Traffic Congestion Reduction and Management System

COMP 8047 – Major Project

Kartik Verma – A01022059

5-12-2024

Table of Contents

1. Introduction

1.1 Student Background

I, Kartik Verma am currently pursuing Bachelor of Technology in Computer Science at British Columbia Institute of Technology (BCIT). I have a strong academic background in Database Administration and Analysis, which has equipped me with the necessary knowledge and skills to embark on the project titled "Traffic Congestion Reduction and Management System."

1.2 Education

British Columbia Institute of Technology (BCIT), Burnaby, BC

Bachelor of Technology computer science (BTech) – Database Option (Graduating in 2024)

Langara College, Vancouver, BC January 2016 – May 2018

Diploma in Computer Studies

1.3 Project Description

The "Traffic Congestion Reduction and Management System" project is conceived as a dynamic and innovative solution to combat the persistent challenge of traffic congestion in urban areas. The project's core objective is to harness cutting-edge data collection, analysis, and management techniques to optimize traffic flow and alleviate congestion in Philadelphia, Pennsylvania. By proactively collecting historical and real-time data from diverse sources, including public and government websites and crowdsourced data from navigation apps such as Google Maps and Apple Maps, and leveraging state-of-the-art data analysis tools and predictive modeling, this project aims to provide real-time traffic monitoring, navigation guidance, and incident reporting through a user-friendly web-based dashboard for City Planners.

This project is both a feasible and innovative solution to address urban traffic congestion. With a strategic approach to time, cost, resource allocation, and with the requisite expertise, this project aspires to revolutionize urban traffic management and enhance the quality of life in our targeted urban area.

This project holds immense potential to benefit various stakeholders. *Urban Planning Departments* can leverage the system's data-driven insights to inform their city planning efforts, enabling more efficient allocation of resources for infrastructure development and traffic management. *Local Municipal Authorities* can enhance their ability to enforce traffic regulations and optimize road maintenance schedules, leading to improved road safety and reduced maintenance costs. For the *public*, the project promises shorter commute times, reduced stress, and improved overall mobility, enhancing the quality of urban life. *Businesses* stand to gain from increased customer accessibility, streamlined supply chain logistics, and reduced operational disruptions, ultimately fostering a more vibrant and prosperous local economy. In summary, this project offers a multifaceted approach to address urban congestion, benefiting urban planning, local authorities, the public, and businesses alike, thereby contributing to the holistic development of urban areas.

1.4 Problem Statement and Background

Traffic congestion is a multifaceted challenge with significant societal and economic implications. In 2019, traffic congestion in the United States incurred nearly $88 billion in costs for drivers, which encompassed wasted time and fuel expenses, according to data from the American Transportation Research Institute. Beyond the financial toll, congestion contributes to excessive carbon emissions and air pollution, negatively impacting public health and the environment. Addressing this issue goes beyond mere convenience; it's a critical step toward creating sustainable and livable urban environments.

The challenge of traffic congestion has persisted for decades, and it's exacerbated by rapid urbanization and the increasing rate of motorization seen worldwide. In 2018, the United Nations estimated that by 2050, approximately 68% of the world's population will reside in urban areas. In response to this urbanization wave, traditional traffic management systems have adapted by integrating static traffic signal timings and traffic cameras. However, these systems often lack the real-time adaptability required to effectively address the dynamic and unpredictable nature of congestion.

Traffic congestion is a multifaceted problem influenced by numerous factors, including road design, population density, time of day, accidents, and weather conditions. Current congestion management strategies primarily rely on historical data and predefined traffic signal timings. While these strategies can offer valuable insights, they fall short when it comes to addressing the immediate and evolving nature of congestion. A comprehensive solution necessitates the utilization of real-time data, predictive analytics, and user engagement to effectively manage congestion in urban environments.

Contemporary navigation applications, such as Google Maps, Apple Maps, and Waze, provide real-time traffic updates and suggest alternative routes based on user data. While these applications offer value for individual navigation, they predominantly serve as passive solutions, lacking proactive congestion management capabilities. Various municipalities employ Traffic Management Systems (TMS) aimed at optimizing traffic signals and monitoring congestion. However, these systems frequently depend on static signal timings and lack the ability to dynamically adapt to real-time traffic conditions.

2. Body

2.1 Background of the Project

To provide a comprehensive understanding of the report's subject matter, it's crucial to establish the background and context of the project. In this section, I aim to offer an objective overview of the factors that led to the inception of the project and the underlying challenges it seeks to address.

Nature of the Project: The project centers around the development of an integrated traffic management system that incorporates real-time weather data, and traffic insights such as total time duration, optimal departure time, and Public Transit data, etc. to enhance urban mobility and transportation efficiency. With the proliferation of urbanization and the increasing frequency of extreme weather events, the need for adaptive traffic management solutions affected by weather and or otherwise has become paramount.

Challenges in Urban Mobility: Urban centers worldwide face mounting challenges in managing traffic congestion, optimizing transportation networks, and ensuring the safety and convenience of commuters. These challenges are compounded by factors such as population growth, rapid urbanization, and the unpredictable nature of weather patterns.

Project Statement within the Context of Project Proposal: The genesis of the project stems from a comprehensive analysis of the existing traffic management landscape and the identification of gaps in addressing weather-induced disruptions. The project proposal outlined the overarching goal of developing an innovative solution that integrates real-time weather data with traffic management algorithms to optimize traffic flow and enhance resilience to adverse weather conditions.

2.2 Essential Problems Being Solved

In this section, I delve into the core problems and challenges that the project aims to tackle. By articulating these challenges concisely, I lay the foundation for understanding the significance and relevance of the proposed solution.

Traffic Congestion and Gridlock: One of the primary challenges facing urban mobility is the prevalence of traffic congestion and gridlock, particularly during peak hours and adverse weather conditions.

Congested roadways not only result in significant time delays and productivity losses but also pose safety hazards and environmental concerns.

Impact of Weather on Traffic Management: Weather variability, including rain, snow, fog, and extreme temperatures, exerts a significant influence on traffic patterns and transportation infrastructure. Adverse weather conditions can lead to reduced visibility, slippery road surfaces, and increased accident rates, further exacerbating traffic congestion and impeding the flow of vehicles.

Need for Adaptive Traffic Management: Conventional traffic management systems often struggle to adapt to dynamic weather conditions, relying on static algorithms and historical data for decision-making. There is a pressing need for adaptive traffic management solutions capable of integrating real-time weather information to optimize route planning, signal timing, and incident management.

Enhancing Resilience and Efficiency: By addressing the interplay between weather and traffic management, the project seeks to enhance the resilience and efficiency of urban transportation networks. By proactively anticipating weather-related disruptions and implementing adaptive strategies, the proposed solution aims to minimize congestion, improve travel times, and enhance overall mobility experiences for commuters.

2.3 Possible Alternative Solutions to the Project

In this section, I will explore alternative approaches and solutions that could potentially address the identified challenges. By considering multiple perspectives, this project will gain insights into the feasibility, advantages, and limitations of different strategies for tackling the problem at hand.

Alternative Approaches to Traffic Management: Numerous approaches exist for managing traffic congestion and improving urban mobility. These include traditional methods such as signal optimization, lane management, and public transportation enhancements. While these approaches have demonstrated effectiveness to some extent, they often lack the flexibility and adaptability needed to cope with dynamic weather conditions.

Weather-Informed Traffic Management Strategies: Another set of alternative solutions involves integrating weather data into traffic management systems. Some existing approaches utilize historical weather data to predict traffic patterns and adjust signal timings accordingly. However, these methods may not provide real-time insights or fail to account for the full range of weather-related impacts on traffic flow.

Emerging Technologies and Innovations: Advancements in technology, such as the Internet of Things (IoT), artificial intelligence (AI), and data analytics, have opened new possibilities for enhancing traffic management. Emerging solutions leverage sensor networks, predictive modeling, and cloud computing to deliver real-time traffic insights and adaptive control strategies. While promising, these technologies may require substantial investment and infrastructure upgrades.

Comparison of Alternative Solutions: Each alternative solution comes with its own set of advantages, challenges, and trade-offs. A comparative analysis is essential to evaluate the effectiveness, scalability, and cost-effectiveness of different approaches. Factors such as data accuracy, computational complexity, and user acceptance play a critical role in determining the suitability of each solution for the project's objectives.

Rationale for the Selected Approach: After careful consideration and evaluation of alternative solutions, I opted for an integrated approach that combines real-time weather data with advanced traffic management algorithms to provide Predictions based on Historical data, and real-time insights on the go at one place with very interactive and human friendly UI Dashboard App. This decision was informed by the need for a proactive, adaptive solution capable of addressing the dynamic nature of urban mobility challenges, particularly in the face of changing weather conditions.

2.4 Solution Chosen for this Project and Rationale Behind

In this section, I delve into the selected solution for addressing the project's objectives and provide insights into why this particular approach was chosen over alternative options.

Selected Solution: Real-Time Weather-Informed Traffic Management System for the City of Philadelphia

The project aims to develop a sophisticated traffic management system that optimizes traffic flow in urban areas, particularly during adverse weather conditions. This solution integrates advanced data analytics, machine learning algorithms, and IoT sensors. Here's how it works:

The system combines real-time weather data from the OpenWeatherMap API with traffic flow data obtained from the Google Maps API. By analyzing historical patterns, the system identifies correlations between weather conditions and traffic congestion. Machine learning models, including Linear Regression and Random Forest, predict traffic patterns based on factors such as time of day, weather, road conditions, and historical traffic data. These models inform real-time adjustments, such as rerouting vehicles or optimizing traffic signals. Additionally, strategically placed IoT sensors collect data on traffic volume, vehicle speeds, and road conditions. The system provides a user-friendly Flask-based web dashboard where users input their location, weather conditions, and desired destination. It then offers real-time traffic predictions and suggests alternate routes. Ultimately, this solution contributes to more efficient and safer urban transportation by addressing congestion challenges.

Rationale Behind the Selected Solution:

1. Proactive Traffic Management: By incorporating real-time weather data into the traffic management system, the proposed solution enables proactive decision-making and adaptive control strategies to mitigate the adverse effects of weather on traffic flow. This proactive approach helps anticipate and respond to changing conditions in advance, reducing the likelihood of congestion and accidents.

2. Enhanced Accuracy and Precision: Unlike traditional traffic management methods that rely solely on historical data or predefined schedules, the real-time weather-informed system provides accurate and precise insights into current weather conditions and their impact on road surfaces, visibility, and driver behavior. This granular level of information allows for more informed and effective traffic management decisions.

3. Dynamic Optimization Algorithms: The system employs advanced optimization algorithms and machine learning techniques to analyze the complex interactions between weather patterns, traffic flow dynamics, and road infrastructure. By continuously adapting and optimizing traffic signal timings, lane assignments, and route guidance based on real-time data inputs, the solution ensures efficient and equitable distribution of traffic resources across the road network.

4. Comprehensive Solution Coverage: This solution encompasses various complexities, including:
   a. Utilization of web framework Flask to develop a user-friendly dashboard for real-time monitoring and control.
   b. Integration of real-time data insights and APIs to capture and process live traffic and weather data streams.
   c. Incorporation of historical data processing and predictive analysis to forecast traffic patterns and optimize resource allocation.
   d. Utilization of Tableau for data visualization, enabling interactive and intuitive visualizations for better user engagement and decision-making.

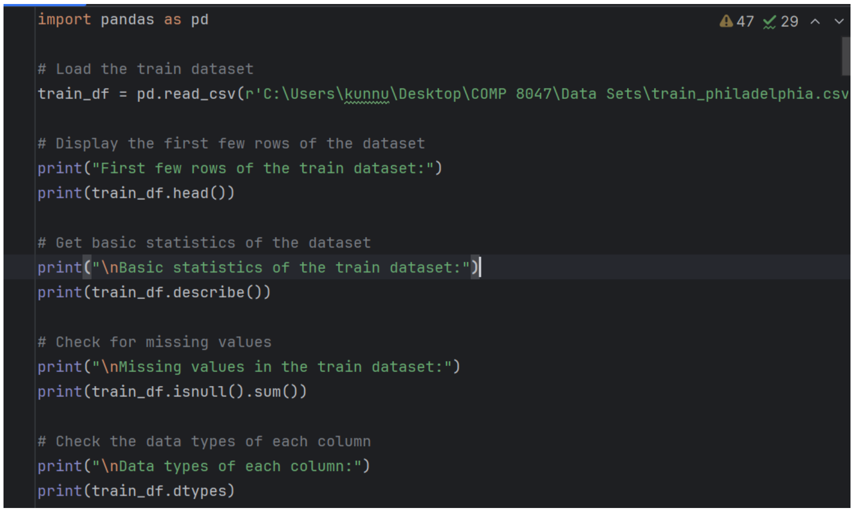
5. Scalability and Flexibility: Designed with scalability and flexibility in mind, the proposed solution can be easily customized and deployed in diverse urban environments with varying traffic patterns, infrastructure configurations, and weather conditions. Modular architecture and cloud-based infrastructure facilitate seamless integration with existing traffic management systems and future upgrades.

2.5 Design and Development

In this project, I performed both historical data analysis and real-time insights to gain a comprehensive understanding of traffic patterns and facilitate optimal route planning for commuters, and City Planners.

Data Analysis:

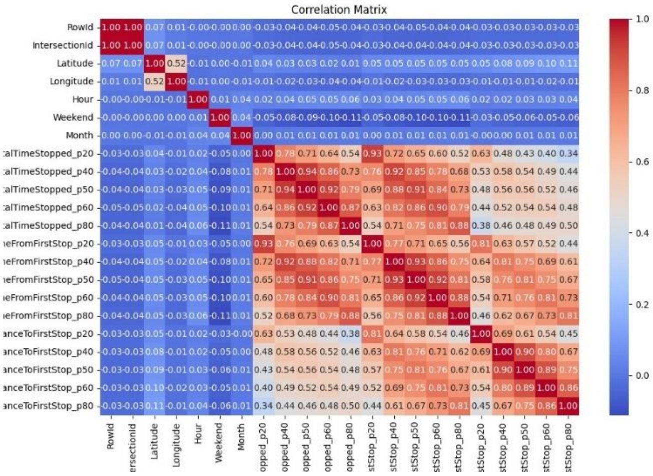
This project starts by analyzing historical traffic and weather data using Python as separate data files (later merged into one) from Kaggle. Key steps in this process include:
- Loading historical data from CSV files using the pandas library.



```python
import pandas as pd

# Load the train dataset
train_df = pd.read_csv(r'C:\Users\kunnu\Desktop\COMP 8047\Data Sets\train_philadelphia.csv

# Display the first few rows of the dataset
print("First few rows of the train dataset:")
print(train_df.head())

# Get basic statistics of the dataset
print("\nBasic statistics of the train dataset:")
print(train_df.describe())

# Check for missing values
print("\nMissing values in the train dataset:")
print(train_df.isnull().sum())

# Check the data types of each column
print("\nData types of each column:")
print(train_df.dtypes)
```

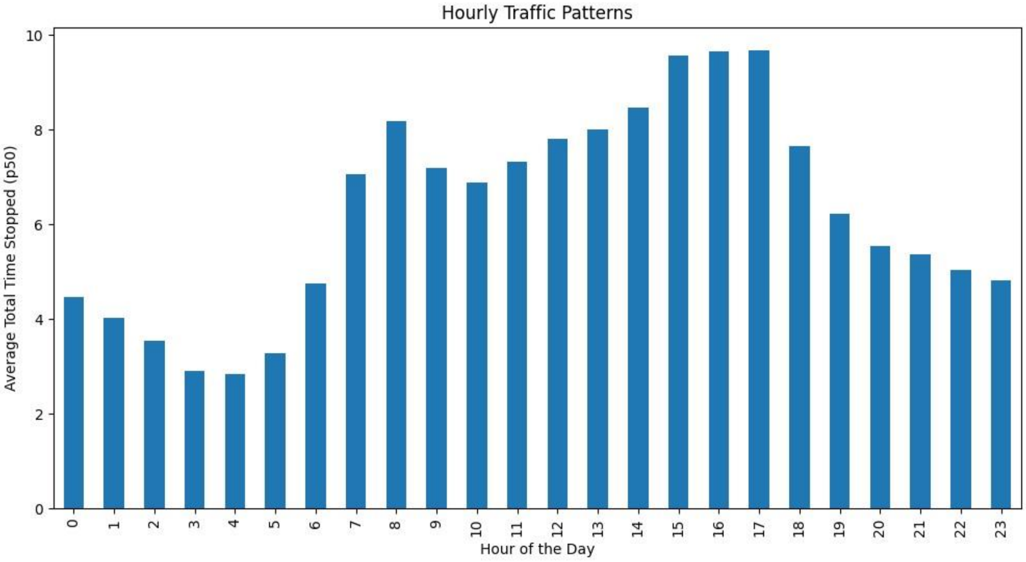
pd.read_csv("file_location") is used to read the dataset and load into the system

- Conducting basic statistics, correlation analysis, and feature distribution analysis to understand the data



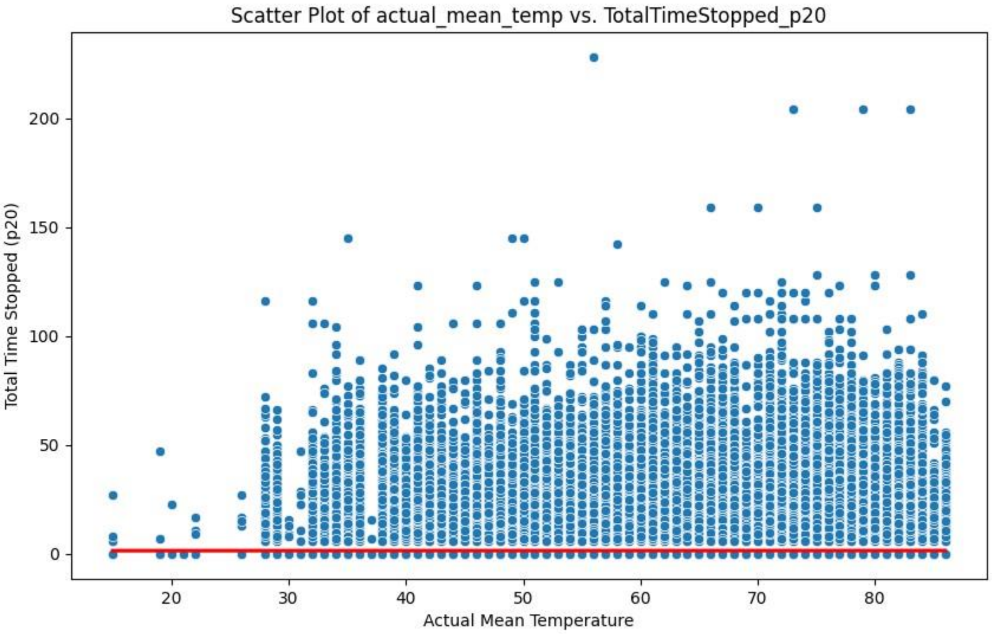
Correlation analysis done and graph created to conduct analysis of the relationship between rows and cols of the dataset.

- Visualizing traffic patterns over time and space through temporal analysis, geospatial analysis, and route analysis.

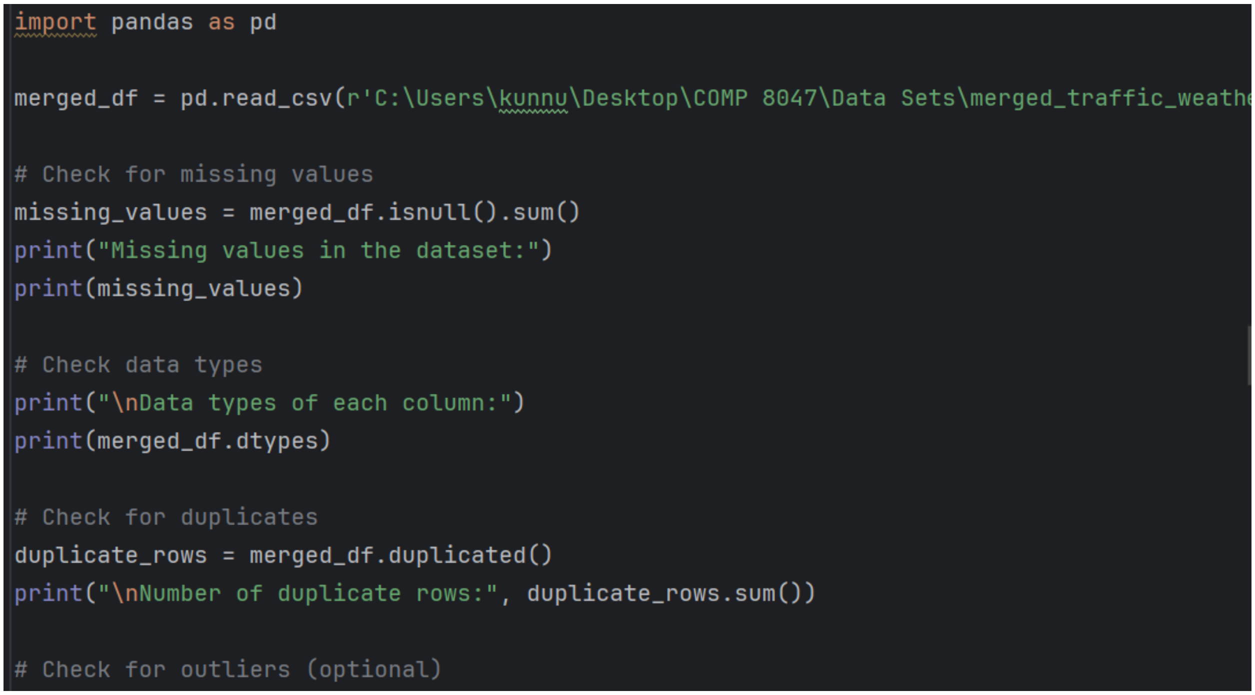


The following graph above shows average traffic on specific hour of the day showing 2-5PM is the busiest time of the day.

- Assessing the impact of weather conditions on traffic metrics by integrating weather data and analyzing correlations.



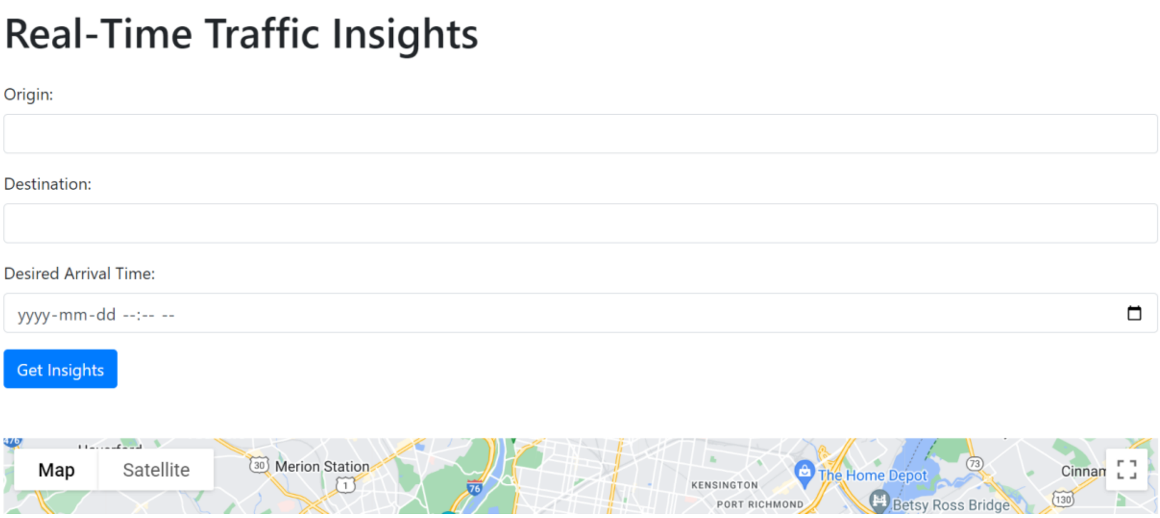
The following scatter plot above shows average traffic and Actual Mean Temperature (in F)
- Cleaning the data by handling missing values, data types, and consistency issues.



```
import pandas as pd

merged_df = pd.read_csv(r'C:\Users\kunnu\Desktop\COMP 8047\Data Sets\merged_traffic_weathe

# Check for missing values
missing_values = merged_df.isnull().sum()
print("Missing values in the dataset:")
print(missing_values)

# Check data types
print("\nData types of each column:")
print(merged_df.dtypes)

# Check for duplicates
duplicate_rows = merged_df.duplicated()
print("\nNumber of duplicate rows:", duplicate_rows.sum())

# Check for outliers (optional)
```

The code snippet above shows that the data is first checked for missing values and duplicates to access and clean later for visualization purposes.

Real-time Insights:
This Project also provide real-time insights to users through a web application powered by Flask and APIs. This functionality includes:



Real-Time Traffic Insights

Origin:

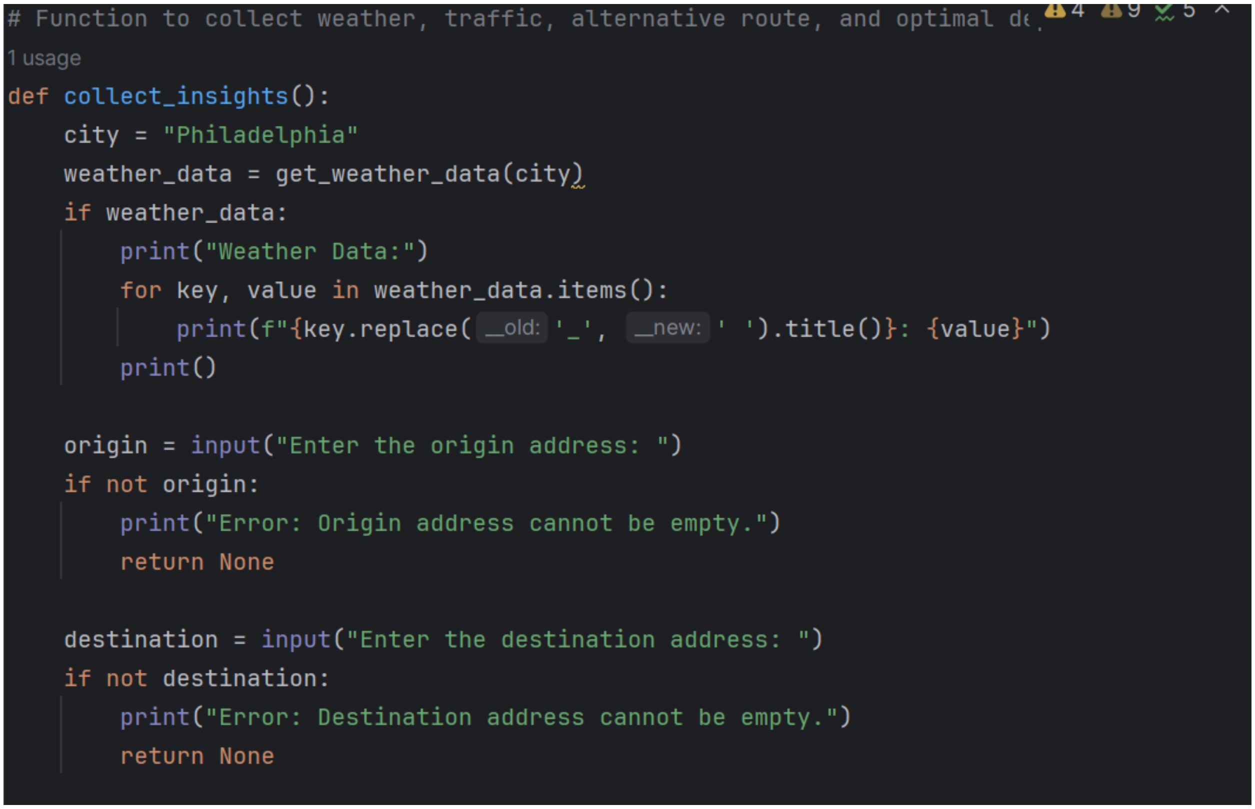
Destination:

Desired Arrival Time:

yyyy-mm-dd --:-- --

- Fetching real-time weather data based on the user's location using the OpenWeatherMap API.

