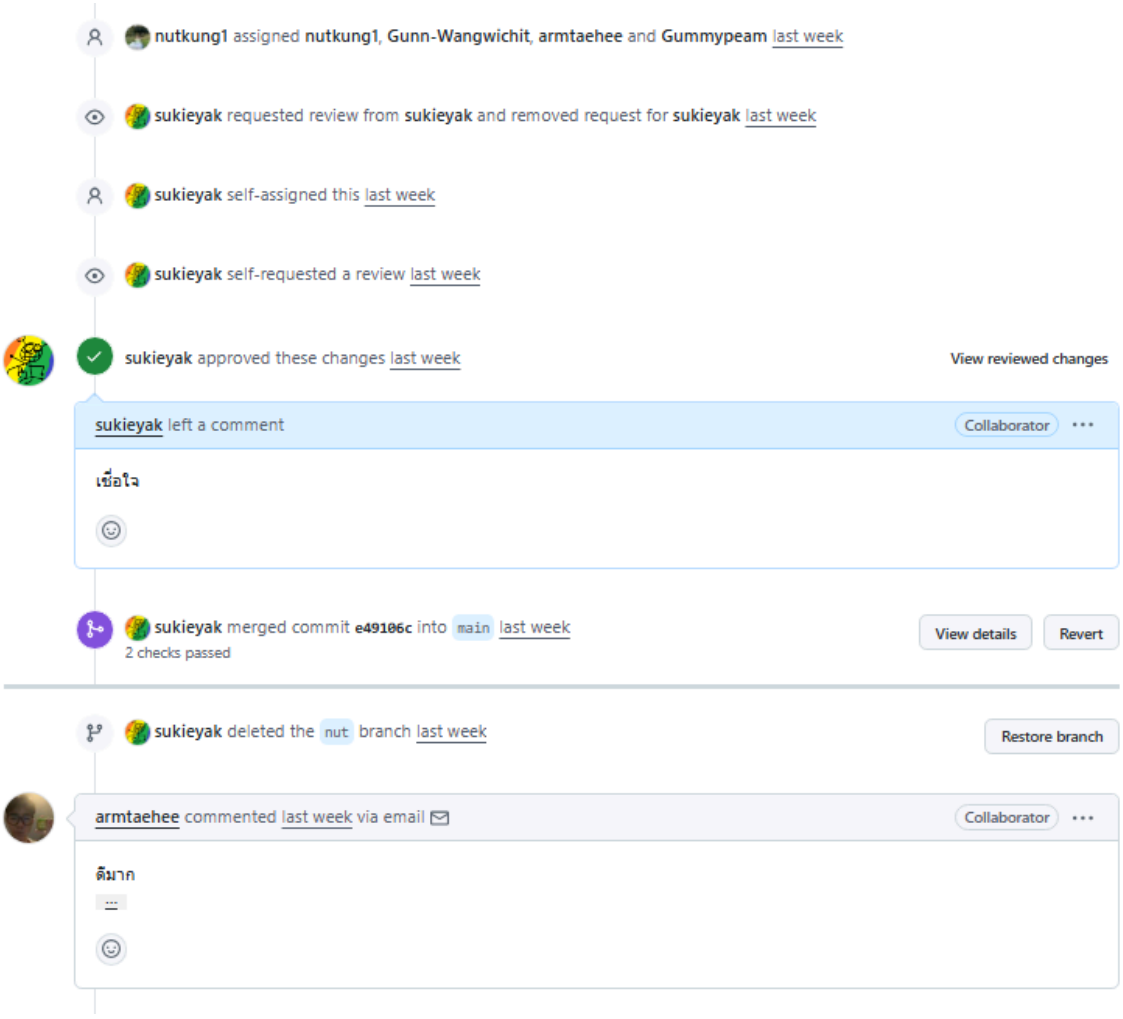
Key benefits of this version control strategy include:

1. Collaboration: Multiple contributors can work in parallel on features or experiments without conflicts, using isolated branches and pull requests.
2. Traceability: Every change is recorded with descriptive commit messages, enabling easy tracking and rollback if needed.
3. Experiment Management: Different versions of the model, datasets, or configurations can be managed across branches or tagged commits.
4. Continuous Integration (CI): The CI pipeline, implemented via GitHub Actions and CML, integrates seamlessly with version control. Each pull request triggers automated training and evaluation, with performance metrics posted directly in the PR for review.

Overall, this version control strategy ensures that the development process remains organized, transparent, and reliable, while enforcing high standards through peer review and automation.

Our Group Github: https://github.com/nutkung1/FinalProjectCPE393


nutkung1 / FinalProjectCPE393
Pull requests
Filters
Labels 9
Milestones 0
New pull request
Clear current search query, filters, and sorts
0 Open
31 Closed
Author
Label
Projects
Milestones
Reviews
Assignee
Sort
add test images and do evalution CM
#31 by Gunn-Wangwichit was merged 2 hours ago • Approved
Ci
#30 by Gunn-Wangwichit was merged 15 hours ago • Approved
Github actions
#29 by nutkung1 was merged yesterday • Approved
Fixed Regression again
#28 by nutkung1 was merged last week • Approved
Fixed Regression bugs
#27 by nutkung1 was merged last week • Approved
Nut
#26 by nutkung1 was merged last week • Review required
add kafka service
#25 by armtaehee was merged last week • Approved
Add airflow and mlflow docker-compose file
#24 by nutkung1 was merged last week • Approved
add fine tune convNeXT model for EV car classification
#23 by Gunn-Wangwichit was merged 2 weeks ago • Approved
add data train80/20
#22 by armtaehee was merged 2 weeks ago • Approved
Add car images
#21 by nutkung1 was merged 2 weeks ago • Approved
Armtaehee
#20 by armtaehee was merged 3 weeks ago • Approved
add image data set chevrolet and kia
#19 by armtaehee was closed 3 weeks ago • Review required
add image datasets (RIVIAN RIT, TESLA X, TESLA Y)
#18 by Gunn-Wangwichit was merged 3 weeks ago • Approved
Armtaehee


nutkung1 assigned nutkung1, Gunn-Wangwichit, armtaehee and Gummypeam last week
sukieyak requested review from sukieyak and removed request for sukieyak last week
sukieyak self-assigned this last week
sukieyak self-requested a review last week
sukieyak approved these changes last week
View reviewed changes
sukieyak left a comment
Collaborator
เชื่อใจ
sukieyak merged commit e49106c into main last week
2 checks passed
View details
Revert
sukieyak deleted the nut branch last week
Restore branch
armtaehee commented last week via email
Collaborator
ดีมาก

References

1. Dai, X., Liu, Z., Huang, D., Xia, Y., He, D., Yuan, L., & Guo, B. (2023). **ConvNeXt V2: Co-designing and Scaling ConvNets with Masked Autoencoders**. *arXiv preprint arXiv:2301.00808*. https://arxiv.org/abs/2301.00808
2. Facebook AI Research. (2023). *ConvNeXt V2 GitHub Repository*. GitHub. https://github.com/facebookresearch/ConvNeXt-V2
3. Hugging Face. (n.d.). *facebook/convnextv2-tiny-22k-224 – Hugging Face Model Card*. https://huggingface.co/facebook/convnextv2-tiny-22k-224
4. Hugging Face. (n.d.). *ConvNeXtV2 Documentation – Hugging Face Transformers*. https://huggingface.co/docs/transformers/main/en/model_doc/convnextv2
5. Noureddine Ridan. (2023). *Electric Vehicle Population Data* [Dataset]. Kaggle. https://www.kaggle.com/datasets/noureddineridanr96/electric-vehicle-population-data