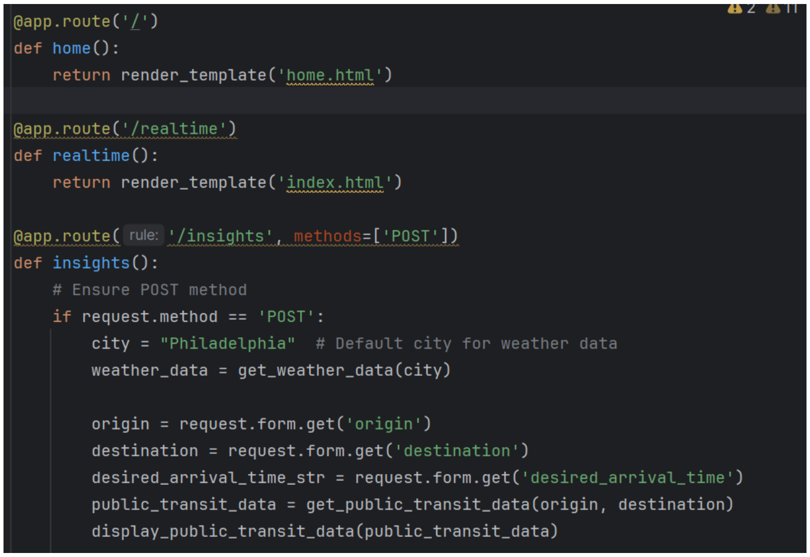
- Retrieving real-time traffic data, including duration and congestion levels between two locations, via the Google Maps API.
- Suggesting an optimal departure time for reaching a destination by a desired arrival time.
- Providing alternative routes and traffic incidents near the user's location for route planning.



```
# Function to collect weather, traffic, alternative route, and optimal de
1 usage
def collect_insights():
    city = "Philadelphia"
    weather_data = get_weather_data(city)
    if weather_data:
        print("Weather Data:")
        for key, value in weather_data.items():
            print(f"{key.replace( _old: '_',  _new: ' ').title()}: {value}")
        print()

    origin = input("Enter the origin address: ")
    if not origin:
        print("Error: Origin address cannot be empty.")
        return None

    destination = input("Enter the destination address: ")
    if not destination:
        print("Error: Destination address cannot be empty.")
        return None
```

The following code snippet above shows that the user inputs are taken to collect the real-time traffic and weather data and insights.

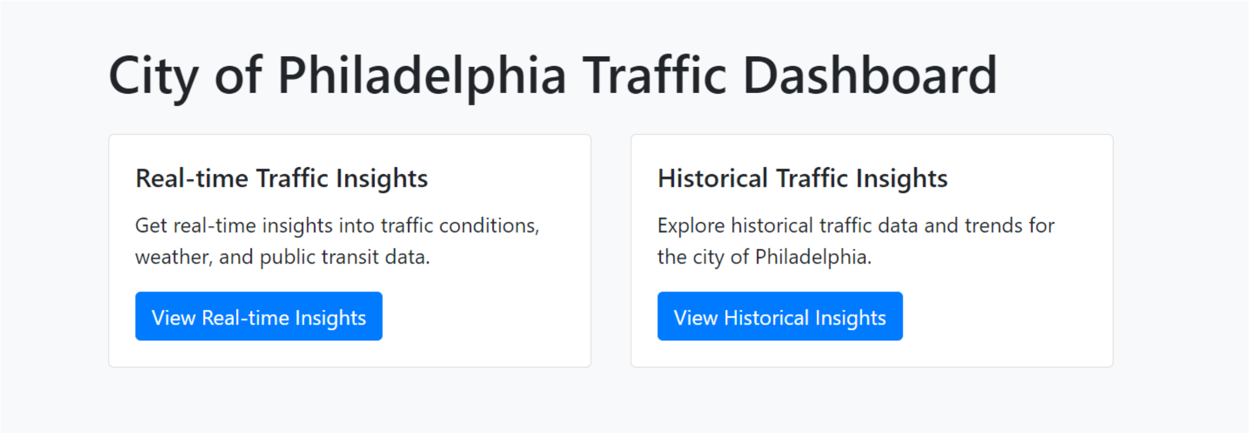
- Fetching and displaying real-time public transit data, enabling users to explore transit options.
- The Flask web application handles user requests, processes data using APIs, and returns JSON responses containing actionable insights.



```
@app.route('/')
def home():
    return render_template('home.html')

@app.route('/realtime')
def realtime():
    return render_template('index.html')

@app.route( rule: '/insights', methods=['POST'])
def insights():
    # Ensure POST method
    if request.method == 'POST':
        city = "Philadelphia"  # Default city for weather data
        weather_data = get_weather_data(city)

        origin = request.form.get('origin')
        destination = request.form.get('destination')
        desired_arrival_time_str = request.form.get('desired_arrival_time')
        public_transit_data = get_public_transit_data(origin, destination)
        display_public_transit_data(public_transit_data)
```

The code snippet above shows 3 pages of the flask app and how they are rendered to produce the flask app dashboard.

By combining historical data analysis with real-time insights, our project aims to empower users with the information needed to make informed decisions and optimize their commuting experience.



City of Philadelphia Traffic Dashboard

Real-time Traffic Insights

Get real-time insights into traffic conditions, weather, and public transit data.

Historical Traffic Insights

Explore historical traffic data and trends for the city of Philadelphia.

The screenshot above shows that the Dashboard caters to both side of the data insights, both Real-time and Historical.

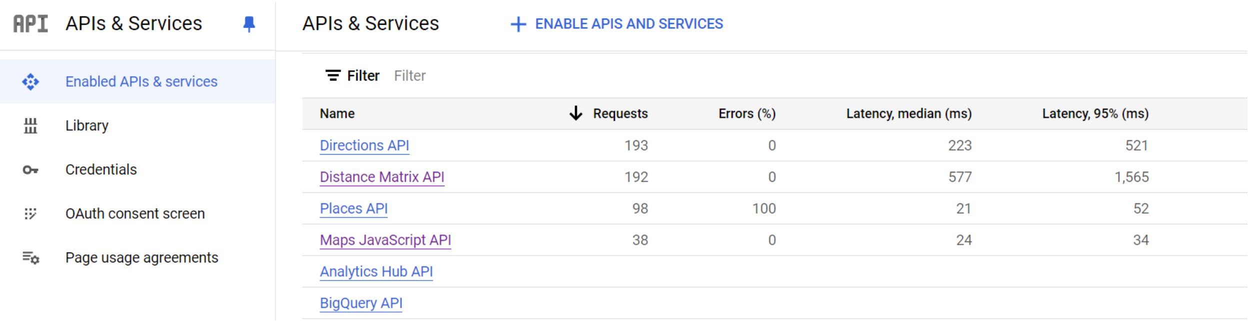
Real-Time Insights and API Integration

Real-Time Traffic and Weather Insights

In the context of traffic management and optimization, real-time insights play a crucial role in understanding current traffic conditions and making informed decisions. This section focuses on integrating APIs to gather real-time traffic and weather data, providing users with valuable insights for route planning and decision-making.

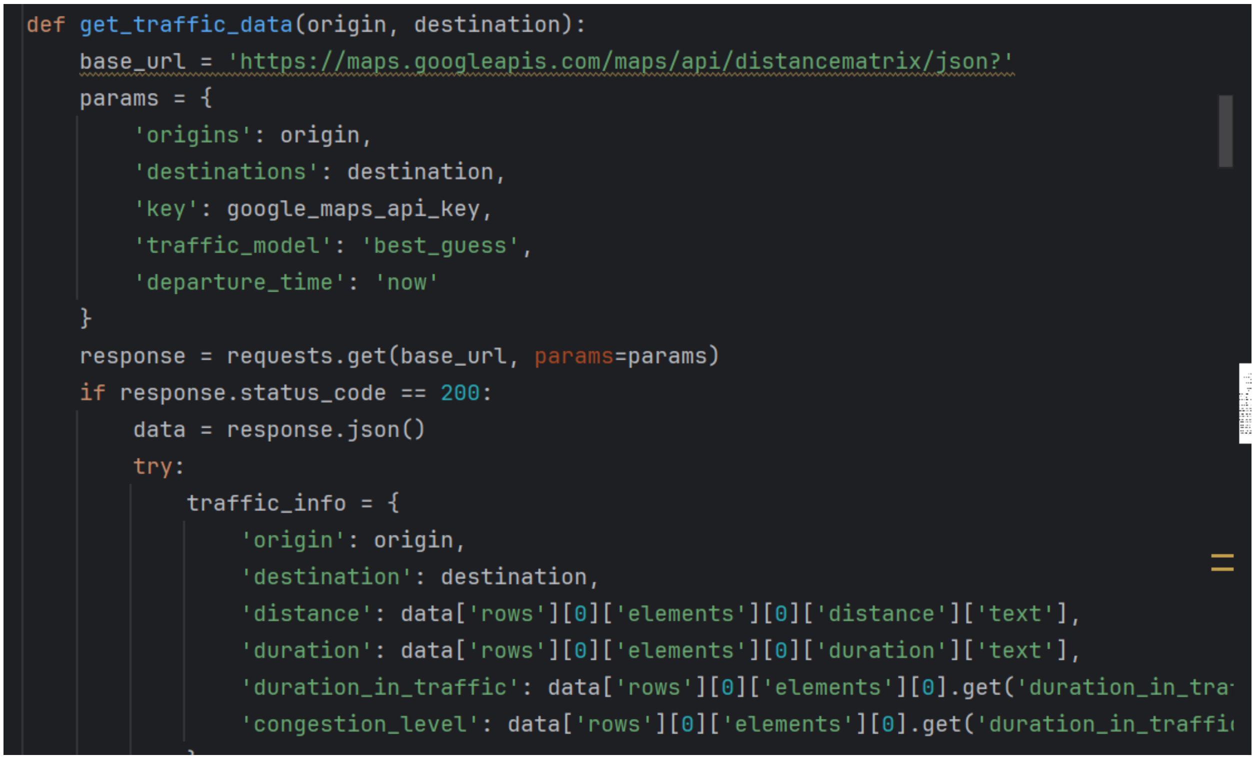
Integration of Google Maps API for Traffic Data

The first step involves integrating the Google Maps API to retrieve real-time traffic data. By leveraging this API, we can obtain information about travel times, congestion levels, and alternative routes between specified locations.



Name	Requests	Errors (%)	Latency, median (ms)	Latency, 95% (ms)
Directions API	193	0	223	521
Distance Matrix API	192	0	577	1,565
Places API	98	100	21	52
Maps JavaScript API	38	0	24	34
Analytics Hub API				
BigQuery API				

The screenshot above shows the APIs and services Google Maps is working on to collect and deliver different traffic insights such as Directions, Public Transit Data etc.



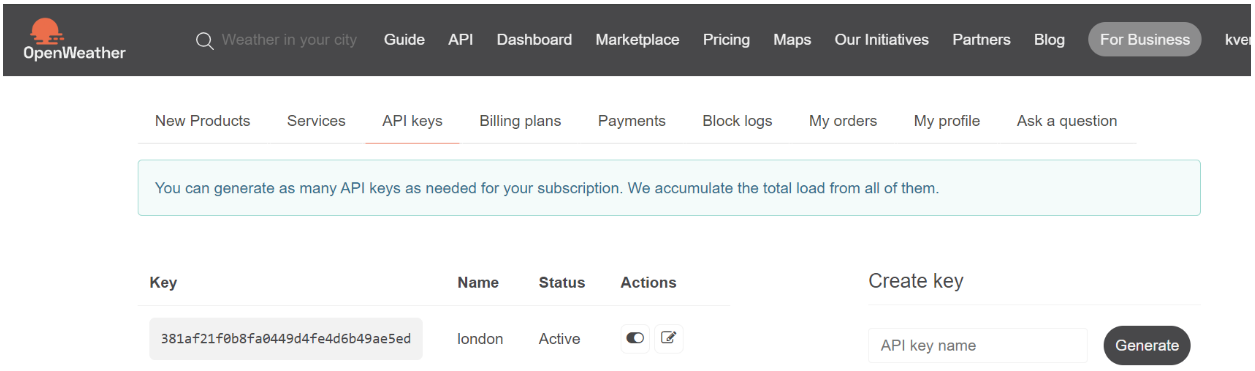
```
def get_traffic_data(origin, destination):
    base_url = 'https://maps.googleapis.com/maps/api/distancematrix/json?'
    params = {
        'origins': origin,
        'destinations': destination,
        'key': google_maps_api_key,
        'traffic_model': 'best_guess',
        'departure_time': 'now'
    }
    response = requests.get(base_url, params=params)
    if response.status_code == 200:
        data = response.json()
        try:
            traffic_info = {
                'origin': origin,
                'destination': destination,
                'distance': data['rows'][0]['elements'][0]['distance']['text'],
                'duration': data['rows'][0]['elements'][0]['duration']['text'],
                'duration_in_traffic': data['rows'][0]['elements'][0].get('duration_in_tra
                'congestion_level': data['rows'][0]['elements'][0].get('duration_in_traffi
```

The code snippet above shows how the requests from Google are collected and then transformed into human readable format for the Web App

This function sends a request to the Google Maps API with the origin and destination addresses, along with parameters for real-time traffic information. It retrieves details such as distance, duration, and congestion level.

Integration of OpenWeatherMap API for Weather Data:

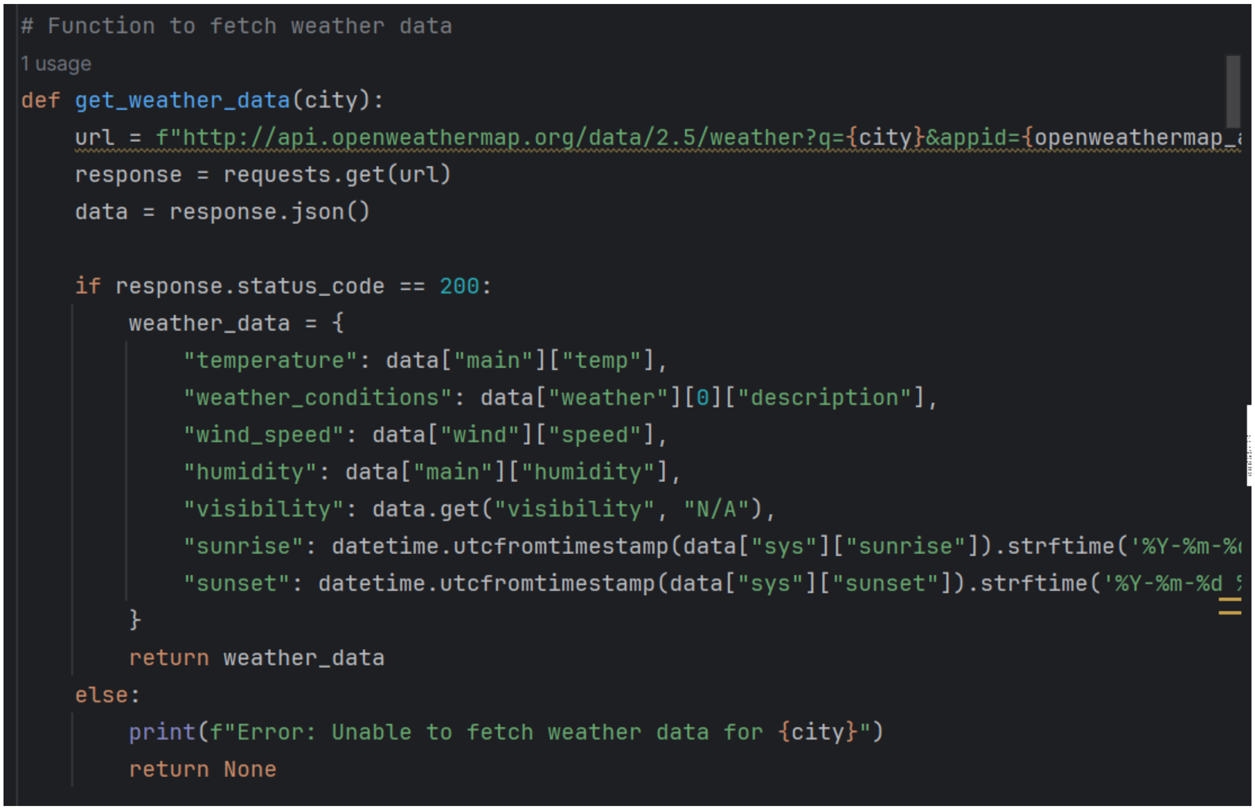
In addition to traffic data, weather conditions significantly impact road conditions and traffic flow. Integrating the OpenWeatherMap API allows us to obtain real-time weather information for a specified location.



You can generate as many API keys as needed for your subscription. We accumulate the total load from all of them.

Key	Name	Status	Actions
381af21f0b8fa0449d4fe4d6b49ae5ed	london	Active	

The screenshot above shows the API from OpenWeatherMap is active.



```
# Function to fetch weather data
1 usage
def get_weather_data(city):
    url = f"http://api.openweathermap.org/data/2.5/weather?q={city}&appid={openweathermap_
    response = requests.get(url)
    data = response.json()

    if response.status_code == 200:
        weather_data = {
            "temperature": data["main"]["temp"],
            "weather_conditions": data["weather"][0]["description"],
            "wind_speed": data["wind"]["speed"],
            "humidity": data["main"]["humidity"],
            "visibility": data.get("visibility", "N/A"),
            "sunrise": datetime.utcfromtimestamp(data["sys"]["sunrise"]).strftime('%Y-%m-%
            "sunset": datetime.utcfromtimestamp(data["sys"]["sunset"]).strftime('%Y-%m-%d 
        }
        return weather_data
    else:
        print(f"Error: Unable to fetch weather data for {city}")
        return None
```

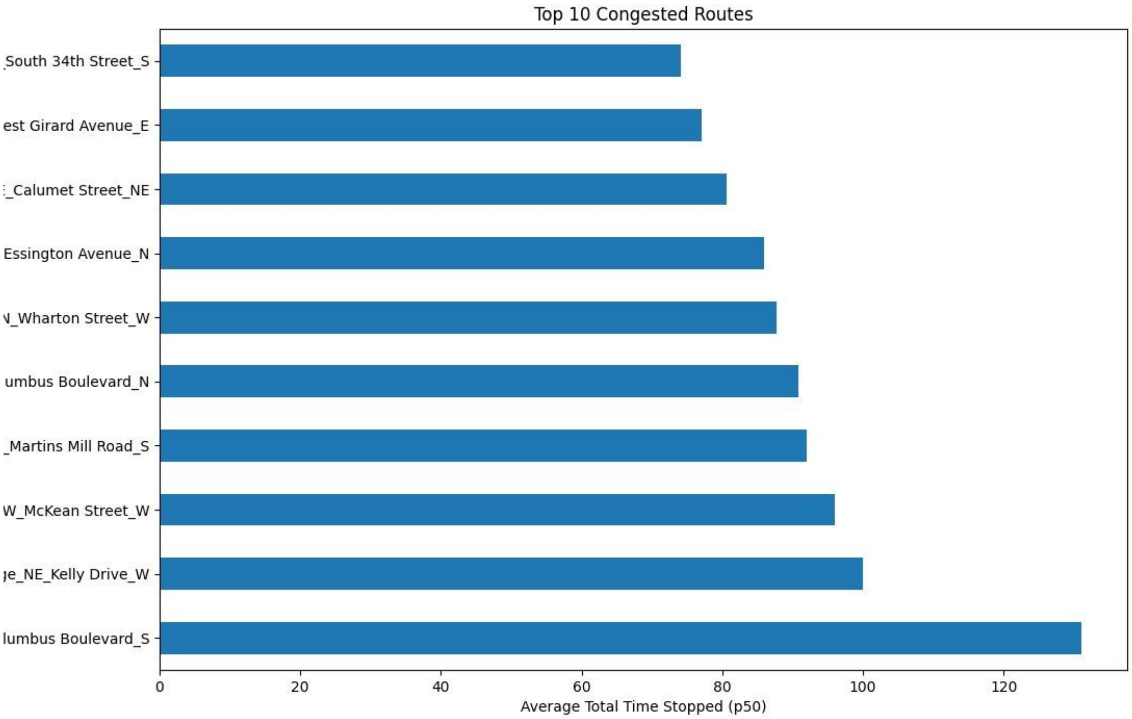
This function shown in the code snippet above sends a request to the OpenWeatherMap API with the specified city and retrieves current weather conditions such as temperature, weather description, wind speed, humidity, visibility, sunrise, and sunset times.

Insights Generation and Visualization

Once the real-time traffic and weather data are fetched, insights can be generated to assist users in making informed decisions. Visualization techniques such as charts and maps can enhance the presentation of these insights.

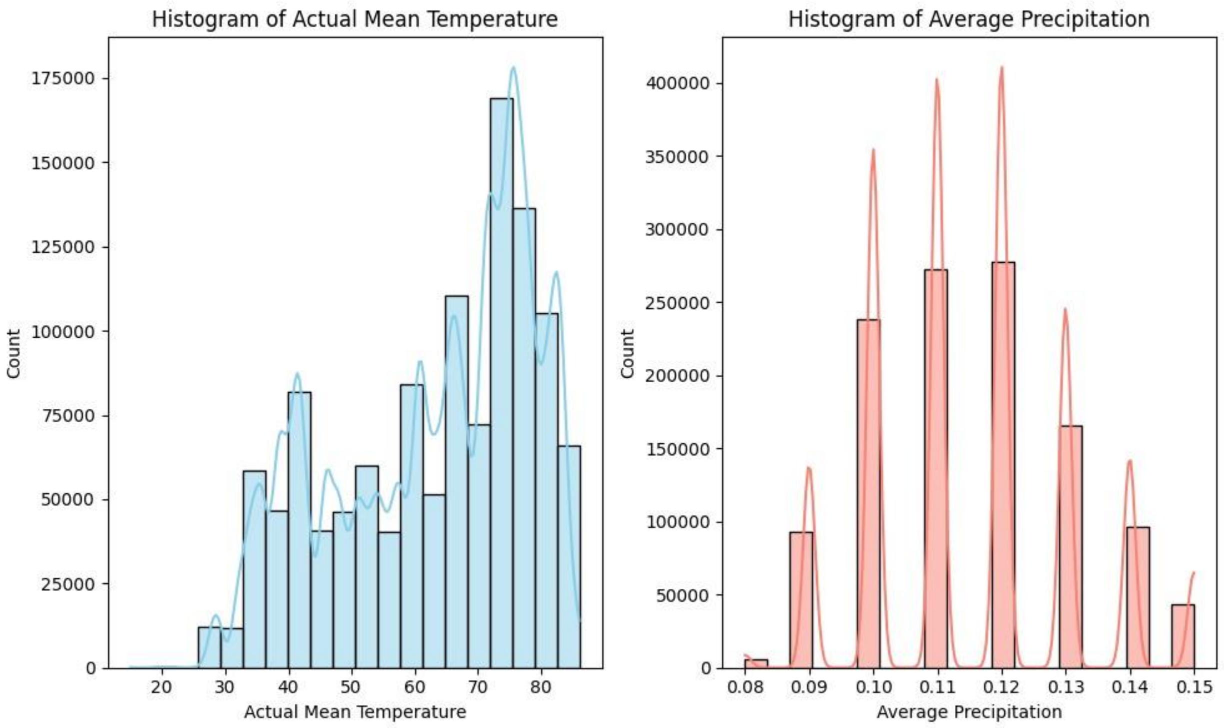
*Charts and Visualizations: *

1. **Traffic Congestion Trends: ** A line chart depicting the variation of traffic congestion levels over time can provide valuable insights into peak hours and congestion patterns. This chart helps users plan their travel routes more effectively.

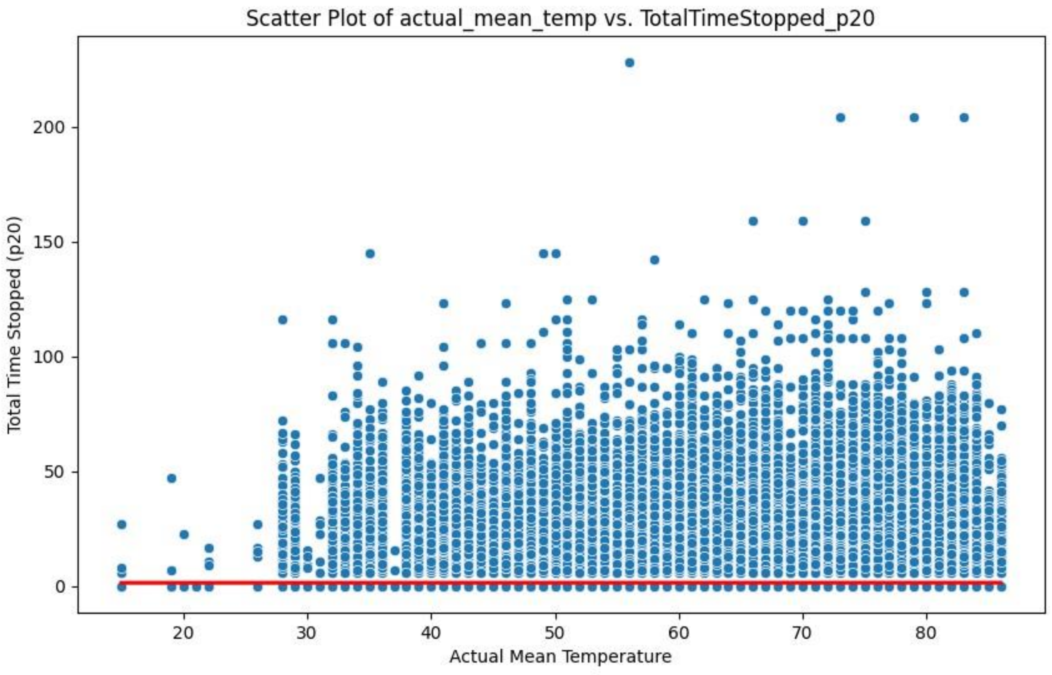


The graph in the above screenshot shows the top 10 congested Routes with their names and how much average time is stopped.

2. **Weather Impact Analysis: ** A scatter plot illustrating the correlation between weather conditions (such as temperature, precipitation) and traffic congestion levels can highlight the influence of weather on traffic flow. This visualization aids in understanding how weather fluctuations affect road conditions.

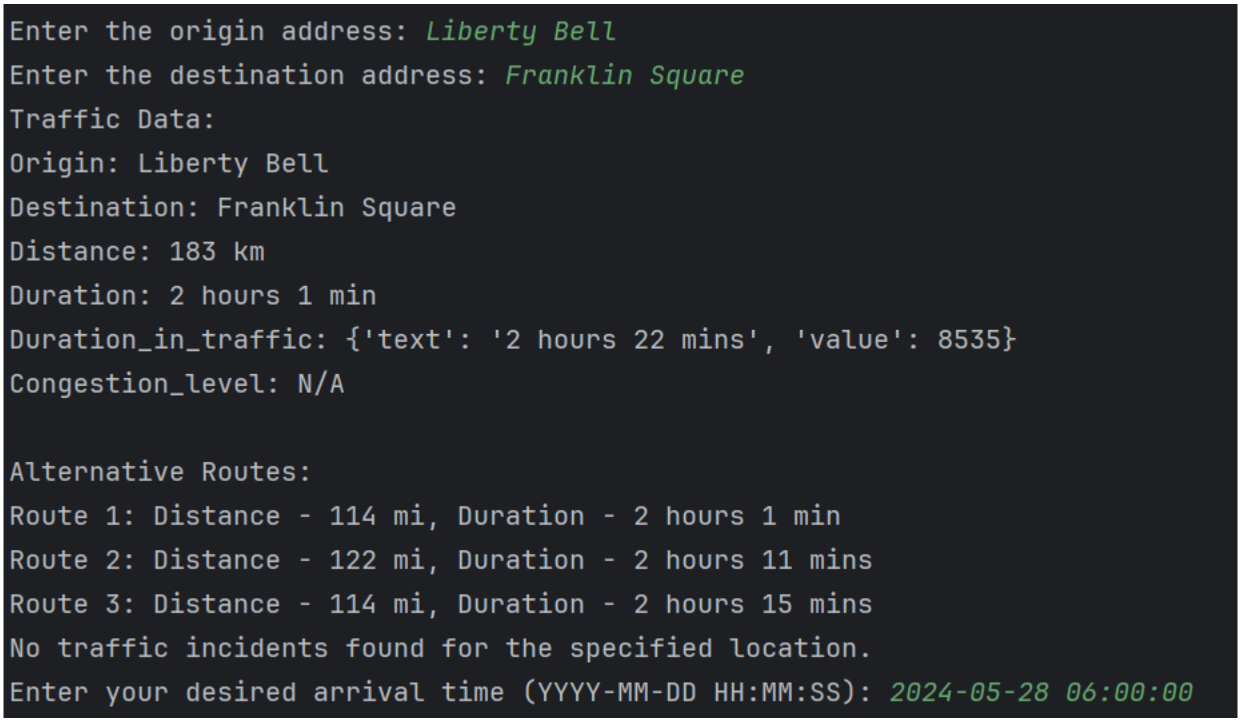


The screenshot above shows the weather data and its fluctuations with Mean and Average Precipitation as factors.

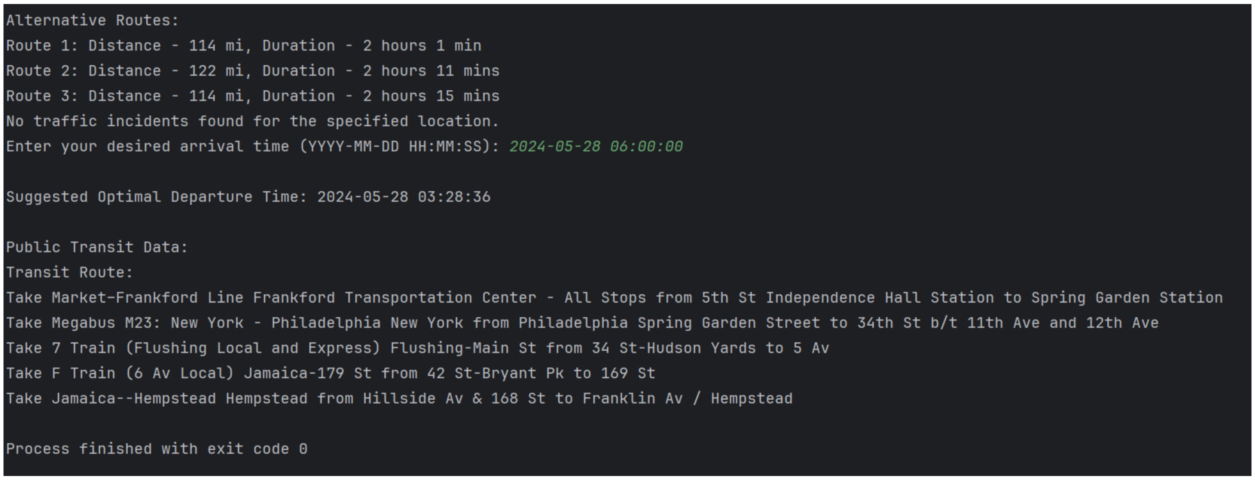


The scatter plot above shows the Traffic or total time stopped affected by the Actual Mean temperature.

3. **Route Optimization: ** A detailed overlay showing alternative routes with varying travel times and congestion levels can assist users in selecting the most efficient route based on real-time traffic data. This result enables users to explore different route options visually.



```
Enter the origin address: Liberty Bell
Enter the destination address: Franklin Square
Traffic Data:
Origin: Liberty Bell
Destination: Franklin Square
Distance: 183 km
Duration: 2 hours 1 min
Duration_in_traffic: {'text': '2 hours 22 mins', 'value': 8535}
Congestion_level: N/A

Alternative Routes:
Route 1: Distance - 114 mi, Duration - 2 hours 1 min
Route 2: Distance - 122 mi, Duration - 2 hours 11 mins
Route 3: Distance - 114 mi, Duration - 2 hours 15 mins
No traffic incidents found for the specified location.
Enter your desired arrival time (YYYY-MM-DD HH:MM:SS): 2024-05-28 06:00:00
```



```
Alternative Routes:
Route 1: Distance - 114 mi, Duration - 2 hours 1 min
Route 2: Distance - 122 mi, Duration - 2 hours 11 mins
Route 3: Distance - 114 mi, Duration - 2 hours 15 mins
No traffic incidents found for the specified location.
Enter your desired arrival time (YYYY-MM-DD HH:MM:SS): 2024-05-28 06:00:00

Suggested Optimal Departure Time: 2024-05-28 03:28:36

Public Transit Data:
Transit Route:
Take Market-Frankford Line Frankford Transportation Center - All Stops from 5th St Independence Hall Station to Spring Garden Station
Take Megabus M23: New York - Philadelphia New York from Philadelphia Spring Garden Street to 34th St b/t 11th Ave and 12th Ave
Take 7 Train (Flushing Local and Express) Flushing-Main St from 34 St-Hudson Yards to 5 Av
Take F Train (6 Av Local) Jamaica-179 St from 42 St-Bryant Pk to 169 St
Take Jamaica--Hempstead Hempstead from Hillside Av & 168 St to Franklin Av / Hempstead

Process finished with exit code 0
```

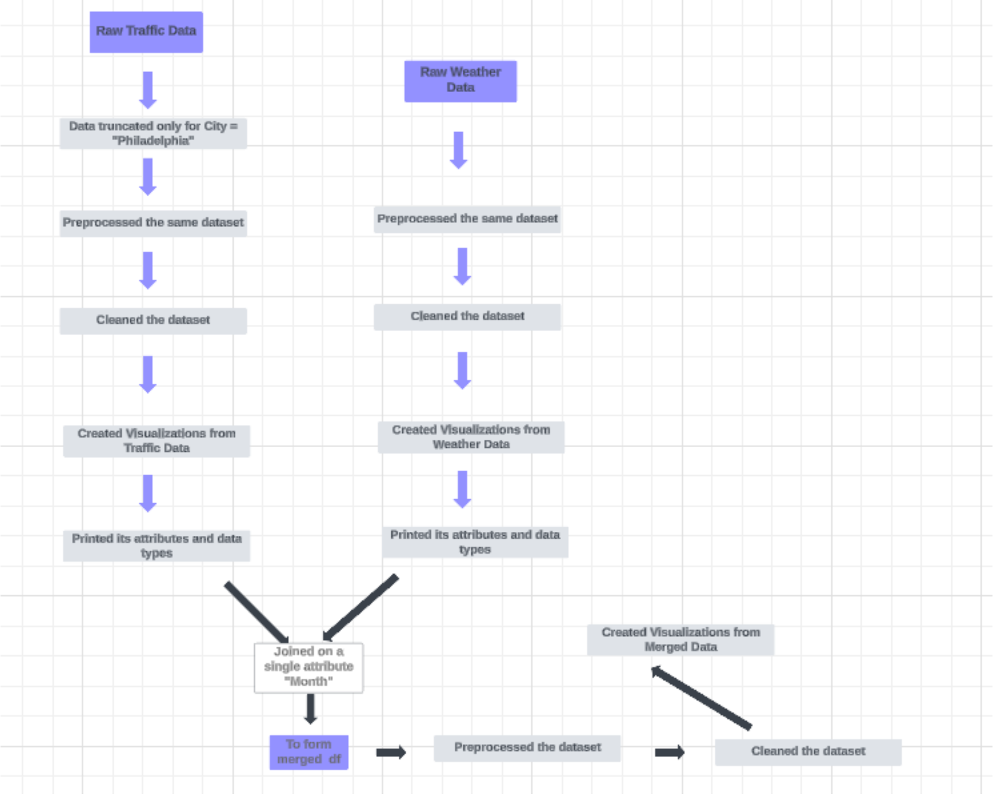
The screenshot of the terminal above shows that the system collects user inputs and provide them with Suggested Optimal Departure time, and Public Transit data so users can plan their trip better.

The integration of real-time traffic and weather APIs provides valuable insights for optimizing travel routes and enhancing traffic management strategies. By leveraging these insights and visualizations, users can make informed decisions to navigate traffic more efficiently and mitigate the impact of adverse weather conditions. This real-time approach to traffic and weather analysis contributes to safer and more efficient transportation systems.

2.6 System/Software Architecture Diagram

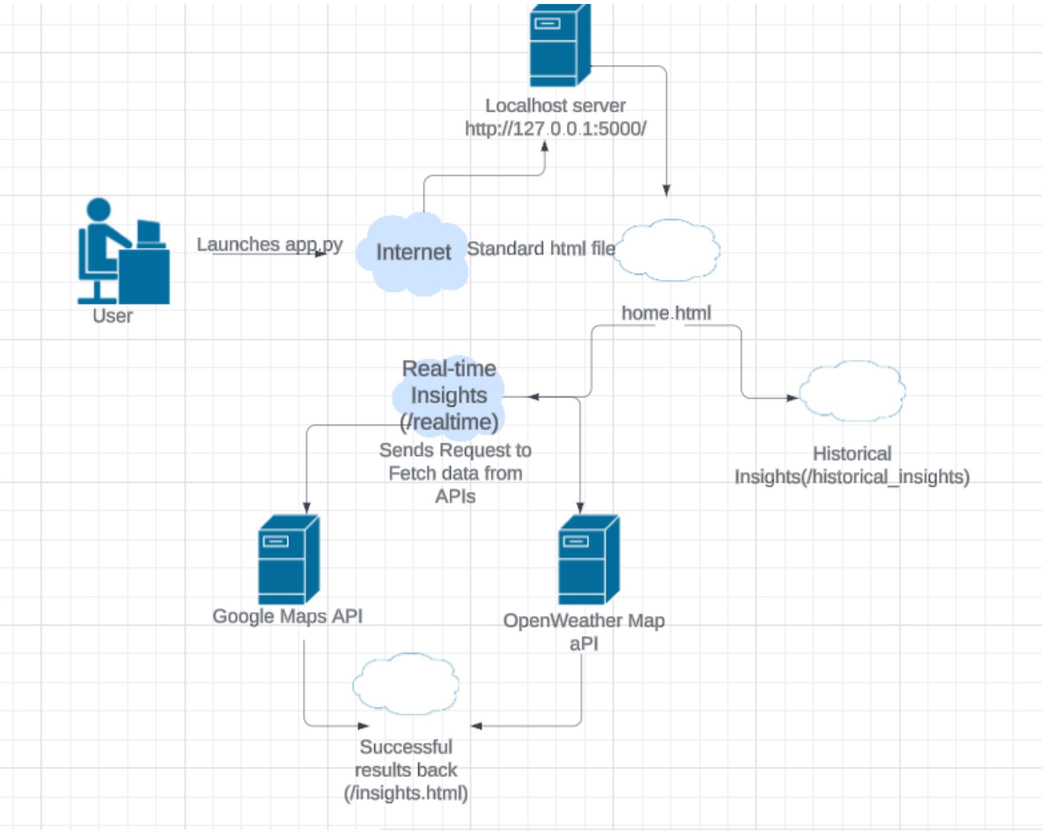
Flowchart Sequence Diagram:

Flowchart Sequence Diagram on how the Processes were completed starting from Data Collection, Preprocessing, cleaning, and Visualization.


Raw Traffic Data
Raw Weather Data
Data truncated only for City = "Philadelphia"
Preprocessed the same dataset
Preprocessed the same dataset
Cleaned the dataset
Cleaned the dataset
Created Visualizations from Traffic Data
Created Visualizations from Weather Data
Printed its attributes and data types
Printed its attributes and data types
Created Visualizations from Merged Data
Joined on a single attribute "Month"
To form merged df
Preprocessed the dataset
Cleaned the dataset

Network Diagram

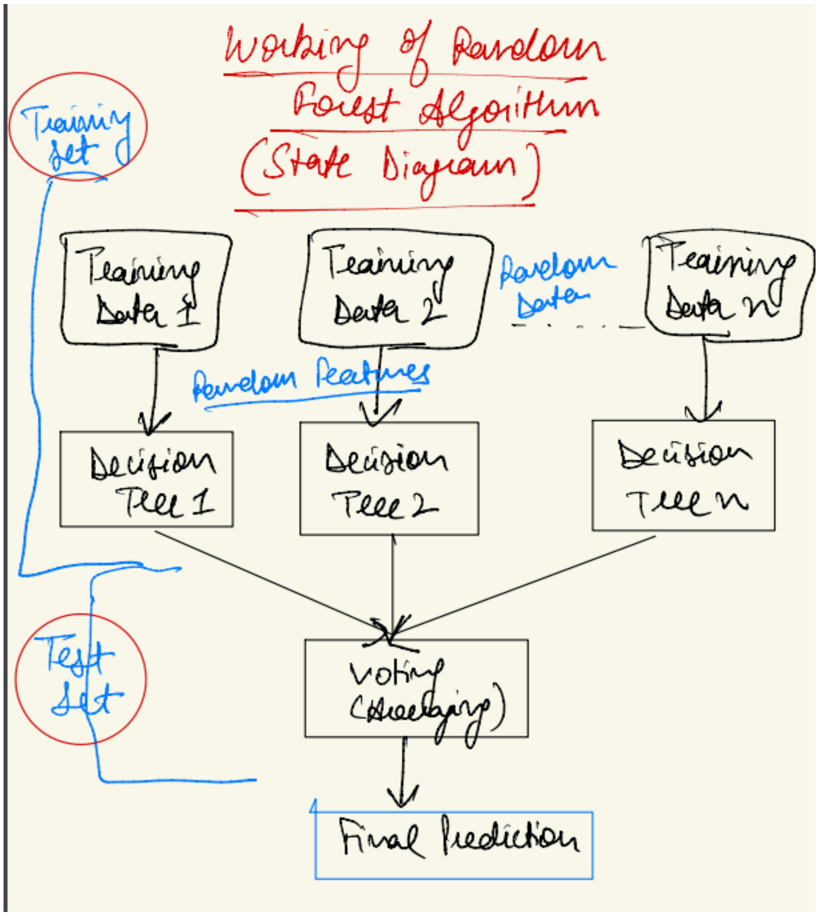
The following diagram shown below will depict how the app connects to the APIs, and how the information is fetched and rendered on to the app dashboard.


Localhost server
http://127.0.0.1:5000/
User
Launches app.py
Internet
Standard html file
home.html
Real-time Insights (/realtime)
Sends Request to Fetch data from APIs
Historical Insights(/historical_insights)
Google Maps API
OpenWeather Map aPI
Successful results back (/insights.html)

State Diagram for Predictive Modelling – Random Forest Algorithm:

[Start] --> [Initialize Random Forest Model] --> [Train Model with Historical Traffic Data] --> [Make Predictions for Real-time Traffic Data] --> [Evaluate Model Performance] --> [End]

The diagram below shows the traffic data is collected, and various data samples are trained to find the traffic routes, and finally out of those samples, an ideal route is selected as final output.



The diagram shown above provides the general flow of processes and is a simplified State Diagram of Random Forest Algorithm.

2.7 Testing

The testing process for the project will encompass various aspects to ensure the robustness, reliability, and usability of the web-based dashboard application. The types of tests to be conducted include:

1. Functionality Testing:

This involves verifying that all features and functionalities of the application work as intended according to the specified requirements.

Test Case 1 - Valid Input

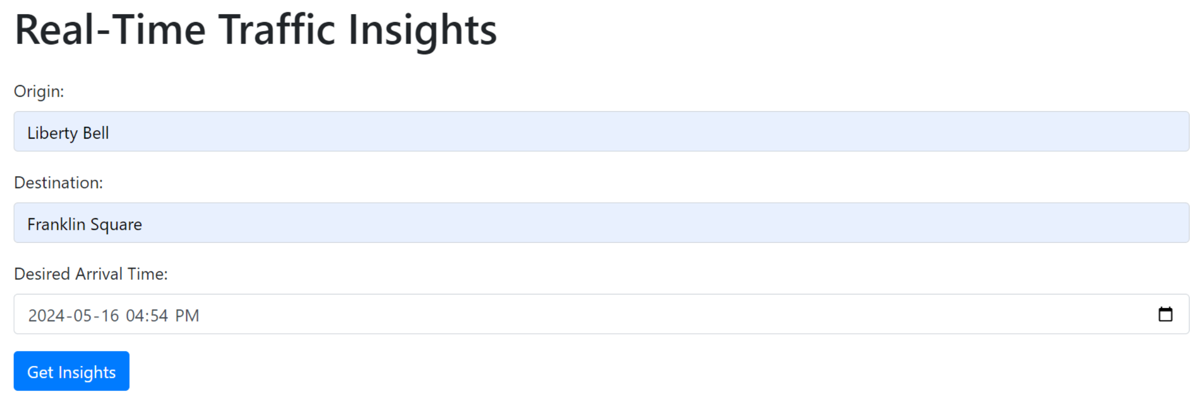
- Description: Ensure the application properly handles valid input for origin, destination, and desired arrival time.

-Steps:

1. Send a POST request to `/insights` with valid input parameters.

2. Verify that the application processes the request successfully and returns insights data.

- Expected Outcome: The application should process valid input and return insights data without errors.


Real-Time Traffic Insights

Origin:
Liberty Bell

Destination:
Franklin Square

Desired Arrival Time:
2024-05-16 04:54 PM

All the inputs here are valid, and then the user clicks on "Get Insights" button to get the insights.

Test Case 2 - Traffic Data Retrieval

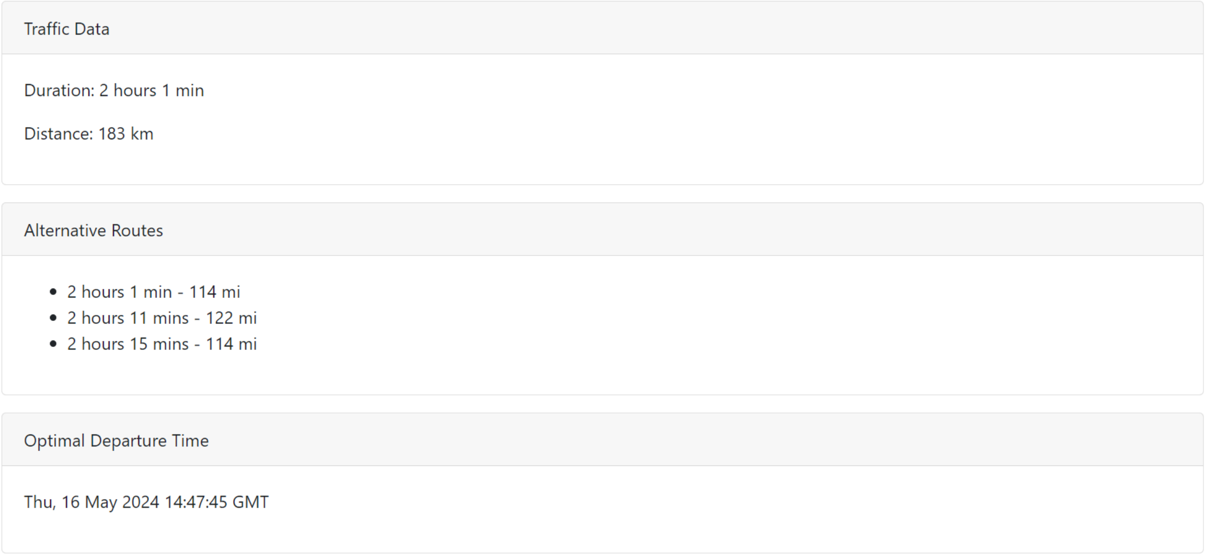
- Description: This test validates the application's ability to retrieve traffic data from the Google Maps API.

-Steps:

1. Send a POST request to `/insights` with valid origin, destination, and arrival time.

2. Verify that the application returns traffic data including distance, duration, and congestion level.

- Expected Outcome: The application should successfully fetch and display traffic data.


Traffic Data

Duration: 2 hours 1 min

Distance: 183 km

Alternative Routes

- 2 hours 1 min - 114 mi
- 2 hours 11 mins - 122 mi
- 2 hours 15 mins - 114 mi

Optimal Departure Time

Thu, 16 May 2024 14:47:45 GMT

Test Case 3 - Weather Data Retrieval

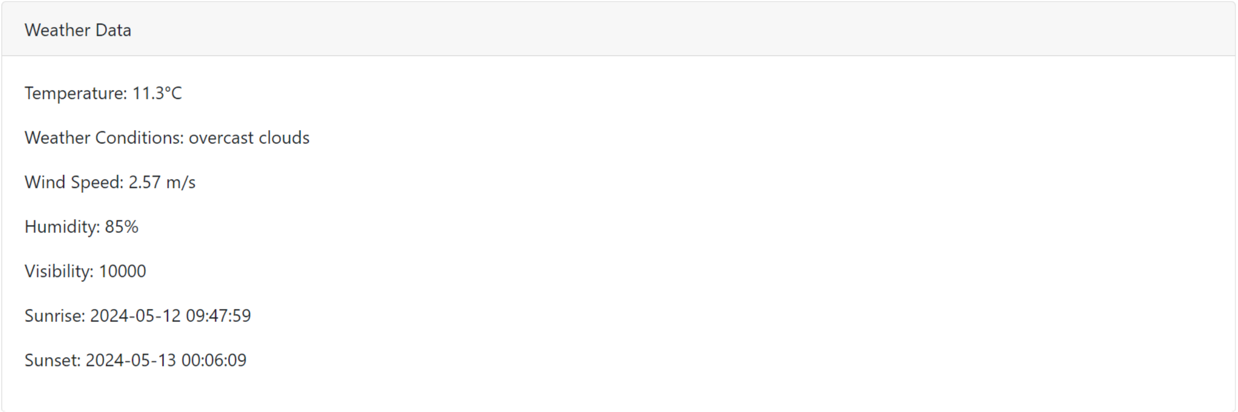
- Description: This test verifies the application's capability to retrieve weather data from the OpenWeatherMap API.

-Steps:

1. Send a POST request to `/insights` with a valid city for weather data retrieval.

2. Verify that the application returns weather information such as temperature, conditions, and sunrise/sunset times.

- Expected Outcome: The application should fetch and display accurate weather data.



Weather Data

Temperature: 11.3°C

Weather Conditions: overcast clouds

Wind Speed: 2.57 m/s

Humidity: 85%

Visibility: 10000

Sunrise: 2024-05-12 09:47:59

Sunset: 2024-05-13 00:06:09

The city entered for this Search is “Philadelphia, PA” – Default city for this Project.

2. Error Handling:

Testing how the application responds to unexpected situations, such as server errors, network issues, or invalid inputs, to provide informative error messages and maintain user experience.

Test Case 4 - Error Handling for Invalid Input Format for Origin

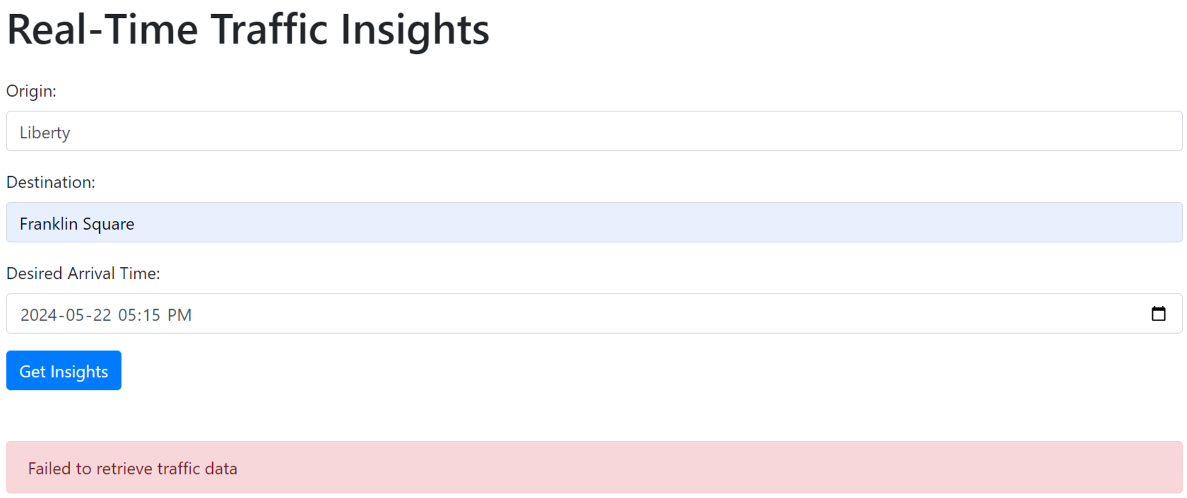
- Description: This test evaluates the application's handling of invalid inputs of Origin and does not produce any results or Weather-Traffic Insights

-Steps:

1. Send a POST request to `/insights` with an invalid origin and hit Get Insights button

2. Verify that the application returns an appropriate error message.

-Expected Outcome: The application should display the error message on the screen.



Real-Time Traffic Insights

Origin: Liberty

Destination: Franklin Square

Desired Arrival Time: 2024-05-22 05:15 PM

Failed to retrieve traffic data

Inputting Incorrect Origin with only “Liberty” instead of “Liberty Bell” produced an error message.

Test Case 5 - Error Handling for Invalid Input Format for Destination

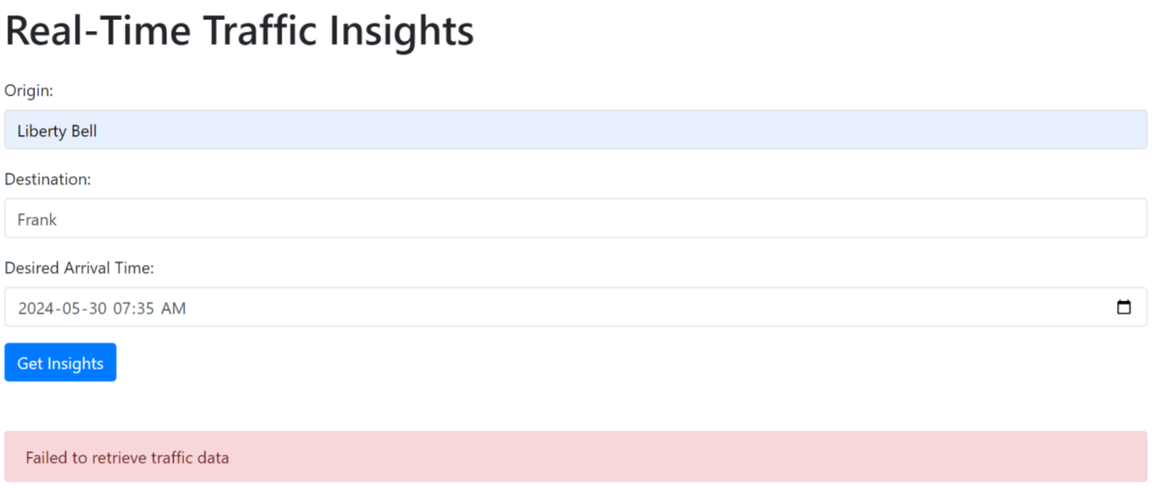
- Description: This test evaluates the application's handling of invalid inputs of Destination and does not produce any results or Weather-Traffic Insights

-Steps:

1. Send a POST request to `/insights` with an invalid destination and hit Get Insights button

2. Verify that the application returns an appropriate error message.

-Expected Outcome: The application should display the error message on the screen.



Real-Time Traffic Insights

Origin: Liberty Bell

Destination: Frank

Desired Arrival Time: 2024-05-30 07:35 AM

Failed to retrieve traffic data

Inputting Incorrect Destination with only “Frank” instead of “Franklin Square” produced an error message.

Test Case 6 - Error Handling for inputting Desired arrival time in the past.

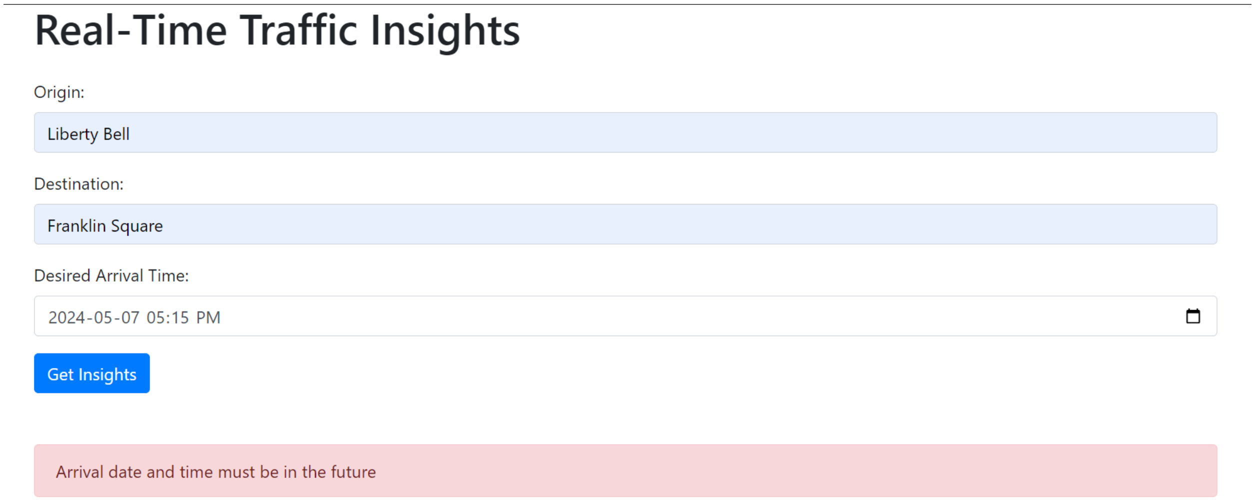
- Description: This test evaluates the application's handling of invalid inputs of desired arrival time and does not produce any results or Weather-Traffic Insights and produce an error message

-Steps:

1. Send a POST request to `/insights` with an invalid arrival time or add time in the past and hit Get Insights button

2. Verify that the application returns an appropriate error message.

-Expected Outcome: The application should display an error message on the screen asking users to input desired arrival time for future.



Real-Time Traffic Insights

Origin: Liberty Bell

Destination: Franklin Square

Desired Arrival Time: 2024-05-07 05:15 PM

Arrival date and time must be in the future

Taking the data on 2024-05-12, the date of 2024-05-07 in the past and thus traffic data in the past is not possible, thus displaying an error message.

3. Input Validation:

The objective of this testing is to ensure that the application properly handles scenarios where required input fields cannot be left empty by the user. When a required field is left empty, the application should display an error message prompting the user to input the missing field before proceeding.

Test Case 7 – Input Validation for Origin

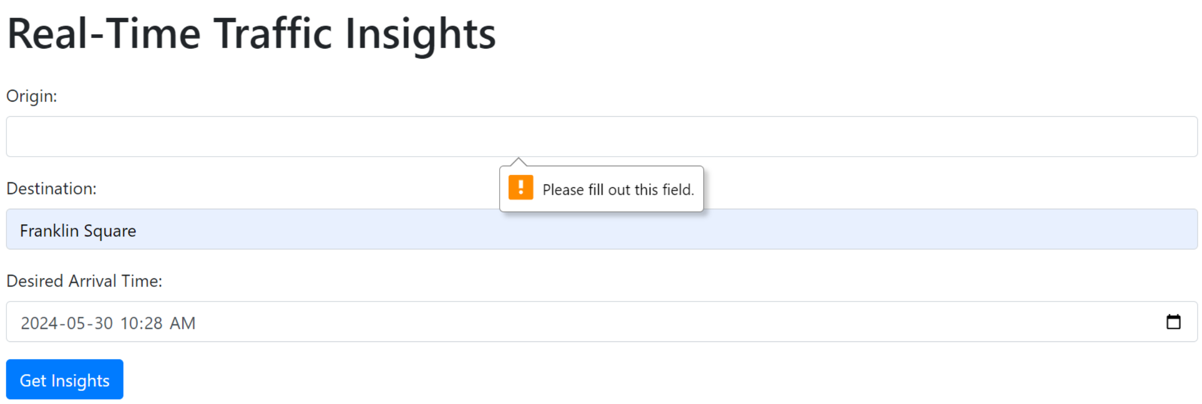
- Description: This test evaluates the application's handling of missing input of origin and does not produce any results or Weather-Traffic Insights but rather asking user to enter the origin field

-Steps:

1. Skip the Origin Field, and enter the valid destination and Arrival time field and click Get Insights

2. Verify that the application returns an alert to fill in the Origin

-Expected Outcome: The application should display an alert asking the user to enter the Origin field.



Real-Time Traffic Insights

Origin:

Please fill out this field.

Destination: Franklin Square

Desired Arrival Time: 2024-05-30 10:28 AM

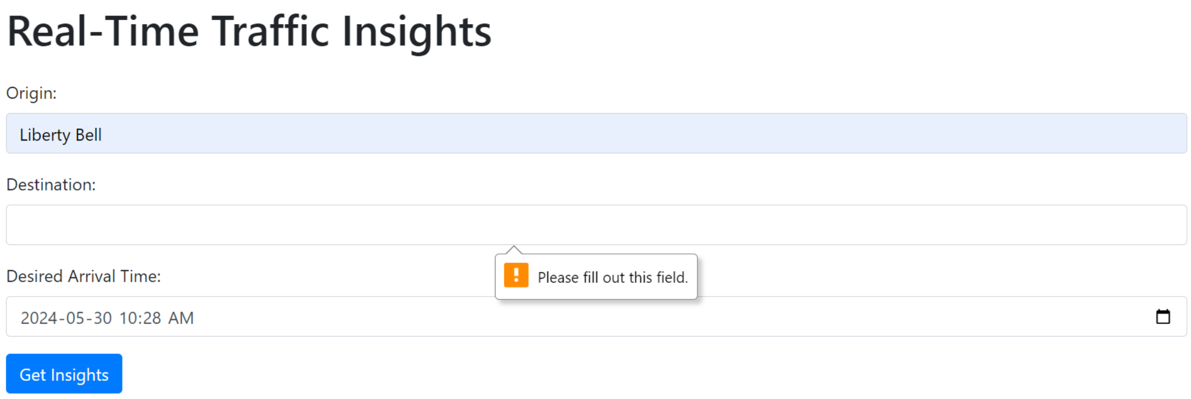
Test Case 8 – Input Validation for Destination

- Description: This test evaluates the application's handling of missing input of destination and does not produce any results or Weather-Traffic Insights but rather asking user to fill out the destination field

-Steps:

1. Skip the destination Field, and enter the valid origin and Arrival time field and click Get Insights
2. Verify that the application returns an alert to fill in the destination

-Expected Outcome: The application should display an alert asking the user to enter the destination field.



Real-Time Traffic Insights

Origin: Liberty Bell

Destination:

Please fill out this field.

Desired Arrival Time: 2024-05-30 10:28 AM

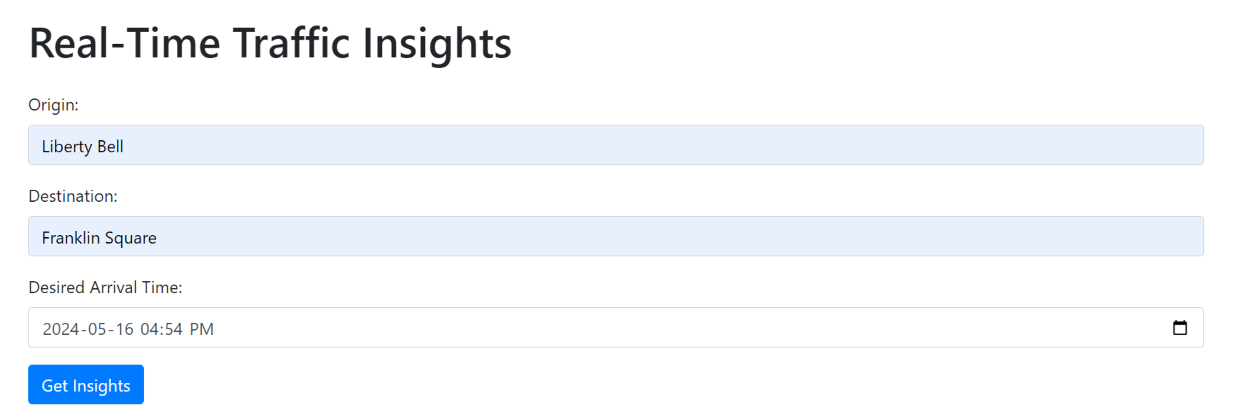
Test Case 9 – Input Validation for Desired Arrival Time

- Description: This test evaluates the application's handling of missing input of Arrival time and does not produce any results or Weather-Traffic Insights but rather asking user to fill out the Arrival time field

-Steps:

1. Skip the Arrival time Field, and enter the valid origin and Destination field and click Get Insights
2. Verify that the application returns an alert to fill in the Arrival time

-Expected Outcome: The application should display an alert asking the user to enter the Arrival time field.



Real-Time Traffic Insights

Origin: Liberty Bell

Destination: Franklin Square

Desired Arrival Time: 2024-05-16 04:54 PM

4. Navigation Testing

This type of testing verifies that all links present in the application, such as hyperlinks, buttons, or navigation menus, correctly navigate the user to the intended pages or destinations.

Test Case 10 – Testing Links on Home Page

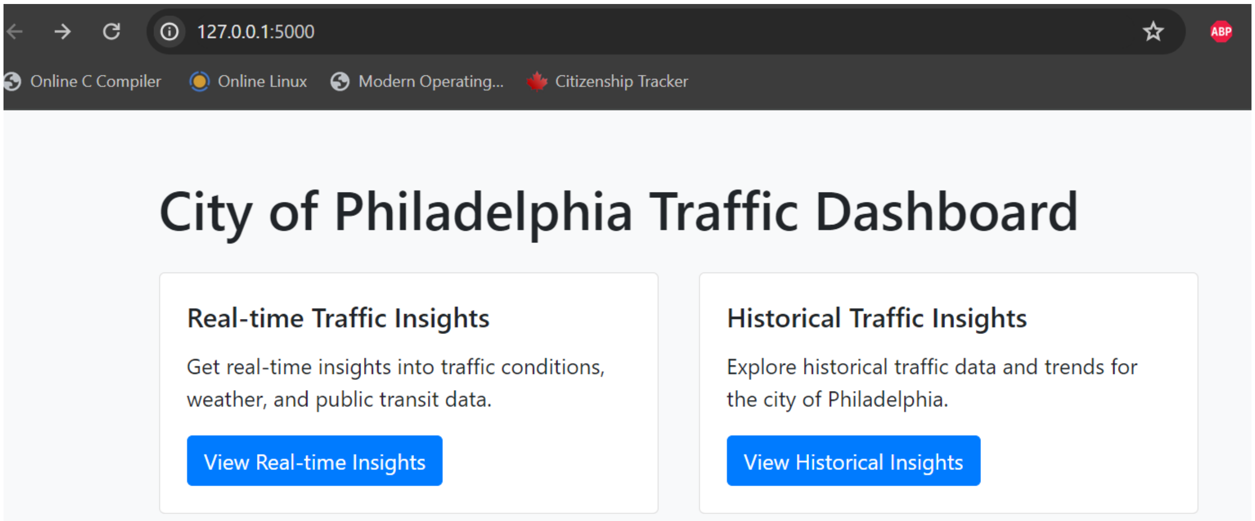
- Description: This test verifies that all links on the home page correctly navigate to the respective pages, such as real-time insights, historical insights, or clustered traffic map.

-Steps:

1. Launch the App – Get to the Home page (home.html)
2. Click on "View Real-Time Insights" and "View Historical Insights" button
3. Verify that first button will take you to "index.html" and second button will take you to "historical_insights.html" page respectively.

-Expected Outcome: The application should open different pages according to the button clicked without any errors

Home Page:



City of Philadelphia Traffic Dashboard

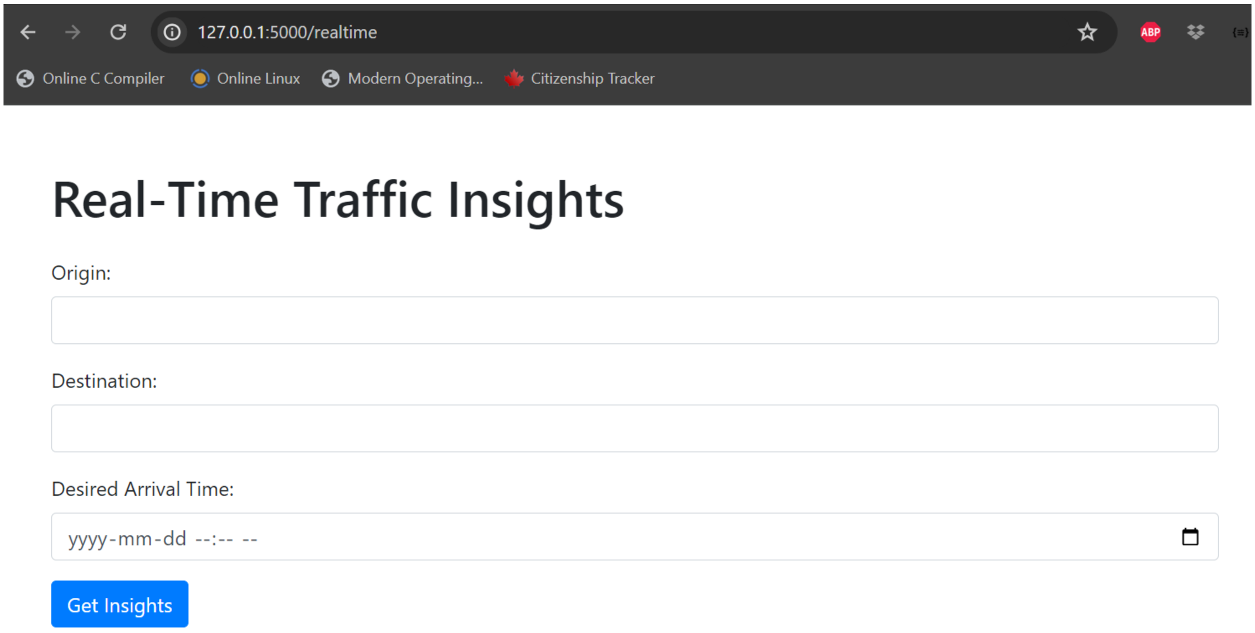
Real-time Traffic Insights

Get real-time insights into traffic conditions, weather, and public transit data.

Historical Traffic Insights

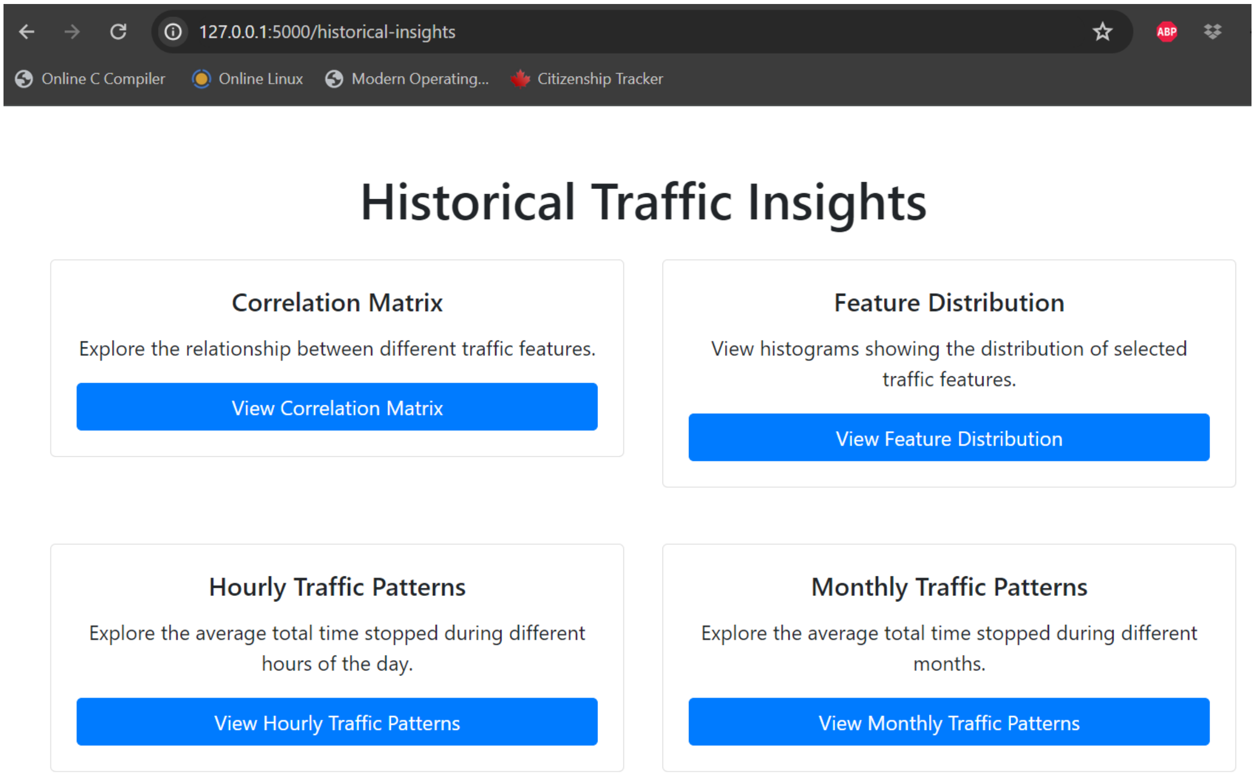
Explore historical traffic data and trends for the city of Philadelphia.

Clicking on “View Real-time Insights” button:



Real-Time Traffic Insights

Clicking on “View Historical Insights" button:



Historical Traffic Insights

Correlation Matrix

Explore the relationship between different traffic features.

Feature Distribution

View histograms showing the distribution of selected traffic features.

Hourly Traffic Patterns

Explore the average total time stopped during different hours of the day.

Monthly Traffic Patterns

Explore the average total time stopped during different months.

Test Case 11 – Testing Links/Buttons on Historical Insights Page

- Description: This test verifies that all buttons on the Historical Insights page correctly display their following insight as per the caption.

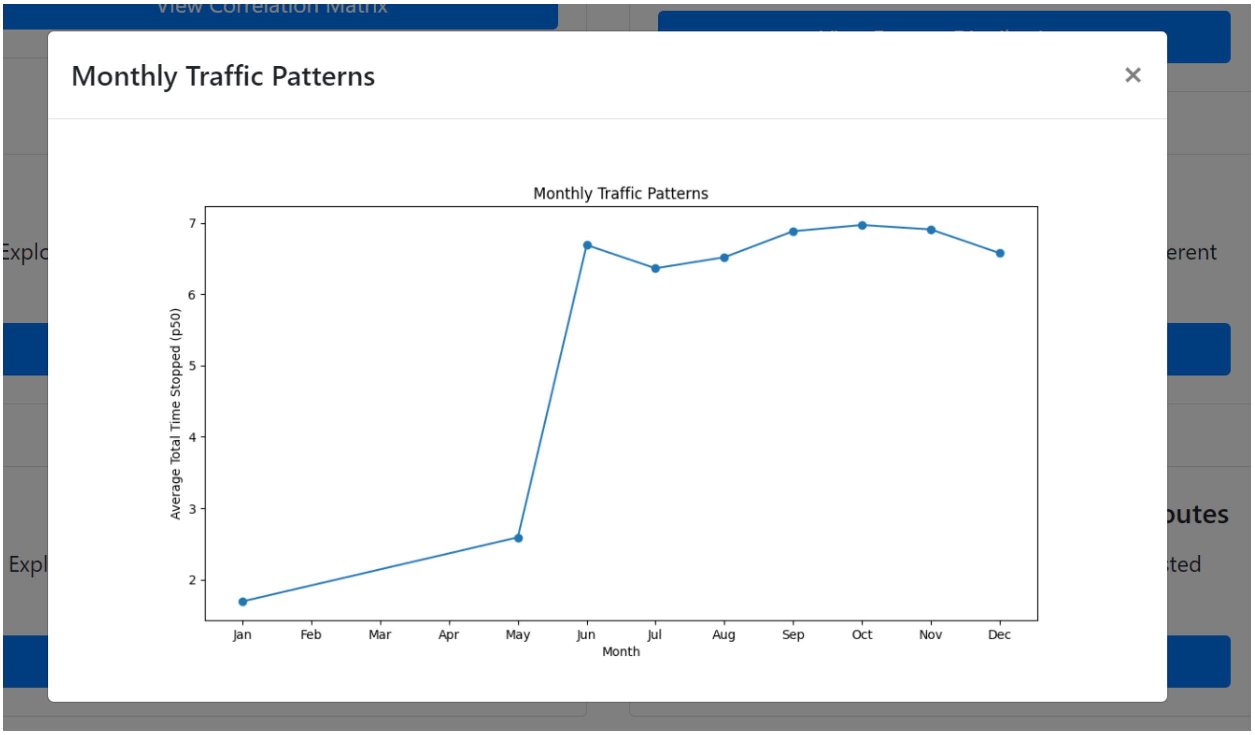
-Steps:

1. Launch the App – Get to the Home -> Historical Insights page

2. Click on all the buttons to see if they open to their respective insights in Modals

-Expected Outcome: The buttons should display the respective insights in modals.

For ex:



Clicking on “View Monthly Traffic Patterns” displayed this graph in modal while the page is in the background.

Test Case 12 – Testing Clustered Traffic Map

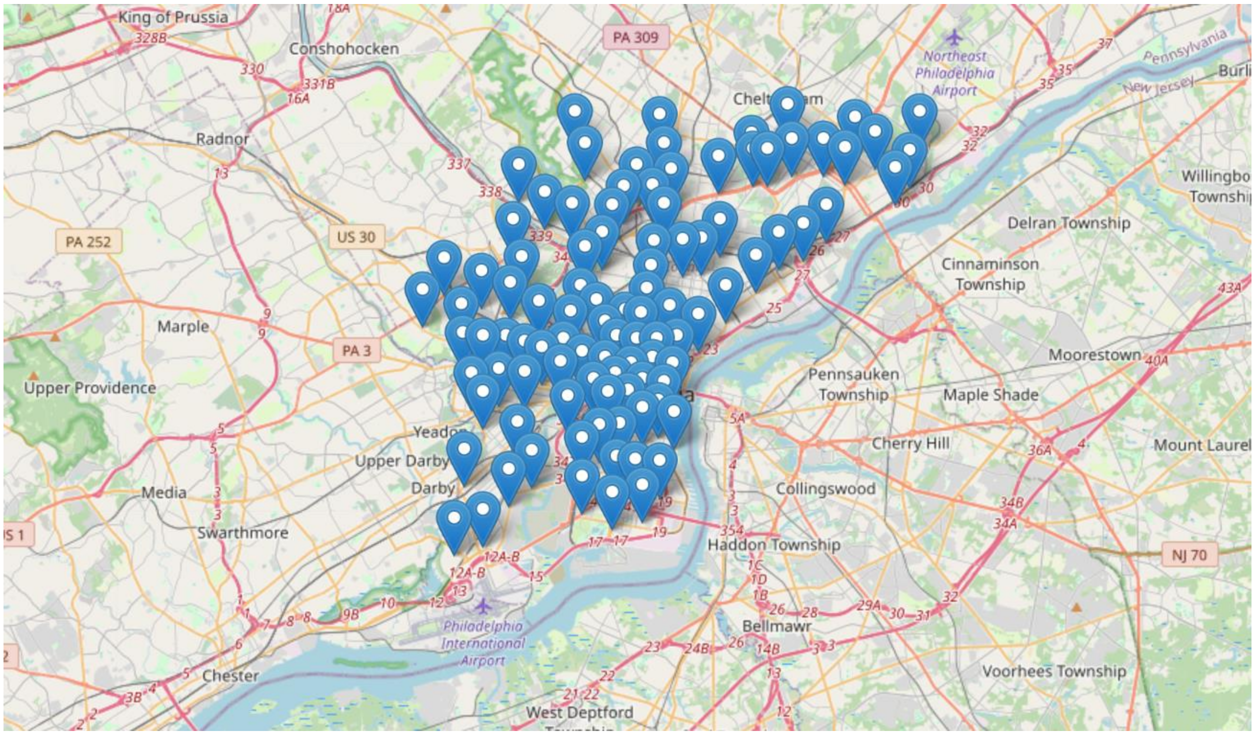
- Description: This test verifies that the button “View Clustered Traffic Map” when clicked, opens to a new page “clustered_traffic_map.html” which is an interactive map of City of Philadelphia showing the clusters of points for the traffic – which can be zoomed in and zoomed out.

-Steps:

1. Launch the App – Get to the Home -> Historical Insights page -> Scroll down to “View Clustered Traffic Map.”

2. Click on “View Clustered Traffic Map” to see if it opens to another page

-Expected Outcome: The buttons should open to a new page which shows Map of City of Philadelphia with clusters of Points showing traffic.



5. Unit Testing

Unit testing is a crucial aspect of software development aimed at ensuring the reliability and correctness of individual units or components of the code. In our project, we employed unit testing to verify the functionality of specific functions and methods within the application. This section outlines the test cases conducted to validate the behavior of various features and functionalities.

Test Case 13 – Testing get_alternative_routes Function.

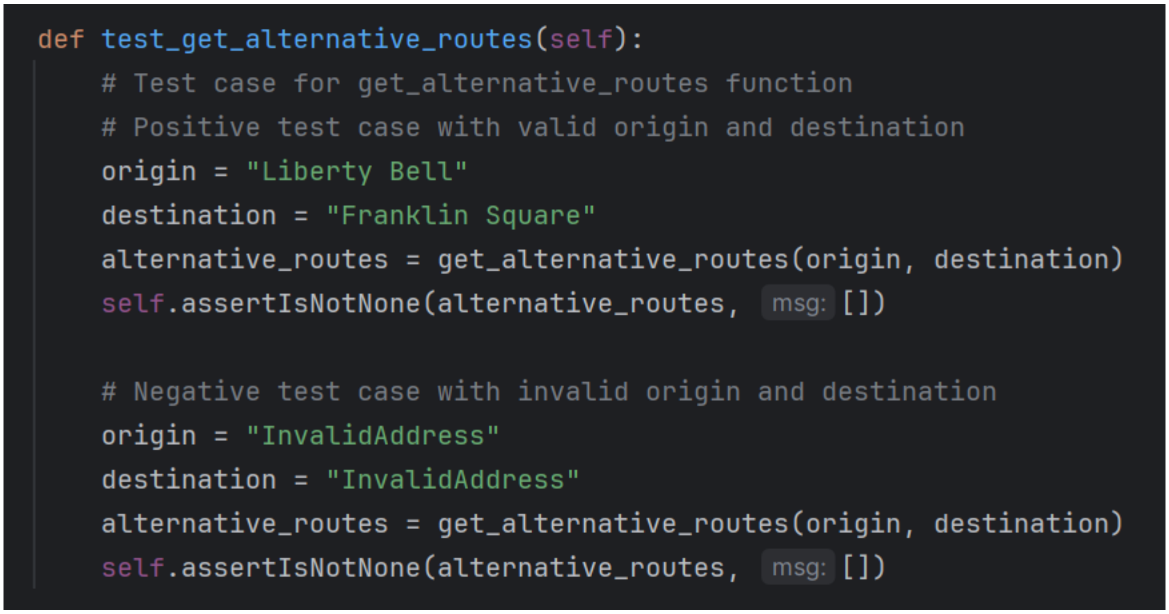
Description: This test verifies the functionality of the get_alternative_routes function, which is responsible for fetching alternative routes between two given locations.

Steps:

Provide valid origin and destination locations.

Invoke the get_alternative_routes function.

Expected Outcome: The function should return a list of alternative routes if they exist. If no alternative routes are available, the function should return an empty list.



```
def test_get_alternative_routes(self):
    # Test case for get_alternative_routes function
    # Positive test case with valid origin and destination
    origin = "Liberty Bell"
    destination = "Franklin Square"
    alternative_routes = get_alternative_routes(origin, destination)
    self.assertIsNotNone(alternative_routes,  msg: [])

    # Negative test case with invalid origin and destination
    origin = "InvalidAddress"
    destination = "InvalidAddress"
    alternative_routes = get_alternative_routes(origin, destination)
    self.assertIsNotNone(alternative_routes,  msg: [])
```

Test Case 14 – Testing get_public_transit_data Function.

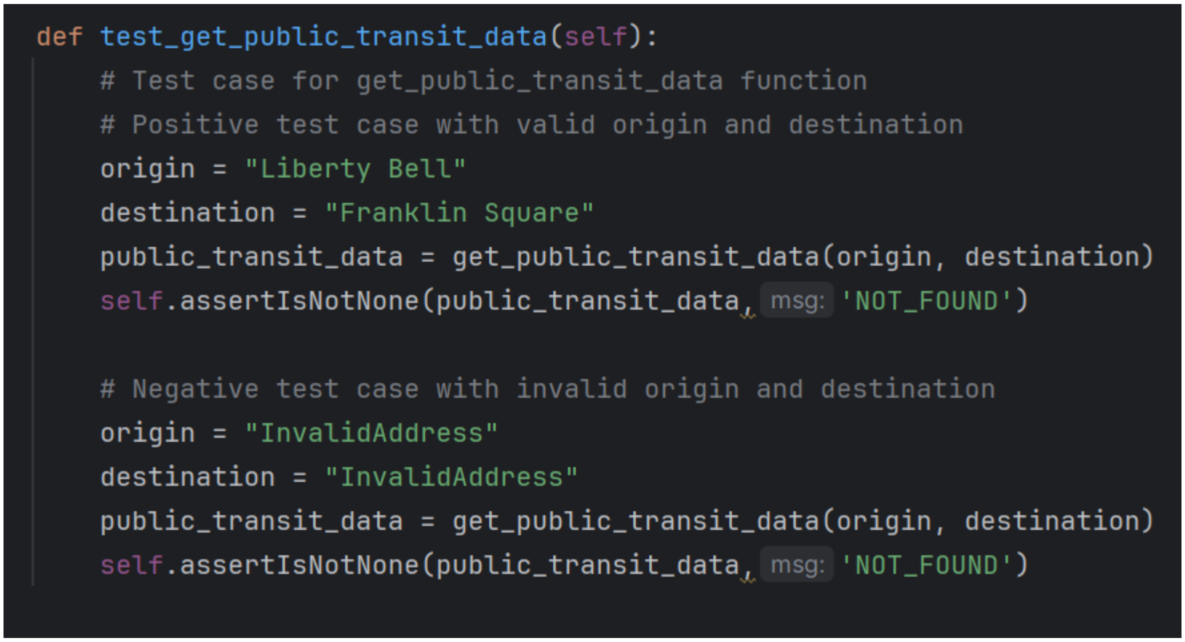
Description: This test validates the behavior of the get_public_transit_data function, responsible for retrieving public transit data between specified origin and destination points.

Steps:

Provide valid origin and destination locations.

Invoke the get_public_transit_data function.

Expected Outcome: The function should return transit data in JSON format if available. In case of invalid addresses or no routes found, the function should return a JSON object with status indicating the issue.



```
def test_get_public_transit_data(self):
    # Test case for get_public_transit_data function
    # Positive test case with valid origin and destination
    origin = "Liberty Bell"
    destination = "Franklin Square"
    public_transit_data = get_public_transit_data(origin, destination)
    self.assertIsNotNone(public_transit_data, msg: 'NOT_FOUND')

    # Negative test case with invalid origin and destination
    origin = "InvalidAddress"
    destination = "InvalidAddress"
    public_transit_data = get_public_transit_data(origin, destination)
    self.assertIsNotNone(public_transit_data, msg: 'NOT_FOUND')
```

Test Case 15 – Testing get_traffic_incidents Function.

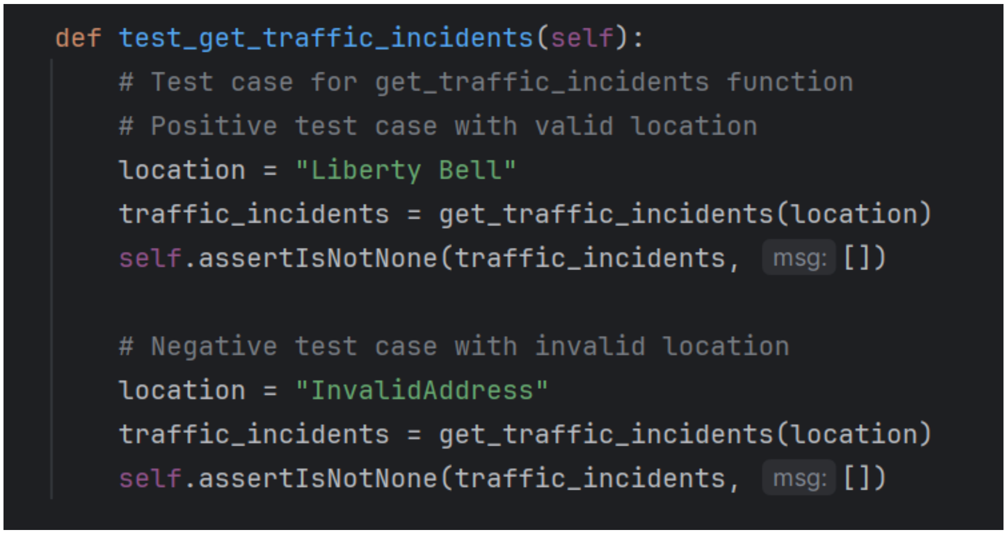
Description: This test assesses the functionality of the get_traffic_incidents function, which retrieves traffic incidents near a given location.

Steps:

Provide a valid location.

Invoke the get_traffic_incidents function.

Expected Outcome: The function should return a list of traffic incidents if they exist. If no incidents are found, the function should return an empty list.



```
def test_get_traffic_incidents(self):
    # Test case for get_traffic_incidents function
    # Positive test case with valid location
    location = "Liberty Bell"
    traffic_incidents = get_traffic_incidents(location)
    self.assertIsNotNone(traffic_incidents, msg: [])

    # Negative test case with invalid location
    location = "InvalidAddress"
    traffic_incidents = get_traffic_incidents(location)
    self.assertIsNotNone(traffic_incidents, msg: [])
```

Test Case 16 – Testing Weather Data Retrieval

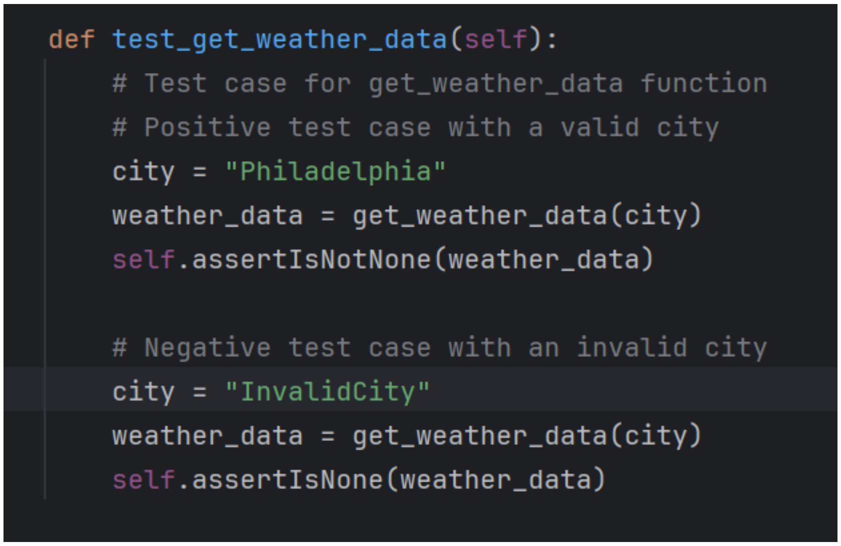
Description: This test evaluates the functionality of the get_weather_data function, responsible for fetching weather data for a specified city.

Steps:

Provide a valid city name.

Invoke the get_weather_data function.

Expected Outcome: The function should return weather data in the form of a dictionary containing temperature, weather conditions, wind speed, humidity, visibility, sunrise time, and sunset time. In case of an error, such as an invalid city name, the function should return None.



```python
def test_get_weather_data(self):
    # Test case for get_weather_data function
    # Positive test case with a valid city
    city = "Philadelphia"
    weather_data = get_weather_data(city)
    self.assertIsNotNone(weather_data)

    # Negative test case with an invalid city
    city = "InvalidCity"
    weather_data = get_weather_data(city)
    self.assertIsNone(weather_data)
```

Test Case 17 – Testing Traffic Data Retrieval

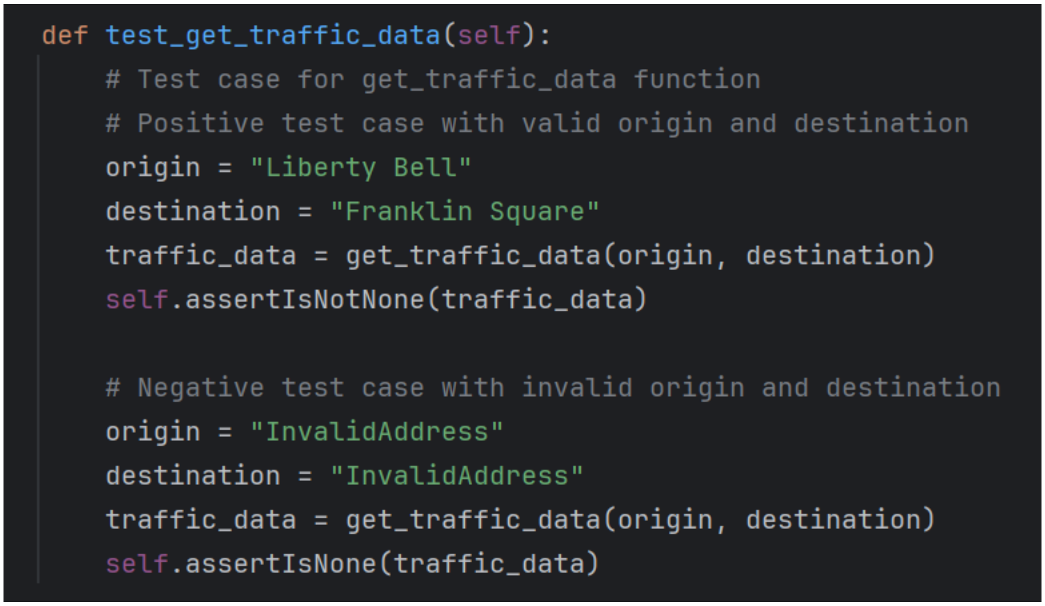
Description: This test verifies the behavior of the get_traffic_data function, which retrieves real-time traffic data between two specified locations.

Steps:

Provide valid origin and destination addresses.

Invoke the get_traffic_data function.

Expected Outcome: The function should return traffic data, including distance, duration, duration in traffic (if available), and congestion level. In case of an error, such as invalid addresses or no routes found, the function should return None.



```
def test_get_traffic_data(self):
    # Test case for get_traffic_data function
    # Positive test case with valid origin and destination
    origin = "Liberty Bell"
    destination = "Franklin Square"
    traffic_data = get_traffic_data(origin, destination)
    self.assertIsNotNone(traffic_data)

    # Negative test case with invalid origin and destination
    origin = "InvalidAddress"
    destination = "InvalidAddress"
    traffic_data = get_traffic_data(origin, destination)
    self.assertIsNone(traffic_data)
```

Test Case 18 – Testing Optimal Departure Time Suggestion

Description: This test assesses the functionality of the suggest_optimal_departure_time function, which suggests an optimal departure time based on desired arrival time and real-time traffic data.

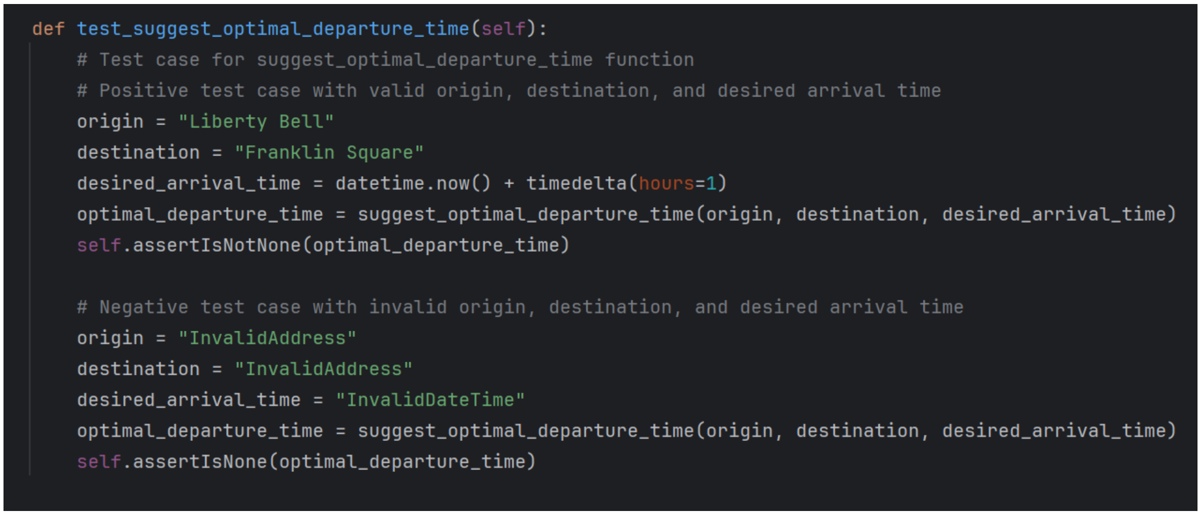
Steps:

Provide valid origin and destination addresses.

Specify a desired arrival time.

Invoke the suggest_optimal_departure_time function.

Expected Outcome: The function should return an optimal departure time that accounts for real-time traffic conditions. If no traffic data is available or if the desired arrival time is not feasible, the function should return None.



```python
def test_suggest_optimal_departure_time(self):
    # Test case for suggest_optimal_departure_time function
    # Positive test case with valid origin, destination, and desired arrival time
    origin = "Liberty Bell"
    destination = "Franklin Square"
    desired_arrival_time = datetime.now() + timedelta(hours=1)
    optimal_departure_time = suggest_optimal_departure_time(origin, destination, desired_arrival_time)
    self.assertIsNotNone(optimal_departure_time)

    # Negative test case with invalid origin, destination, and desired arrival time
    origin = "InvalidAddress"
    destination = "InvalidAddress"
    desired_arrival_time = "InvalidDateTime"
    optimal_departure_time = suggest_optimal_departure_time(origin, destination, desired_arrival_time)
    self.assertIsNone(optimal_departure_time)
```

Test Case 19 – Testing Optimal Departure Time with Future Arrival

Description: This test verifies the calculation of the optimal departure time when provided with a future arrival time.

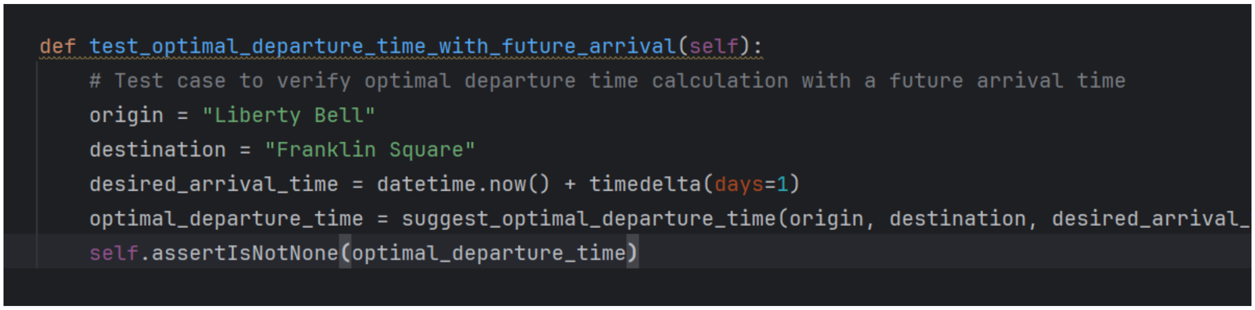
Steps:

Specify a valid origin and destination.

Define a desired arrival time in the future.

Calculate the optimal departure time using the suggest_optimal_departure_time function.

Expected Outcome: The function should return a valid optimal departure time based on the specified future arrival time.



```python
def test_optimal_departure_time_with_future_arrival(self):
    # Test case to verify optimal departure time calculation with a future arrival time
    origin = "Liberty Bell"
    destination = "Franklin Square"
    desired_arrival_time = datetime.now() + timedelta(days=1)
    optimal_departure_time = suggest_optimal_departure_time(origin, destination, desired_arrival_
    self.assertIsNotNone(optimal_departure_time)
```

Test Case 20 – Testing No Alternative Routes for Straight Path

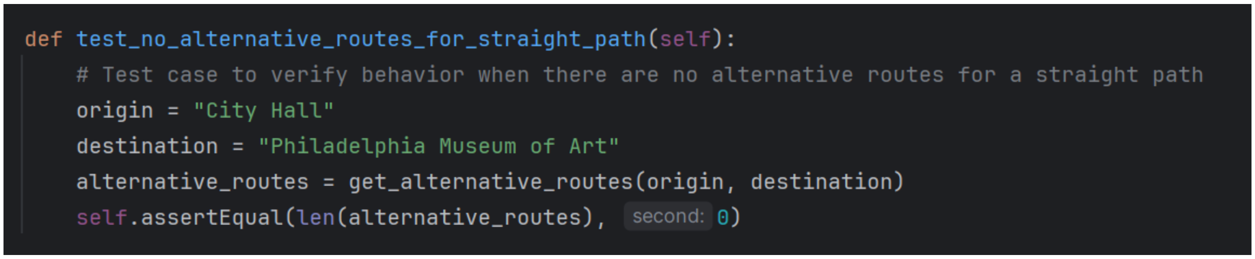
Description: This test checks the application's behavior when no alternative routes are available for a straight path between two locations.

Steps:

Provide two locations that form a straight path.

Retrieve alternative routes using the get_alternative_routes function.

Expected Outcome: The function should return an empty list since there are no alternative routes available for the straight path between the provided locations.



```python
def test_no_alternative_routes_for_straight_path(self):
    # Test case to verify behavior when there are no alternative routes for a straight path
    origin = "City Hall"
    destination = "Philadelphia Museum of Art"
    alternative_routes = get_alternative_routes(origin, destination)
    self.assertEqual(len(alternative_routes),  second: 0)
```

Test Case 21 – Testing Traffic Incidents in Remote Location

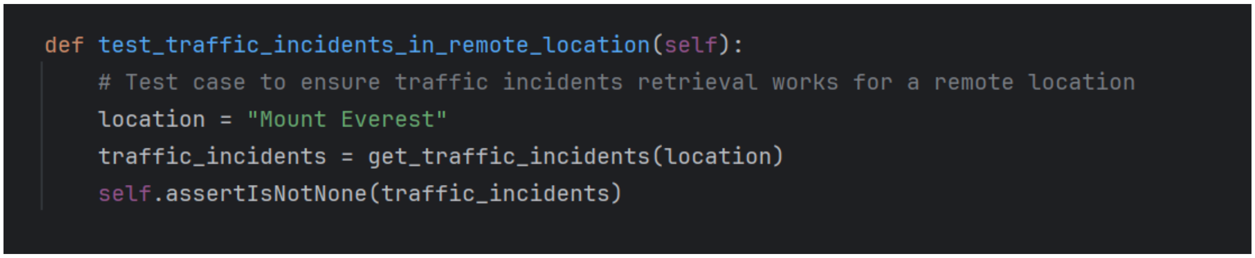
Description: This test verifies the functionality of fetching traffic incidents in a remote location.

Steps:

Specify a remote location, such as Mount Everest.

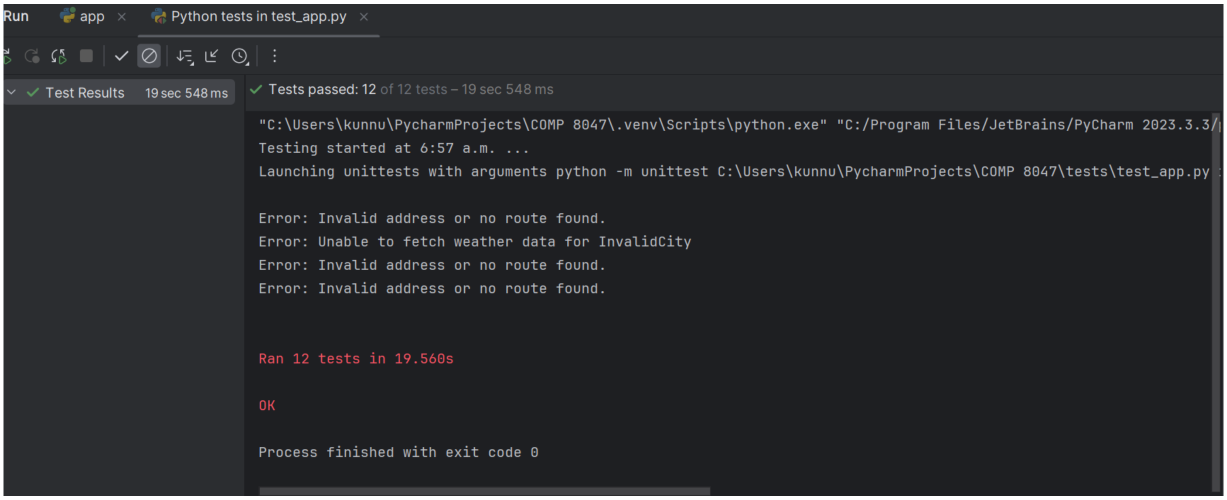
Fetch traffic incidents using the get_traffic_incidents function.

Expected Outcome: The function should return a list of traffic incidents near the remote location, demonstrating its ability to fetch incidents even in remote areas.



```python
def test_traffic_incidents_in_remote_location(self):
    # Test case to ensure traffic incidents retrieval works for a remote location
    location = "Mount Everest"
    traffic_incidents = get_traffic_incidents(location)
    self.assertIsNotNone(traffic_incidents)
```

All the tests were successful:



Tests passed: 12 of 12 tests – 19 sec 548 ms

```
"C:\Users\kunnu\PycharmProjects\COMP 8047\.venv\Scripts\python.exe" "C:/Program Files/JetBrains/PyCharm 2023.3.3/
Testing started at 6:57 a.m. ...
Launching unittests with arguments python -m unittest C:\Users\kunnu\PycharmProjects\COMP 8047\tests\test_app.py

Error: Invalid address or no route found.
Error: Unable to fetch weather data for InvalidCity
Error: Invalid address or no route found.
Error: Invalid address or no route found.


Ran 12 tests in 19.560s

OK

Process finished with exit code 0
```

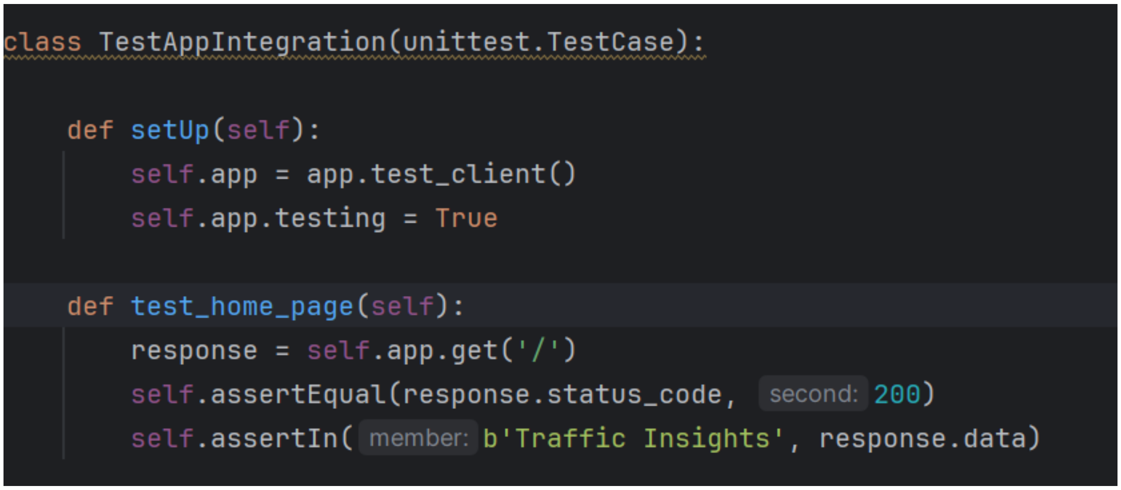
6. Integration Testing

Integration testing is a level of software testing where individual units or components of a software application are combined and tested as a group. The purpose of integration testing is to verify that the interactions between these units function correctly when integrated together. This type of testing helps ensure that the software components work together as expected and that the integrated system meets the specified requirements.

Test Case 22: Testing Home Page

Description: This test verifies that the home page of the application is accessible and displays the correct content.

Expected Outcome: The HTTP response status code should be 200, indicating a successful request. The response should contain the text "Traffic Insights".

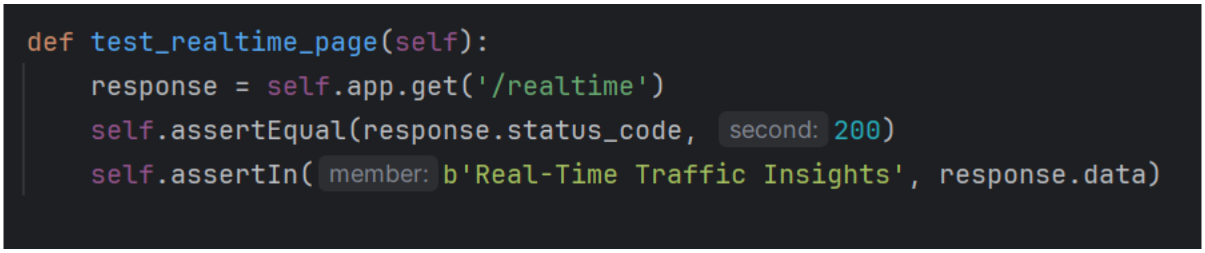


```
class TestAppIntegration(unittest.TestCase):

    def setUp(self):
        self.app = app.test_client()
        self.app.testing = True

    def test_home_page(self):
        response = self.app.get('/')
        self.assertEqual(response.status_code, second: 200)
        self.assertIn( member: b'Traffic Insights', response.data)
```

Test Case 23: Real-Time Page

Description: This test verifies that the real-time traffic insights page is accessible and displays the correct content.

Expected Outcome: The HTTP response status code should be 200, indicating a successful request. The response should contain the text "Real-Time Traffic Insights".

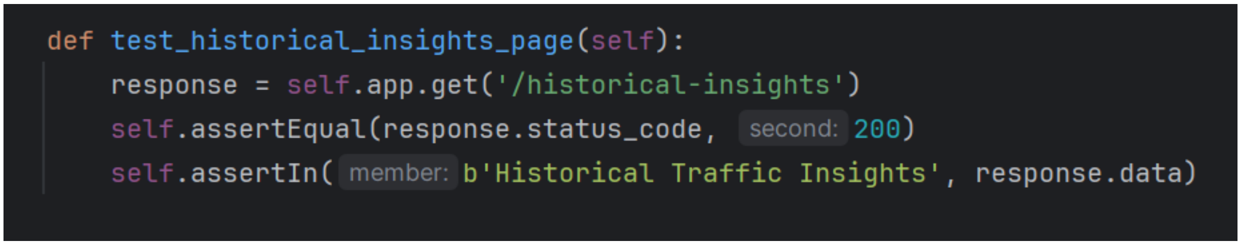


```
def test_realtime_page(self):
    response = self.app.get('/realtime')
    self.assertEqual(response.status_code,  second: 200)
    self.assertIn( member: b'Real-Time Traffic Insights', response.data)
```

Test Case 24: Historical Insights Page

Description: This test verifies that the historical traffic insights page is accessible and displays the correct content.

Expected Outcome: The HTTP response status code should be 200, indicating a successful request. The response should contain the text "Historical Traffic Insights".

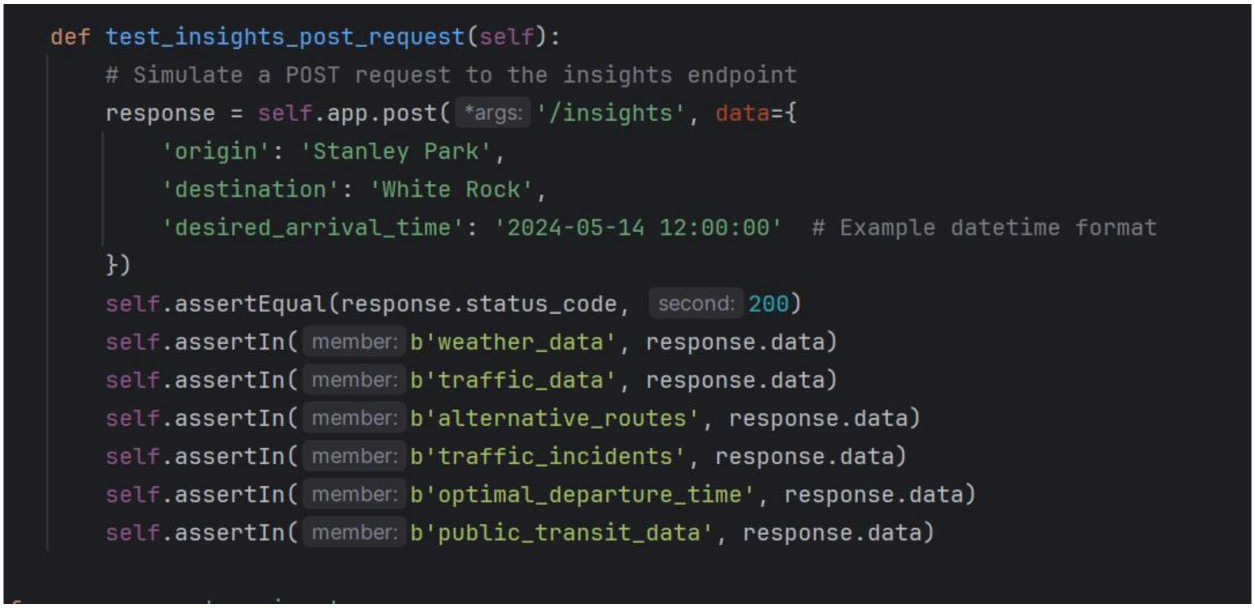


```
def test_historical_insights_page(self):
    response = self.app.get('/historical-insights')
    self.assertEqual(response.status_code,  second: 200)
    self.assertIn( member: b'Historical Traffic Insights', response.data)
```

Test Case 25: Insights POST Request

Description: This test verifies that the POST request to the insights endpoint returns the expected data.

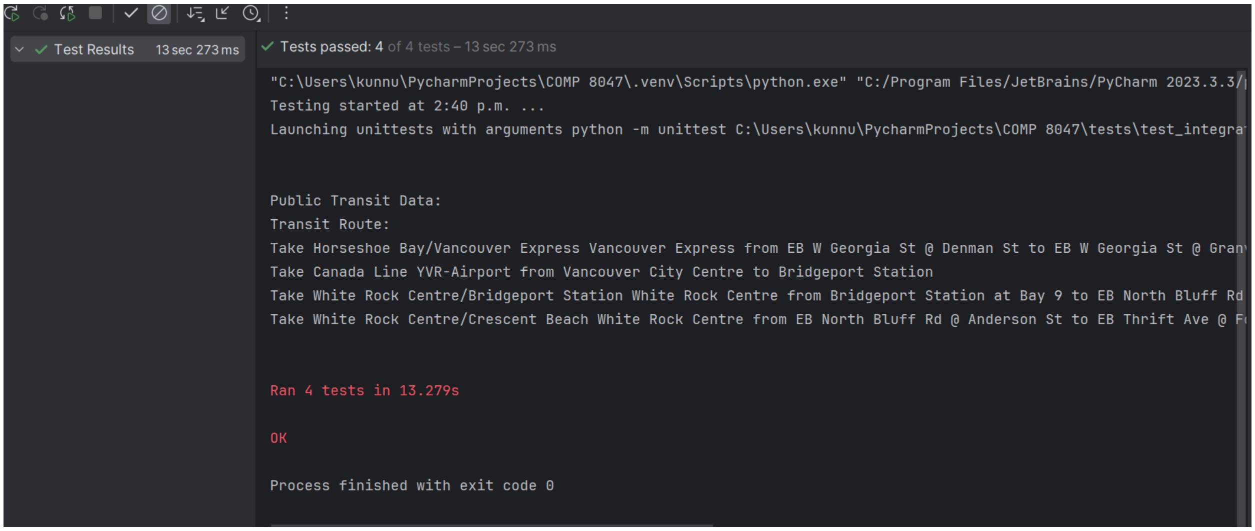
Expected Outcome: The HTTP response status code should be 200, indicating a successful request. The response should contain various data fields including weather data, traffic data, alternative routes, traffic incidents, optimal departure time, and public transit data.



```python
def test_insights_post_request(self):
    # Simulate a POST request to the insights endpoint
    response = self.app.post( *args: '/insights', data={
        'origin': 'Stanley Park',
        'destination': 'White Rock',
        'desired_arrival_time': '2024-05-14 12:00:00'  # Example datetime format
    })
    self.assertEqual(response.status_code,  second: 200)
    self.assertIn( member: b'weather_data', response.data)
    self.assertIn( member: b'traffic_data', response.data)
    self.assertIn( member: b'alternative_routes', response.data)
    self.assertIn( member: b'traffic_incidents', response.data)
    self.assertIn( member: b'optimal_departure_time', response.data)
    self.assertIn( member: b'public_transit_data', response.data)
```

The integration tests were designed to verify the functionality and integration of key components within the application. All test cases passed successfully, indicating that the application's components are integrated correctly and functioning as expected. Integration testing helps ensure that the application meets the specified requirements and delivers the intended functionality to end-users.

All 4 Integration tests passed:



```
✓ Tests passed: 4 of 4 tests – 13 sec 273 ms
"C:\Users\kunnu\PycharmProjects\COMP 8047\.venv\Scripts\python.exe" "C:/Program Files/JetBrains/PyCharm 2023.3.3/
Testing started at 2:40 p.m. ...
Launching unittests with arguments python -m unittest C:\Users\kunnu\PycharmProjects\COMP 8047\tests\test_integra


Public Transit Data:
Transit Route:
Take Horseshoe Bay/Vancouver Express Vancouver Express from EB W Georgia St @ Denman St to EB W Georgia St @ Gran
Take Canada Line YVR-Airport from Vancouver City Centre to Bridgeport Station
Take White Rock Centre/Bridgeport Station White Rock Centre from Bridgeport Station at Bay 9 to EB North Bluff Rd
Take White Rock Centre/Crescent Beach White Rock Centre from EB North Bluff Rd @ Anderson St to EB Thrift Ave @ F


Ran 4 tests in 13.279s

OK

Process finished with exit code 0
```

7. Network Testing:

Network testing involves assessing the performance, reliability, and security of computer networks, including both local area networks (LANs) and wide area networks (WANs). It encompasses various techniques and methodologies to evaluate different aspects of network functionality. The primary objectives of network testing are to ensure that the network infrastructure meets the requirements for data transmission, communication, and resource sharing while maintaining security and integrity.

Test Case 26 – Test if the results are shown when the application stops.

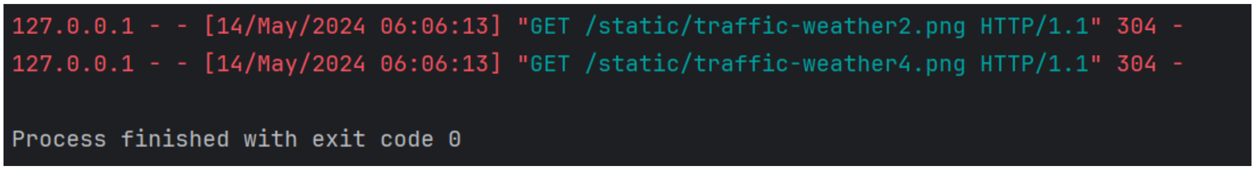
Description: This tests whether the application still produces the result when stopped. Checking if all the pages can still work or load information even after app.py stops running.

Steps:

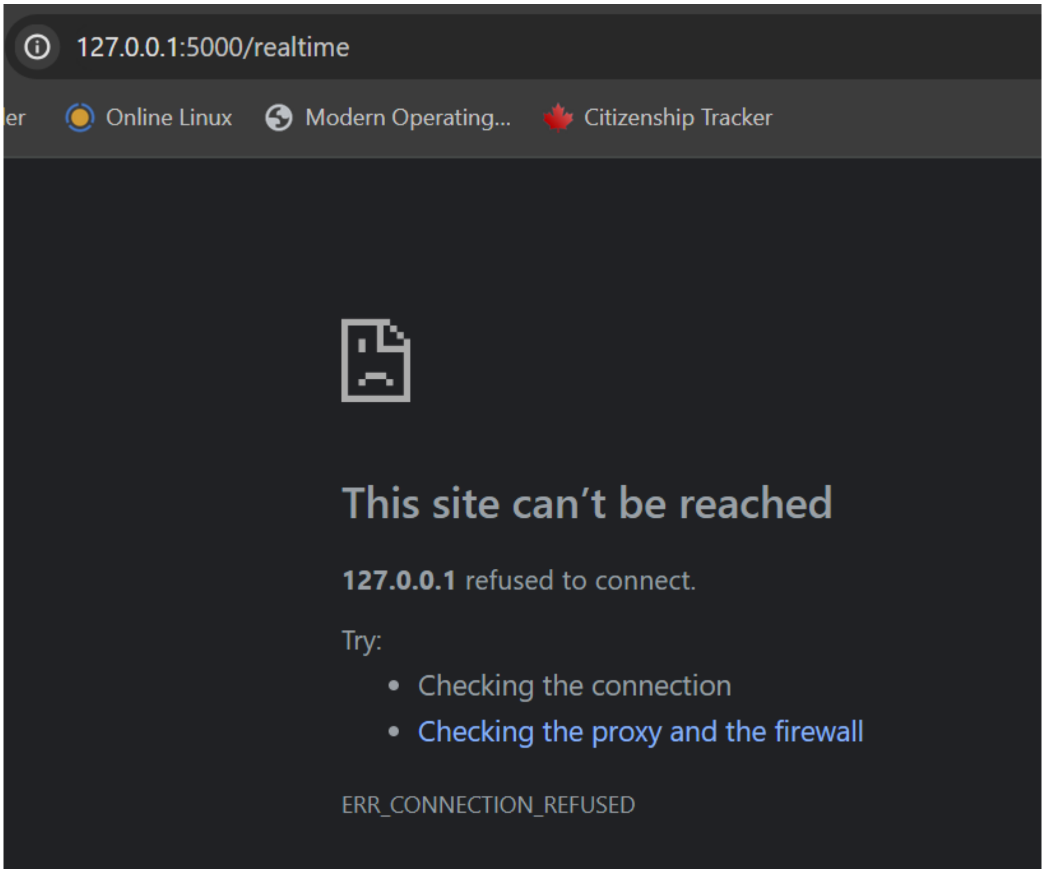
Run the app and load one of the html pages and see the working.

Stop the app from running and check the working again of buttons and links.

Expected Outcome: Static Pages can be loaded from the cache, but after deleting the cache memory, no site, or insight should be displayed.



```
127.0.0.1 - - [14/May/2024 06:06:13] "GET /static/traffic-weather2.png HTTP/1.1" 304 -
127.0.0.1 - - [14/May/2024 06:06:13] "GET /static/traffic-weather4.png HTTP/1.1" 304 -

Process finished with exit code 0
```



This site can’t be reached

127.0.0.1 refused to connect.

Try:

- Checking the connection
- Checking the proxy and the firewall

ERR_CONNECTION_REFUSED

2.8 Predictive Analysis

Data Overview

Independent Variables (Features)

- **IntersectionId:** Represents a unique intersectionID for some intersection of roads within a city.

- **Latitude:** The latitude of the intersection.

- **Longitude:** The longitude of the intersection.

- **EntryStreetName:** The street name from which the vehicle entered towards the intersection.

- **ExitStreetName:** The street name to which the vehicle goes from the intersection.

- **EntryHeading:** Direction to which the car was heading while entering the intersection.

- **ExitHeading:** Direction to which the car went after it went through the intersection.

- **Hour:** The hour of the day.

- **Weekend:** It's weekend or not.

- **Month:** Which Month it is.

- **Path:** It is a concatenation in the format: EntryStreetName_EntryHeading ExitStreetName_ExitHeading.

- **City:** Name of the city.

Dependent Variables (Targets)

- **TotalTimeStopped_p20:** Total time for which 20% of the vehicles had to stop at an intersection.

- **TotalTimeStopped_p40:** Total time for which 40% of the vehicles had to stop at an intersection.

- **TotalTimeStopped_p50:** Total time for which 50% of the vehicles had to stop at an intersection.

- **TotalTimeStopped_p60:** Total time for which 60% of the vehicles had to stop at an intersection.

- **TotalTimeStopped_p80:** Total time for which 80% of the vehicles had to stop at an intersection.

- **TimeFromFirstStop_p20:** Time taken for 20% of the vehicles to stop again after crossing an intersection.

- **TimeFromFirstStop_p40:** Time taken for 40% of the vehicles to stop again after crossing an intersection.

- **TimeFromFirstStop_p50:** Time taken for 50% of the vehicles to stop again after crossing an intersection.

- **TimeFromFirstStop_p60:** Time taken for 60% of the vehicles to stop again after crossing an intersection.

- **TimeFromFirstStop_p80:** Time taken for 80% of the vehicles to stop again after crossing an intersection.

- **DistanceToFirstStop_p20:** How far before the intersection the 20% of the vehicles stopped for the first time.

- **DistanceToFirstStop_p40:** How far before the intersection the 40% of the vehicles stopped for the first time.

- **DistanceToFirstStop_p50:** How far before the intersection the 50% of the vehicles stopped for the first time.

- **DistanceToFirstStop_p60:** How far before the intersection the 60% of the vehicles stopped for the first time.

- **DistanceToFirstStop_p80:** How far before the intersection the 80% of the vehicles stopped for the first time.

Target Output

- Total time stopped at an intersection, 20th, 50th, 80th percentiles and Distance between the intersection and the first place the vehicle stopped and started waiting, 20th, 50th, 80th percentiles

  - **TotalTimeStopped_p20**

  - **TotalTimeStopped_p50**

  - **TotalTimeStopped_p80**

  - **DistanceToFirstStop_p20**

  - **DistanceToFirstStop_p50**

  - **DistanceToFirstStop_p80**

Data Preprocessing

This raw data contained information for four cities from https://www.kaggle.com/code/fayzaalmukharreq/traffic-congestion/notebook

However, for this analysis, the data was filtered to focus solely on the City of Philadelphia. Following this, the dataset underwent preprocessing steps, including cleaning and visualization. Additionally, a separate dataset containing weather information for Philadelphia was preprocessed from https://www.kaggle.com/code/grzegorzlippe/import-json-weather-data

, cleaned, and visualized before merging it with the main dataset based on the common `Month` column. This merged dataset (`merged_df`) was then further visualized to gain insights into the relationships between different variables.

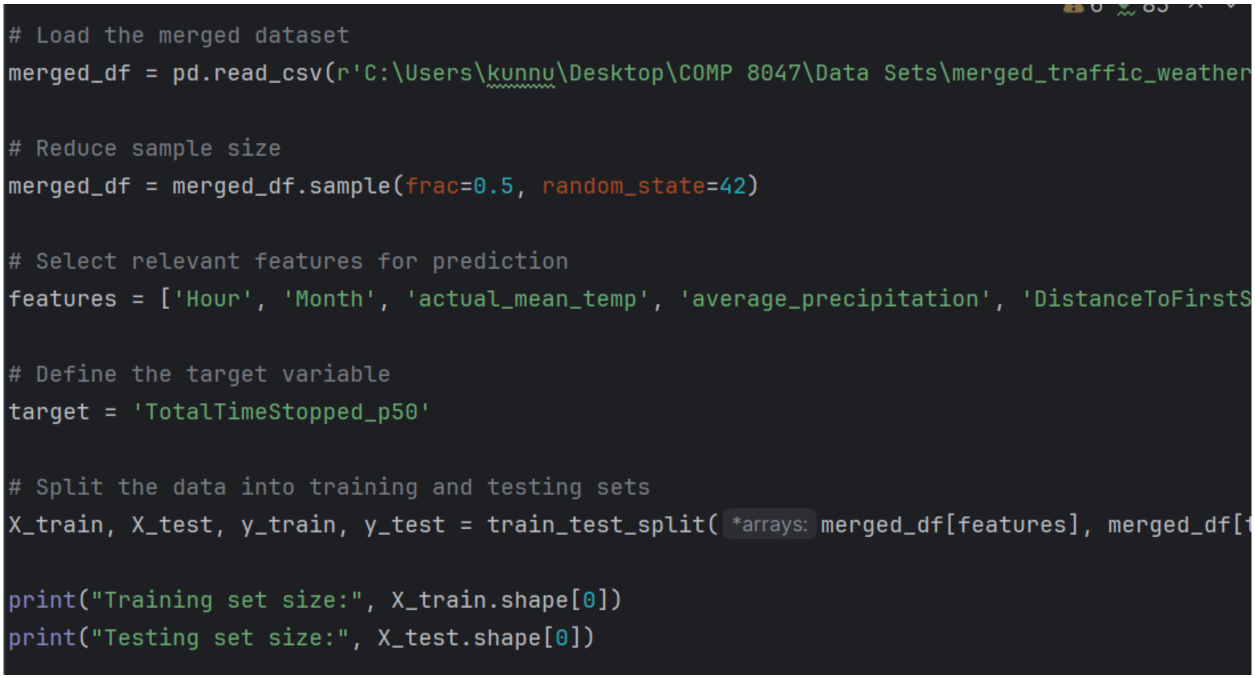
With this information included, the Predictive Analysis Report provides a comprehensive understanding of the dataset and the preprocessing steps undertaken before building predictive models.

Steps taken for Predictive Analysis:

- ***Loading Data***
  The predictive analysis starts by loading a dataset named merged_traffic_weather_data.csv using Pandas' read_csv function. This dataset likely contains merged traffic and weather data, combining information about traffic conditions and weather conditions.

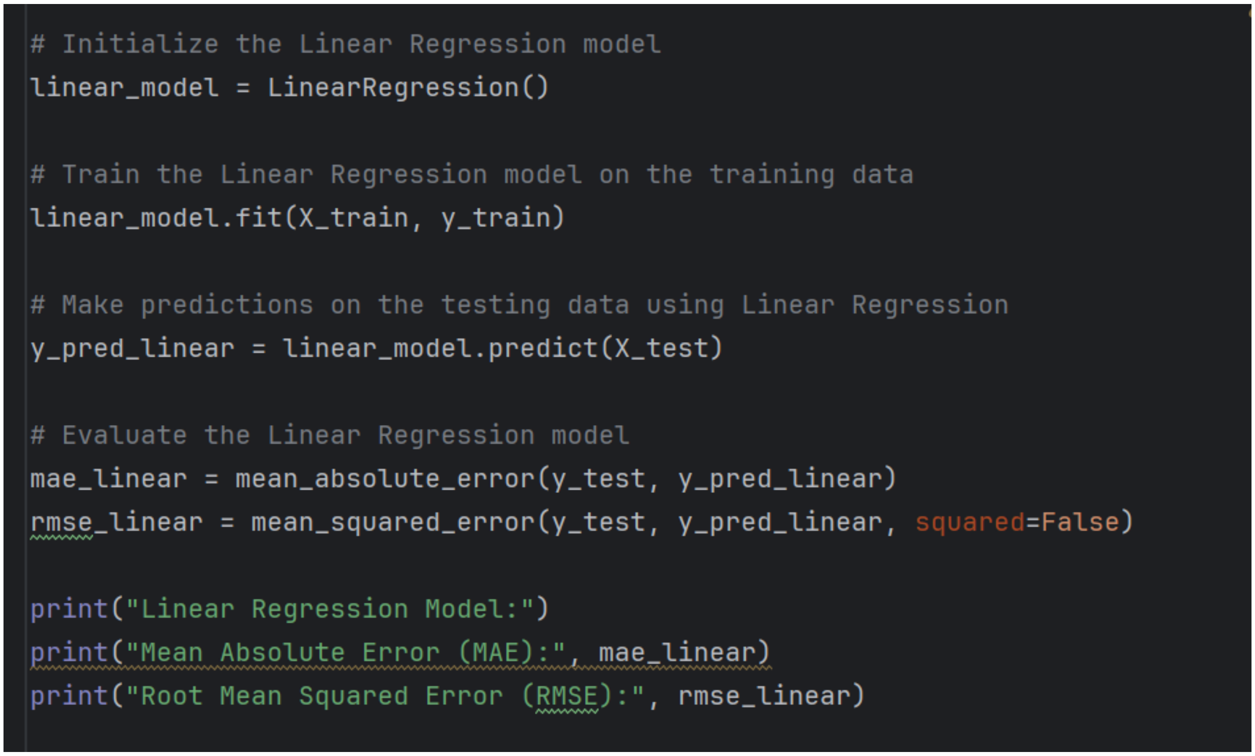
- ***Feature Selection and Target Definition***
  The relevant features for prediction are selected based on domain knowledge and analysis requirements. These features include:
  'Hour': The hour of the day when the data was recorded.
  'Month': The month in which the data was recorded.
  'actual_mean_temp': The actual mean temperature recorded at the time.
  'average_precipitation': The average precipitation recorded.
  'DistanceToFirstStop_p50': The distance to the first stop, representing traffic congestion.
  The target variable for prediction is defined as 'TotalTimeStopped_p50', which represents the total time for which 50% of the vehicles had to stop at an intersection.

- ***Data Splitting***
  The dataset is split into training and testing sets using the train_test_split function from Scikit-learn. This is done to assess the performance of the predictive models on unseen data.
  80% of the data is used for training (X_train, y_train), and 20% is kept aside for testing (X_test, y_test).



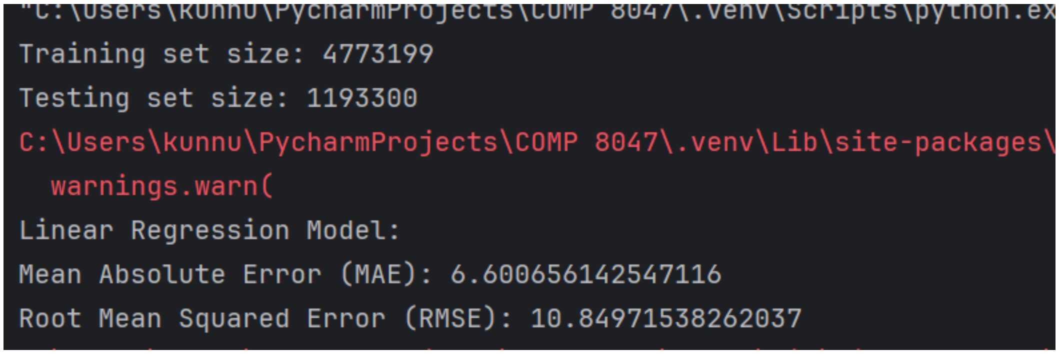
```
# Load the merged dataset
merged_df = pd.read_csv(r'C:\Users\kunnu\Desktop\COMP 8047\Data Sets\merged_traffic_weather

# Reduce sample size
merged_df = merged_df.sample(frac=0.5, random_state=42)

# Select relevant features for prediction
features = ['Hour', 'Month', 'actual_mean_temp', 'average_precipitation', 'DistanceToFirstS

# Define the target variable
target = 'TotalTimeStopped_p50'

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split( *arrays: merged_df[features], merged_df[t

print("Training set size:", X_train.shape[0])
print("Testing set size:", X_test.shape[0])
```

- ***Linear Regression Model***
  A linear regression model is initialized using LinearRegression from Scikit-learn.
  The model is trained on the training data using the fit method, where it learns the relationship between the selected features and the target variable.



```
# Initialize the Linear Regression model
linear_model = LinearRegression()

# Train the Linear Regression model on the training data
linear_model.fit(X_train, y_train)

# Make predictions on the testing data using Linear Regression
y_pred_linear = linear_model.predict(X_test)

# Evaluate the Linear Regression model
mae_linear = mean_absolute_error(y_test, y_pred_linear)
rmse_linear = mean_squared_error(y_test, y_pred_linear, squared=False)

print("Linear Regression Model:")
print("Mean Absolute Error (MAE):", mae_linear)
print("Root Mean Squared Error (RMSE):", rmse_linear)
```

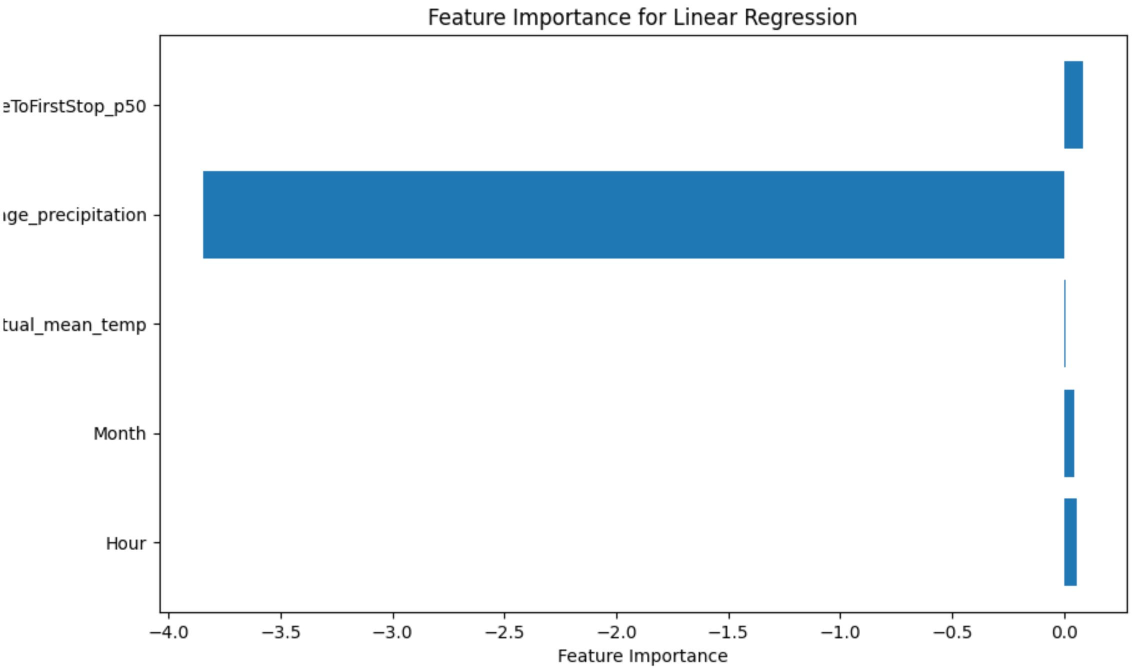
- ***Model Evaluation***
  Mean Absolute Error (MAE) and Root Mean Squared Error (RMSE) are calculated to evaluate the performance of the linear regression model on the testing data. These metrics provide insights into how well the model's predictions align with the actual values.



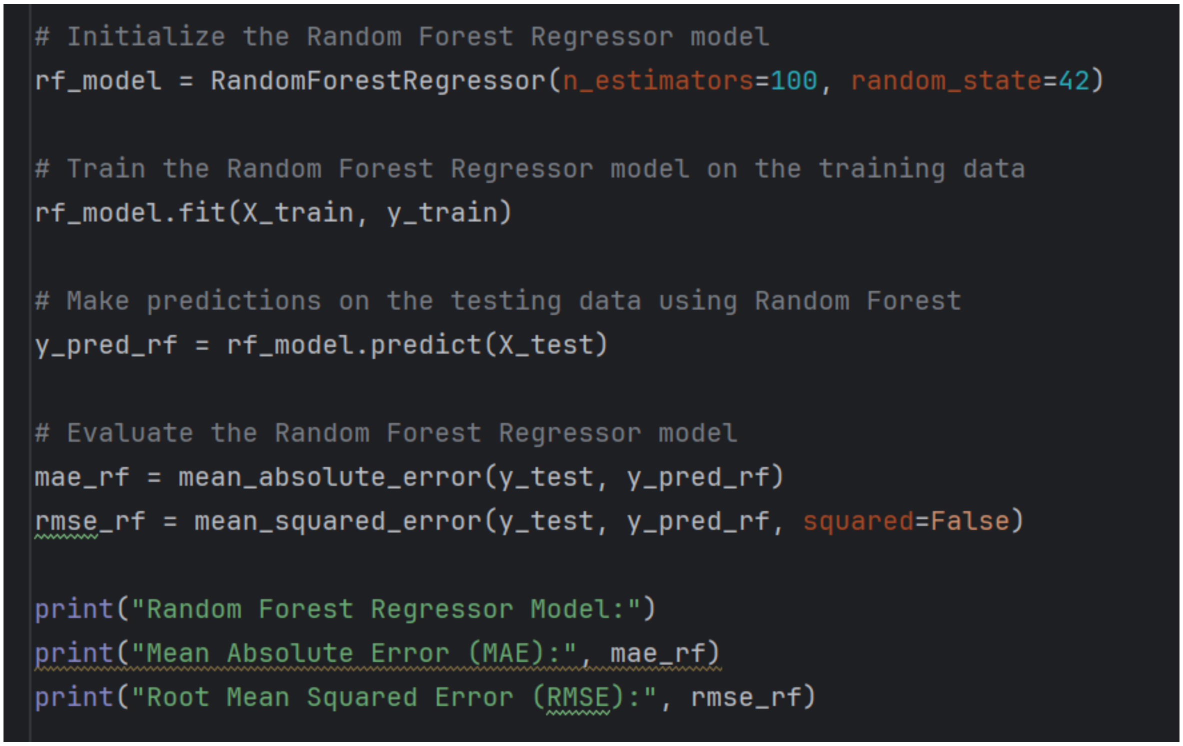
```
"C:\Users\kunnu\PycharmProjects\COMP 8047\.venv\Scripts\python.ex
Training set size: 4773199
Testing set size: 1193300
C:\Users\kunnu\PycharmProjects\COMP 8047\.venv\Lib\site-packages\
  warnings.warn(
Linear Regression Model:
Mean Absolute Error (MAE): 6.600656142547116
Root Mean Squared Error (RMSE): 10.84971538262037
```

- ***Prediction Example***
  An example prediction is made using the trained linear regression model for a new sample (new_sample). This sample contains values for the selected features.
  The predicted value represents the estimated total time stopped at an intersection for this scenario.

- ***Feature Importance Visualization***
  The importance of each feature in predicting the target variable ('TotalTimeStopped_p50') is visualized using a horizontal bar plot. This visualization helps in understanding which features have the most significant impact on the prediction.

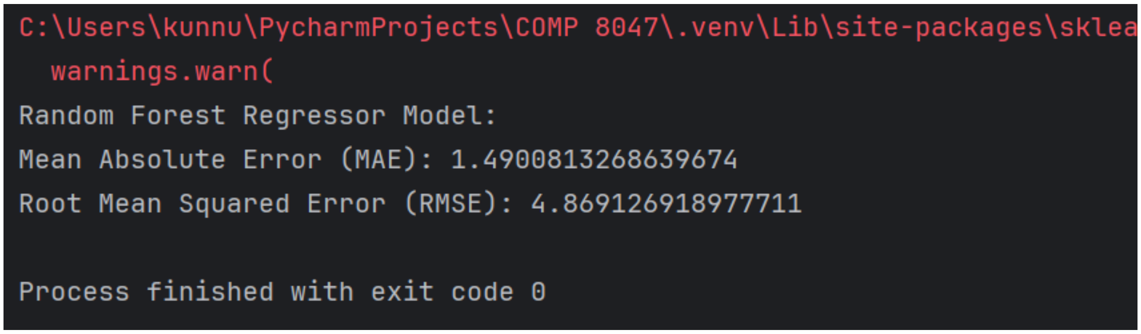


- ***Random Forest Regressor Model***
  A Random Forest Regressor model is initialized using RandomForestRegressor from Scikit-learn. The model is trained on the training data using the fit method. Random Forest is chosen as it can handle complex relationships between features and the target variable.



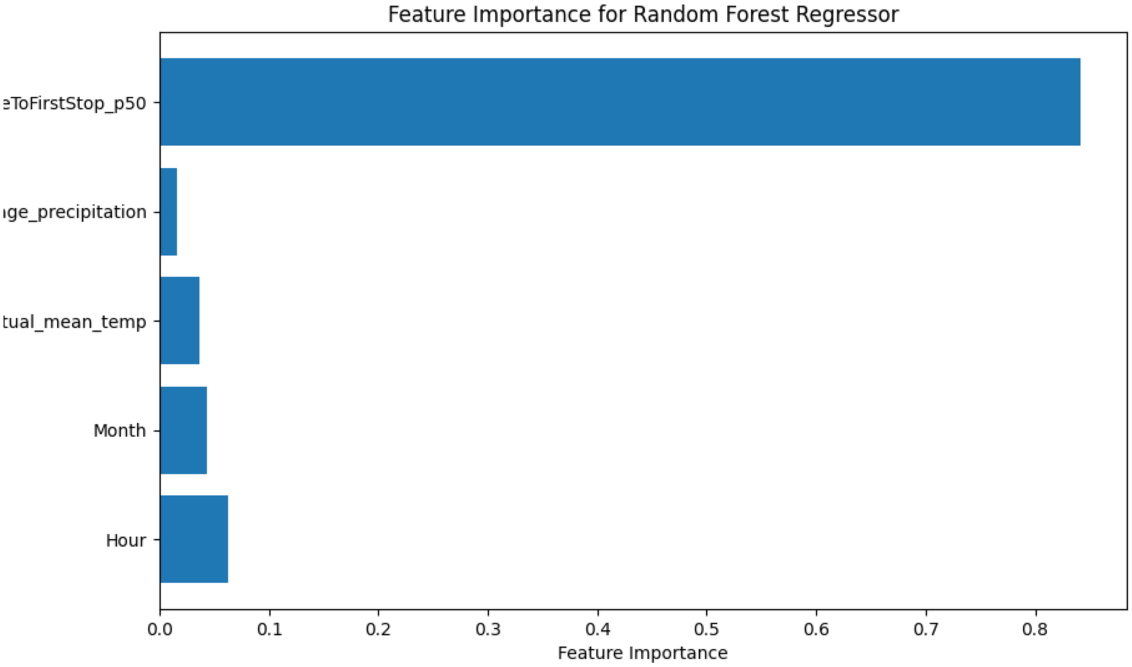
```
# Initialize the Random Forest Regressor model
rf_model = RandomForestRegressor(n_estimators=100, random_state=42)

# Train the Random Forest Regressor model on the training data
rf_model.fit(X_train, y_train)

# Make predictions on the testing data using Random Forest
y_pred_rf = rf_model.predict(X_test)

# Evaluate the Random Forest Regressor model
mae_rf = mean_absolute_error(y_test, y_pred_rf)
rmse_rf = mean_squared_error(y_test, y_pred_rf, squared=False)

print("Random Forest Regressor Model:")
print("Mean Absolute Error (MAE):", mae_rf)
print("Root Mean Squared Error (RMSE):", rmse_rf)
```

- ***Model Evaluation (Random Forest)***
  Mean Absolute Error (MAE) and Root Mean Squared Error (RMSE) are calculated to evaluate the performance of the Random Forest model on the testing data. This evaluation provides insights into the effectiveness of the Random Forest algorithm compared to linear regression.

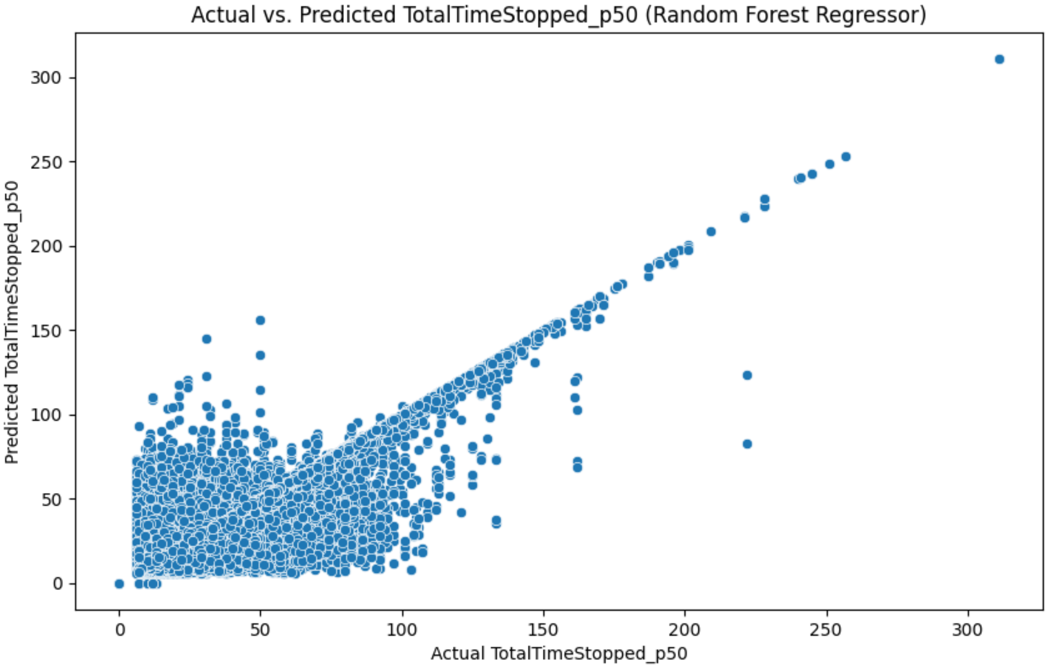


```
C:\Users\kunnu\PycharmProjects\COMP 8047\.venv\Lib\site-packages\sklea
  warnings.warn(
Random Forest Regressor Model:
Mean Absolute Error (MAE): 1.4900813268639674
Root Mean Squared Error (RMSE): 4.869126918977711

Process finished with exit code 0
```

- ***Scatter Plot Visualization***
  The predicted values versus the actual values of the target variable ('TotalTimeStopped_p50') are visualized using a scatter plot for the Random Forest Regressor model. This visualization helps in understanding the model's performance by comparing its predictions to the actual values.



And at the end Visualizing predicted versus actual TotalTimeStopped_p50 values using scatter plot for Random Forest



2.9 Methodology

The methodology adopted for the development of the "Traffic Congestion Reduction and Management System" entails a comprehensive and iterative approach that encompasses data collection, preprocessing, exploratory data analysis (EDA), feature engineering, model development, evaluation, and user engagement strategies. This section provides a detailed breakdown of each phase:

1. Data Collection and Preprocessing:

Real-Time Data Acquisition:

The system leverages real-time data sources, including the Google Maps API for traffic information and the OpenWeatherMap API for weather updates. Data retrieval mechanisms are implemented to ensure continuous access to up-to-date information.

Historical Data Retrieval:

Historical traffic and weather datasets are obtained from reliable sources, such as government repositories or open data platforms such as Kaggle. Data integrity checks are performed to ensure the accuracy and completeness of the historical records.

Data Cleaning Techniques:

Various data cleaning techniques are employed to address inconsistencies, missing values, and anomalies in the datasets. These techniques include:

- Missing Values Handling: Strategies such as imputation, deletion, or interpolation are applied to handle missing data in a manner that minimizes bias and preserves data integrity.
- Outlier Detection and Treatment: Statistical methods or domain knowledge-based approaches are used to identify and mitigate outliers that may adversely affect analysis results.
- Normalization and Standardization: Numerical features are scaled to a common range or standardized to ensure consistent data representation and facilitate model convergence.

2. Exploratory Data Analysis (EDA):

Descriptive Statistics:

Summary statistics, including measures of central tendency, dispersion, and distributional properties, are computed to characterize the datasets and identify potential trends or patterns.

Data Visualization:

Graphical techniques such as histograms, box plots, scatter plots, and heatmaps are utilized to visualize the distributions of variables, explore relationships between features, and uncover underlying patterns or correlations.

Correlation Analysis:

Correlation matrices and correlation heatmaps are generated to quantify the relationships between variables and identify significant predictors or multicollinearity issues.

3. Feature Engineering:

Feature Creation and Transformation:

New features are engineered based on domain knowledge, data insights from EDA, and domain-specific considerations. Feature transformation techniques, such as polynomial features, interaction terms, and logarithmic transformations, are applied to enhance the predictive power of the models.

Dimensionality Reduction:

Dimensionality reduction techniques, including Principal Component Analysis (PCA) or feature selection algorithms like Recursive Feature Elimination (RFE), are employed to reduce the dimensionality of the feature space and mitigate the curse of dimensionality.

Text and Categorical Data Processing:

Textual and categorical features are encoded using techniques such as one-hot encoding, label encoding, or embedding methods to convert them into numerical representations suitable for modeling.

4. Model Development and Evaluation:

Model Selection:

A diverse range of machine learning algorithms, including regression models, decision trees, ensemble methods (e.g., Random Forest, Gradient Boosting), and deep learning architectures, are considered for model development based on the problem complexity and dataset characteristics.

Model Training and Validation:

The datasets are split into training, validation, and test sets using appropriate sampling strategies (e.g., stratified sampling for imbalanced datasets). Models are trained on the training data and evaluated using cross-validation techniques to assess their generalization performance.

Performance Metrics:

Various performance metrics, such as accuracy, precision, recall, F1-score, ROC-AUC, and Mean Absolute Error (MAE), are computed to evaluate the models' predictive performance and assess their suitability for the intended application.

5. Model Optimization and Fine-Tuning:

Hyperparameter Tuning:

Hyperparameter optimization techniques, including grid search, random search, Bayesian optimization, or genetic algorithms, are employed to fine-tune the models and optimize their performance.

Ensemble Methods and Model Stacking:

Ensemble learning techniques, such as bagging, boosting, or model stacking, are explored to combine multiple base models and improve predictive accuracy, robustness, and generalization capability.

6. User Engagement Strategies:

Dashboard Development:

An interactive web-based dashboard is designed and developed to provide users with access to real-time traffic updates, weather forecasts, route planning functionalities, incident reporting features, and personalized recommendations.

7. Technologies to Be Used:

Languages:

The project primarily utilized Python for real-time data analysis and web development. Python offers a rich ecosystem of libraries and frameworks suited for various tasks involved in the project.

Libraries and Frameworks of Python:

- NumPy and Pandas: These libraries are used for efficient data manipulation and analysis, including preprocessing and feature engineering tasks.
- Flask is a micro web framework written in Python. It is lightweight and easy to use, making it ideal for developing web applications.
- The requests library is used to send HTTP requests to web servers and retrieve data from APIs.
- The datetime module provides classes for manipulating dates and times in Python.
- Matplotlib is a plotting library for Python, while Seaborn is a statistical data visualization library based on Matplotlib.
- Plotly is an interactive visualization library that allows for the creation of interactive and dynamic plots.
- Folium is a Python library used for creating interactive maps with Leaflet.js.
- Scikit-Learn is a machine learning library for Python that provides simple and efficient tools for data mining and data analysis.

These libraries and modules play crucial roles in various aspects of the project, including data retrieval, preprocessing, visualization, and machine learning model development. By leveraging these tools, the project can efficiently analyze traffic and weather data, develop predictive models, and visualize insights for effective decision-making and traffic management.

8. Web Development:

Frameworks like Flask are employed for web-based dashboard creation. Flask is a lightweight and versatile web framework that allows for rapid development of web applications, making it well-suited for creating interactive dashboards to visualize traffic and weather data.

HTML is the standard markup language for creating web pages. It provides the structure and content of a web page through a system of tags and attributes. In the context of the project, HTML is used to define the layout and structure of the dashboard interface. It includes elements such as headers, paragraphs, tables, forms, and containers to organize and display information to users.

CSS is a style sheet language used to define the presentation and appearance of HTML elements on a web page. It allows developers to customize the visual aspects of a website, including colors, fonts, layout, and animations. In the project, CSS is employed to style the HTML elements and improve the overall aesthetics of the dashboard. This includes setting fonts, colors, margins, padding, borders, and backgrounds to create a visually appealing and cohesive design.

JavaScript is a programming language that enables interactive and dynamic behavior on web pages. It allows developers to manipulate the HTML DOM (Document Object Model), handle events, and create interactive features such as animations, form validation, and real-time updates. In the context of the project, JavaScript is used to enhance the interactivity of the dashboard by implementing client-side functionality. This includes handling user interactions, such as clicks and inputs, and making asynchronous requests to the backend server to fetch data or perform actions without reloading the entire page.

By integrating HTML, CSS, and JavaScript with Flask, the project delivered a seamless and interactive dashboard experience for users. The combination of these technologies enables the creation of visually appealing interfaces, smooth navigation, and dynamic content updates, enhancing the overall usability and effectiveness of the traffic and weather visualization application.

9. Machine Learning:

Scikit-Learn:

Scikit-Learn is a powerful library for machine learning in Python, offering a wide range of tools for various tasks such as classification, regression, clustering, and dimensionality reduction. It provides a user-friendly interface for implementing machine learning algorithms and evaluating model performance. Scikit-Learn's extensive documentation and well-designed API make it easy for developers to build predictive models and perform tasks like feature selection, hyperparameter tuning, and cross-validation.

TensorFlow:

TensorFlow is an open-source machine learning framework developed by Google that offers scalability and flexibility for building and deploying machine learning models, particularly deep learning models. TensorFlow allows developers to define computational graphs and train complex neural networks efficiently using GPUs and distributed computing. It provides high-level APIs like Keras for building and training neural networks with ease. TensorFlow's extensive ecosystem includes tools for model serving, optimization, and deployment, making it suitable for a wide range of applications, from image recognition to natural language processing.

K-Means Clustering:

K-Means clustering is an unsupervised learning algorithm used for clustering or grouping data points into K clusters based on their similarity. In the context of the project, K-Means clustering is employed for incident detection by grouping traffic data points into clusters based on their spatial proximity. By analyzing the characteristics of each cluster, such as traffic density, speed, and direction, the algorithm can help identify regions with abnormal traffic patterns or incidents. K-Means clustering is computationally efficient and easy to implement, making it suitable for real-time analysis of traffic data.

Model Training Results:

Training set size: 4,773,199

Testing set size: 1,193,300

Linear Regression:

Mean Absolute Error (MAE): 6.60

Root Mean Squared Error (RMSE): 10.85

Random Forest Regressor:

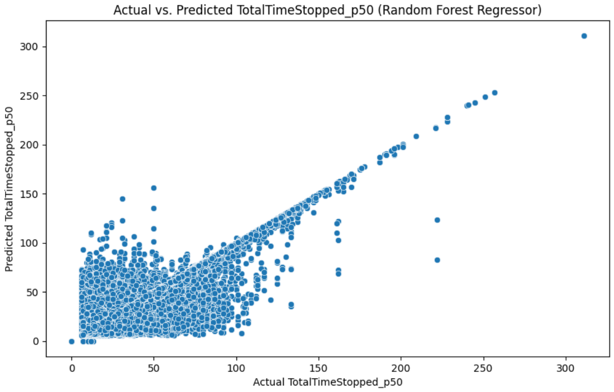
Mean Absolute Error (MAE): 1.49

Root Mean Squared Error (RMSE): 4.87

The Random Forest Regressor outperformed the Linear Regression model significantly, achieving lower MAE and RMSE. This indicates that the Random Forest model provides more accurate predictions of TotalTimeStopped_p50 compared to the Linear Regression model.

The feature importance analysis using the trained model revealed valuable insights into the predictive power of each feature. For instance, higher temperatures (`actual_mean_temp`) were associated with lower predicted TotalTimeStopped_p50, while higher precipitation levels (`average_precipitation`) were associated with higher predicted TotalTimeStopped_p50.

Additionally, the scatter plot visualizing the predicted versus actual TotalTimeStopped_p50 values for the Random Forest Regressor demonstrates the model's ability to capture the underlying patterns in the data and make accurate predictions.



Overall, the model training results indicate the effectiveness of the machine learning approach in predicting traffic congestion, which can be instrumental in developing a robust Traffic Congestion Reduction and Management System.Database

10. Predictive Analysis

The model was trained to predict traffic congestion, specifically focusing on predicting a metric called "TotalTimeStopped_p50."

Why the model was trained:

- Traffic congestion is a common problem in many cities, leading to delays, frustration, and economic losses. Predicting traffic congestion can help authorities and individuals make informed decisions to avoid or mitigate congestion.

- By analyzing historical data such as traffic flow, weather conditions, and time of day, machine learning models can learn patterns and relationships to make predictions about future traffic congestion.

- The goal of training the model is to develop a tool that can accurately forecast traffic congestion, enabling better planning, routing, and management of traffic flow in real-time.

What the model predicts:

- The model predicts a metric called "TotalTimeStopped_p50." This metric represents the estimated total time (in minutes) that vehicles are stopped or delayed at a particular location or segment of the road network.

- For example, if the model predicts a TotalTimeStopped_p50 of 10 minutes for a certain road segment during a specific time period, it means that, on average, vehicles traveling through that segment are expected to experience a 10-minute delay due to congestion or traffic incidents.

Result of prediction in layman's terms:

- Let's say you're planning to drive from point A to point B at a certain time of day. Before starting your journey, you check a traffic prediction app or dashboard that utilizes the trained machine learning model.

- The app provides you with an estimated TotalTimeStopped_p50 for your route. If the predicted value is low (e.g., 5 minutes), it indicates that traffic congestion is expected to be minimal, and you can expect a smooth and relatively quick journey.

- Conversely, if the predicted value is high (e.g., 30 minutes), it suggests that traffic congestion is likely to be significant, and you should anticipate delays along your route.

- Armed with this information, you can make decisions such as choosing an alternative route, adjusting your departure time, or using public transportation to avoid getting stuck in traffic.

In essence, the trained model empowers individuals and transportation authorities with insights into future traffic conditions, allowing them to take proactive measures to minimize the impact of congestion and improve overall traffic flow and efficiency.

11. Algorithms used:

1. Linear Regression Algorithm (from app.py):

- Linear regression is a statistical method used to model the relationship between a dependent variable and one or more independent variables.

- In this case, the algorithm predicts the TotalTimeStopped_p50 (dependent variable) based on various features such as hour, month, actual_mean_temp, and average_precipitation (independent variables).

- The model coefficients and intercept are estimated to minimize the difference between the actual and predicted TotalTimeStopped_p50 values.

Pseudo code:

```
# Initialize the Linear Regression model
model = LinearRegression()
# Train the model on the training data
model.fit(X_train, y_train)

# Make predictions on the testing data
y_pred = model.predict(X_test)
# Evaluate the model using metrics such as MAE and RMSE
mae = calculate_mean_absolute_error(y_test, y_pred)
```

```
rmse = calculate_root_mean_squared_error(y_test, y_pred)
# Display the model coefficients and intercept
coefficients = model.get_coefficients()
intercept = model.get_intercept()
# Visualize feature importance
visualize_feature_importance(features, coefficients)
# Display the model evaluation results
display_evaluation_results(mae, rmse)
# Initialize the Linear Regression model
model = LinearRegression()
# Train the model on the training data
model.fit(X_train, y_train)

# Make predictions on the testing data
y_pred = model.predict(X_test)
# Evaluate the model using metrics such as MAE and RMSE
mae = calculate_mean_absolute_error(y_test, y_pred)
rmse = calculate_root_mean_squared_error(y_test, y_pred)
# Display the model coefficients and intercept
coefficients = model.get_coefficients()
intercept = model.get_intercept()
# Visualize feature importance
visualize_feature_importance(features, coefficients)
# Display the model evaluation results
display_evaluation_results(mae, rmse)
```

2. Random Forest Regressor Algorithm (from predictive analysis code):

- Random Forest Regressor is an ensemble learning method that combines multiple decision trees to make predictions.

- Each decision tree is trained on a random subset of features and data points, and predictions are made by averaging the predictions of individual trees.

- The algorithm is used to predict TotalTimeStopped_p50 based on various features from the dataset.

Here's the pseudo code for the algorithm:

~~~
# Initialize the Random Forest Regressor model
rf_model = RandomForestRegressor(n_estimators=100, random_state=42)
# Train the model on the training data
rf_model.fit(X_train, y_train)
# Make predictions on the testing data
y_pred_rf = rf_model.predict(X_test)
# Evaluate the model using metrics such as MAE and RMSE
mae_rf = calculate_mean_absolute_error(y_test, y_pred_rf)
rmse_rf = calculate_root_mean_squared_error(y_test, y_pred_rf)
# Visualize predicted versus actual TotalTimeStopped_p50 values
visualize_predictions(y_test, y_pred_rf)
# Display the model evaluation results
display_evaluation_results(mae_rf, rmse_rf)
```
~~~

3. Data Cleaning and Preprocessing:

Data cleaning and preprocessing tasks are performed implicitly within the functions like get_weather_data, get_traffic_data, and generate_graphs.

These functions handle API requests to fetch weather and traffic data, which may involve data cleaning steps to handle missing or inconsistent data.

generate_graphs function preprocesses the dataset before generating visualizations, such as selecting relevant columns, grouping data, and calculating correlations.

4. K-Means Clustering Algorithm (inside generate_graphs function):

K-Means clustering is used to cluster traffic data points based on their spatial proximity.

The algorithm segments the data into a specified number of clusters (e.g., 100 clusters) by minimizing the within-cluster variance.

After clustering, a Folium map is generated to visualize the clusters on a map of Philadelphia.

Here's a summary of the pseudocode for the additional algorithms:

K-Means Clustering Algorithm (Pseudo code):

```
# Load the dataset
X = load_dataset()
# Perform K-means clustering
kmeans = KMeans(n_clusters=100, random_state=42)
cluster_labels = kmeans.fit_predict(X)
# Visualize clusters on a map
generate_folium_map(cluster_labels)
```

These pseudocode snippets outline the steps involved in performing K-Means clustering on the traffic data to identify spatial patterns and visualize the clusters on a map.

By leveraging a combination of technologies, development processes, design patterns, and algorithms, the "Traffic Congestion Reduction and Management System" aims to provide actionable insights and effective solutions for managing urban traffic congestion. Through iterative development, user-centric design, and continuous improvement, the project strives to address the complex challenges of traffic management and enhance the overall efficiency and sustainability of transportation systems.

2.10 Implications of the Implementation

This section of the report will delve into various aspects of the implementation, including performance, functionality, security, usability, scalability, maintainability, and associated limitations or challenges. Here's a detailed breakdown:

Performance:

By conducting thorough performance evaluations of API integration, data processing, and graph generation, stakeholders can gain insights into the efficiency, scalability, and potential bottlenecks of the application. This analysis will inform optimization efforts aimed at improving the overall responsiveness and user experience of the application.

- API Integration: The code integrates with external APIs such as OpenWeatherMap and Google Maps Distance Matrix API to fetch weather data, real-time traffic data, alternative routes, traffic incidents, and public transit data. Assess the performance impact of these API calls on the overall responsiveness of the application.

- OpenWeatherMap API: Evaluate the performance impact of fetching weather data from the OpenWeatherMap API. Measure the latency of API requests and responses, considering factors such as network latency and server response time. Assess the reliability and availability of weather data retrieved from the API, accounting for potential downtime or service interruptions.
- Google Maps Distance Matrix API: Analyze the performance of fetching real-time traffic data, alternative routes, traffic incidents, and public transit data from the Google Maps Distance Matrix API. Measure the response time for various API endpoints and assess the scalability of API calls to handle concurrent user requests. Evaluate the consistency and accuracy of traffic and transit information retrieved from the API, considering factors like data freshness and relevance.
- Data Processing: Analyze the efficiency of data processing operations, such as parsing JSON responses from API calls, calculating durations, and handling large datasets for generating insights.
- JSON Parsing: Evaluate the efficiency of parsing JSON responses from API calls using built-in Python libraries or third-party packages. Measure the time taken to extract relevant data fields from JSON payloads and convert them into Python data structures. Assess the memory consumption and CPU utilization during JSON parsing operations, especially when handling large volumes of data.
- Duration Calculation: Analyze the computational complexity of calculating durations, such as travel time and duration in traffic, based on data retrieved from API responses. Evaluate the performance of algorithms used for time calculations and assess their scalability to handle varying input parameters and data formats.
- Large Dataset Handling: Assess the efficiency of handling large datasets for generating traffic and weather insights. Measure the memory usage and processing time required to aggregate, filter, and analyze large volumes of data collected over time. Evaluate the scalability of data processing algorithms to accommodate increasing dataset sizes without compromising performance.
- Graph Generation: Evaluate the performance of graph generation functions using libraries like Matplotlib, Seaborn, Plotly, and Folium. Consider the time taken to generate various types of graphs and maps based on traffic and weather data.
- Matplotlib and Seaborn: Evaluate the performance of generating static graphs using libraries like Matplotlib and Seaborn. Measure the time taken to plot various types of graphs, such as bar charts, line plots, and histograms, based on traffic and weather data. Assess the memory consumption and CPU usage during graph generation, especially when rendering complex visualizations with large datasets.
- Plotly and Folium: Analyze the performance of generating interactive plots and maps using libraries like Plotly and Folium. Measure the time taken to render dynamic visualizations with interactive features, such as zooming, panning, and tooltips. Evaluate the responsiveness of interactive plots and maps to user interactions, considering factors like rendering speed and data loading time.

Functionality:

By conducting thorough functionality assessments of weather data retrieval, traffic insights, and public transit information, stakeholders can ensure that the application meets user expectations and delivers valuable services for planning travel routes and optimizing commute experiences.

- Weather Data Retrieval: Assess the functionality of fetching weather data based on user-provided city names. Evaluate the accuracy and reliability of weather information retrieved from the OpenWeatherMap API.
- Accuracy and Reliability: Evaluate the functionality of fetching weather data from the OpenWeatherMap API based on user-provided city names. Verify the accuracy and reliability of weather information, including temperature, weather conditions, wind speed, humidity, visibility, sunrise time, and sunset time. Conduct comparative analysis against other weather forecasting sources to validate the consistency of weather predictions.
- Error Handling: Assess the application's ability to handle errors gracefully when fetching weather data. Evaluate error messages and notifications presented to users in case of invalid city names, network connectivity issues, or API service errors. Verify that users receive informative feedback and instructions for resolving errors or updating their input.
- Traffic Insights: Evaluate the functionality of fetching real-time traffic data, alternative routes, and traffic incidents using the Google Maps Distance Matrix API. Verify the correctness and relevance of traffic information presented to users.
- Real-time Traffic Data: Evaluate the functionality of fetching real-time traffic data from the Google Maps Distance Matrix API. Assess the accuracy and freshness of traffic information, including travel duration, congestion levels, and estimated arrival times. Compare the predicted travel time with actual travel experiences reported by users to validate the reliability of traffic insights.
- Alternative Routes: Analyze the functionality of fetching alternative routes based on user-provided origin and destination addresses. Evaluate the diversity and efficiency of alternative routes presented to users, considering factors such as travel time, distance, and traffic conditions. Verify that users can select preferred routes based on their preferences and constraints.
- Traffic Incidents: Assess the functionality of fetching traffic incidents near user-specified locations. Evaluate the timeliness and relevance of traffic incident reports, including accidents, road closures, construction zones, and other disruptions. Verify that users receive timely notifications and alerts about potential traffic hazards along their planned routes.
- Public Transit Information: Analyze the functionality of fetching public transit data and presenting transit routes to users. Evaluate the usability and comprehensiveness of public transit information retrieved from the Google Maps Directions API.
- Transit Data Retrieval: Analyze the functionality of fetching public transit data from the Google Maps Directions API. Evaluate the comprehensiveness and accuracy of transit information, including transit routes, schedules, fares, and transfer options. Verify that users receive up-to-date and reliable information about available public transit services in their area.

- Route Presentation: Evaluate the functionality of presenting transit routes to users in a clear and intuitive manner. Assess the usability of route visualization tools, such as maps, directions, and step-by-step instructions. Verify that users can easily understand and follow transit routes, including walking directions to and from transit stops.

Security:

By addressing these security considerations related to API key management and data transmission, stakeholders can enhance the overall security posture of the application and mitigate risks associated with unauthorized access, data breaches, and security vulnerabilities.

- API Key Management: Assess the security of API key management for accessing external services like OpenWeatherMap and Google Maps APIs. Ensure that sensitive API keys are securely stored and managed to prevent unauthorized access or misuse.
- Secure Storage: Assess how API keys for services like OpenWeatherMap and Google Maps APIs are managed within the application. Verify that sensitive information, such as API keys, is securely stored and not directly exposed in the source code or configuration files. Utilize secure methods for storing sensitive information, such as environment variables, configuration files with restricted access, or secure key management services.
- Access Control: Evaluate the access control mechanisms implemented for managing API keys. Ensure that only authorized personnel or components have access to API keys and that access permissions are strictly enforced. Implement role-based access control (RBAC) or similar mechanisms to restrict access to sensitive resources based on user roles and privileges.
- Key Rotation: Assess the key rotation practices for API keys to mitigate the risk of key exposure and unauthorized access. Implement periodic key rotation procedures to generate new API keys and revoke old keys. Ensure that key rotation processes are automated, auditable, and well-documented to maintain security hygiene.
- Data Transmission: Evaluate the security of data transmission between the application server and external APIs. Ensure that HTTPS protocols are used to encrypt data during transit to protect against eavesdropping and tampering.

Usability:

By focusing on usability aspects such as user interface design, accessibility, and error handling, stakeholders can enhance the overall user experience of the web application and improve user satisfaction, engagement, and retention. Conducting usability testing with representative users can provide valuable insights into usability issues and opportunities for improvement.

- User Interface: Evaluate the usability of the web application interface provided by Flask templates. Assess the clarity, intuitiveness, and accessibility of user interactions for fetching traffic and weather insights.

- Clarity and Intuitiveness: Evaluate the clarity and intuitiveness of the web application interface provided by Flask templates. Assess the layout, design, and organization of elements such as input forms, buttons, and navigation menus. Ensure that the interface follows established UX/UI design principles to provide a seamless and intuitive user experience. Consider factors such as visual hierarchy, consistency, and responsiveness to different screen sizes and devices.
- Accessibility: Analyze the accessibility features implemented in the web application interface to ensure inclusivity for users with disabilities. Verify compliance with accessibility standards, such as Web Content Accessibility Guidelines (WCAG), to enhance accessibility features such as keyboard navigation, screen reader compatibility, and alternative text for images. Conduct usability testing with assistive technologies and users with disabilities to identify and address any usability barriers or accessibility issues.
- Error Handling: Analyze the effectiveness of error handling mechanisms for notifying users about invalid inputs, failed API requests, or other errors encountered during data retrieval and processing.
- Input Validation: Evaluate the effectiveness of input validation mechanisms for detecting and handling invalid user inputs. Verify that the application validates user inputs, such as city names, origin-destination addresses, and desired arrival times, to prevent submission of invalid or malicious data. Provide clear and descriptive error messages to users indicating the nature of the input error and instructions for correcting it.
- API Error Handling: Analyze the error handling mechanisms implemented for handling API errors, network failures, and other technical issues encountered during data retrieval from external APIs. Ensure that the application gracefully handles API errors, such as HTTP status codes indicating server errors, rate limiting, or authentication failures. Provide informative error messages to users and log detailed error information for debugging and troubleshooting purposes.
- Feedback Mechanisms: Assess the feedback mechanisms implemented to notify users about the status of their requests and actions. Provide real-time feedback to users, such as loading indicators, progress bars, or success/error messages, to indicate the progress and outcome of API requests and data processing operations. Ensure that feedback messages are clear, concise, and contextually relevant to help users understand the system's response and take appropriate actions.

Scalability and Maintainability:

By focusing on scalability and maintainability aspects such as code structure, modularity, readability, and data handling, stakeholders can ensure that the Flask application remains flexible, robust, and scalable to meet evolving requirements and handle increasing volumes of traffic and weather data effectively. Regular code reviews, refactoring, and performance tuning can help maintain the codebase's quality and agility over time.

- Code Structure: Evaluate the modularity, readability, and maintainability of the Flask application code. Assess the adherence to best practices and design patterns to facilitate future updates, enhancements, and bug fixes.

- Modularity: Evaluate the modularity of the Flask application codebase, including the organization of code into reusable components, such as functions, classes, and modules. Assess the separation of concerns to ensure that different aspects of the application, such as data retrieval, processing, and presentation, are encapsulated in distinct modules with well-defined interfaces.
- Readability: Assess the readability of the codebase by considering factors such as variable naming conventions, code comments, and documentation. Ensure that the code is self-explanatory and easy to understand for developers who may need to maintain or extend the application in the future.
- Maintainability: Analyze the maintainability of the codebase in terms of its ability to accommodate future updates, enhancements, and bug fixes. Evaluate the flexibility and extensibility of the codebase to support changes in requirements or technology stack. Assess the ease of integrating new features or functionality without introducing regressions or breaking existing functionality.
- Data Handling: Analyze the scalability of data handling operations, such as caching, filtering, and aggregation, to accommodate increasing volumes of traffic and weather data without compromising performance or reliability.
- Caching: Evaluate the use of caching mechanisms to improve the scalability and performance of data retrieval operations. Implement caching strategies, such as memorization or caching HTTP responses, to reduce the overhead of repeated API calls for frequently accessed data. Monitor cache utilization and expiration to ensure data consistency and freshness.
- Filtering and Aggregation: Analyze the scalability of data handling operations, such as filtering and aggregation, to efficiently process large volumes of traffic and weather data. Optimize data retrieval queries and processing algorithms to minimize resource consumption and response times. Consider using database indexes, query optimization techniques, and parallel processing to improve scalability and throughput.

Limitations and Challenges:

By addressing these limitations and challenges associated with API rate limits, data accuracy, and handling heavy historical data, stakeholders can gain a comprehensive understanding of the potential constraints and considerations affecting the implementation of the application. This analysis will help inform decision-making regarding optimization strategies, risk mitigation measures, and resource allocation to maximize the effectiveness and reliability of the application in real-world scenarios.

- API Rate Limits: Discuss any limitations or challenges imposed by API rate limits, quota restrictions, or usage policies of external service providers. Consider strategies for optimizing API usage and minimizing the impact of rate limiting on application functionality.
- Impact on Application Functionality: Discuss the impact of API rate limits, quota restrictions, or usage policies of external service providers on the application's functionality. Rate limits imposed by APIs can constrain the frequency and volume of requests that the application can make within a given time frame, potentially affecting the responsiveness and reliability of data retrieval operations.

- Optimization Strategies: Consider strategies for optimizing API usage to mitigate the impact of rate limiting on application functionality. This may include implementing caching mechanisms to reduce the number of API calls, optimizing data retrieval queries to minimize request overhead, and prioritizing critical API endpoints to ensure timely access to essential data.
- Data Accuracy: Address potential limitations in data accuracy or completeness, especially when relying on third-party APIs for weather, traffic, and transit information. Discuss the implications of inaccurate or outdated data on the reliability of application insights.
- Reliability of Third-Party Data: Address potential limitations in the accuracy or completeness of data obtained from third-party APIs, such as OpenWeatherMap and Google Maps Distance Matrix API. Acknowledge that reliance on external data sources introduces the risk of encountering inaccuracies, inconsistencies, or outdated information, which can undermine the reliability of application insights.
- Implications of Inaccurate Data: Discuss the implications of inaccurate or outdated data on the reliability of application insights and user experience. Inaccurate weather forecasts, traffic conditions, or transit information can lead to erroneous recommendations or decisions, affecting user trust and satisfaction with the application.
- Handling Heavy data: Handling heavy historical data presents several challenges, particularly in terms of processing, cleaning, visualization, and sampling. This involves leveraging computational resources efficiently, employing appropriate data processing techniques, and selecting suitable visualization and sampling methods to derive meaningful insights from large datasets while mitigating the impact of resource constraints and processing limitations.

Processing and Cleaning:

- Computational Resources: Large volumes of historical data require substantial computational resources for processing and cleaning. Tasks such as data parsing, transformation, and quality assurance may demand significant processing power and memory allocation to ensure efficient handling of the dataset.
- Time-Intensive Operations: Processing and cleaning operations can be time-consuming, especially when dealing with extensive datasets. Tasks such as data normalization, deduplication, and error correction may take a considerable amount of time to execute, delaying the overall data preparation process.
- Trade-offs: When processing and cleaning historical data, there's often a trade-off between data quality and processing time. Striking the right balance between thorough data cleansing and timely data preparation is essential to ensure that the resulting data set is accurate, reliable, and suitable for analysis.

Visualization and Sampling:

- Resource-Intensive Visualization: Visualizing extensive historical data can strain computational resources and prolong visualization rendering times. Graphs, charts, and maps generated from large datasets may require significant processing power and memory, impacting the responsiveness and usability of the visualization interface.
- Sampling Techniques: To expedite visualization without compromising accuracy significantly, sampling techniques can be employed to select representative subsets of the data for

visualization. Random sampling, systematic sampling, or stratified sampling methods can help reduce the dataset size while preserving the overall data distribution and characteristics.

- Importance of Representative Samples: When employing sampling techniques, it's crucial to ensure that the selected samples are representative of the entire data set. Biased or unrepresentative samples may skew insights and conclusions drawn from visualization, leading to inaccurate or misleading interpretations of the data.
- Consideration of Implications: It's essential to consider the implications of data reduction on the reliability of insights derived from historical data. While sampling can expedite visualization and analysis, it may also introduce sampling errors or overlook critical patterns and outliers present in the full dataset. Evaluating the trade-offs between data reduction and insight accuracy is essential to make informed decisions about visualization strategies.

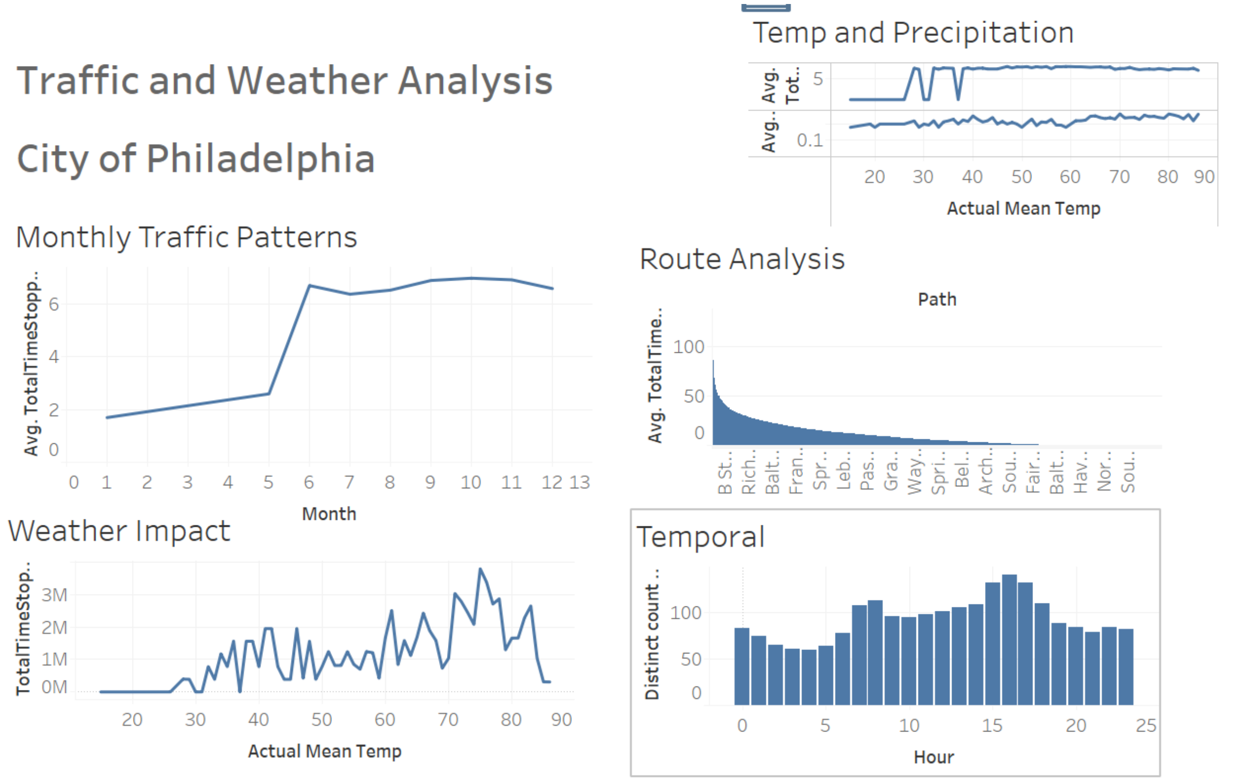
In summary, a comprehensive assessment of the implications of the implementation provides valuable insights into the application's strengths, weaknesses, and areas for improvement. By leveraging these insights, stakeholders can make informed decisions, prioritize enhancements, and drive continuous improvement to deliver a robust, reliable, and user-centric solution for traffic and weather insights.

2.11 Research on the Use of New Technologies

In the process of developing the application and conducting research, several new technologies, tools, and methodologies were explored. These explorations yielded valuable insights into emerging trends, innovative solutions, and potential areas for further investigation. Here's a breakdown of the key findings and observations:

- API Integration: Through research and practical application, a deeper understanding of API integration was gained. Various APIs, such as OpenWeatherMap and Google Maps Distance Matrix API, were utilized to fetch weather data, real-time traffic information, alternative routes, and public transit data. This experience highlighted the importance of effective API selection, proper authentication, and error handling mechanisms to ensure seamless data retrieval and processing.
- Data Visualization Libraries: Experimentation with data visualization libraries such as Matplotlib, Seaborn, Plotly, and Folium provided insights into their capabilities and limitations for generating graphs, charts, and maps. Each library offers unique features and functionalities, enabling the creation of informative and visually appealing visualizations to convey insights derived from traffic and weather data.
- Tableau Integration: Integration of Tableau into the project brought a significant enhancement in data visualization capabilities, enabling stakeholders to gain deeper insights from the analyzed data through interactive dashboards and visualizations. Here's a detailed expansion on the integration with Tableau and APIs:
- Advanced Data Visualization with Tableau: Tableau's powerful visualization tools allowed for the creation of dynamic and interactive dashboards that presented complex data in an easily understandable format. The integration with Tableau enabled the generation of various types of charts, graphs, maps, and other visual elements to represent data trends, patterns, and correlations effectively. Stakeholders could interact with the dashboards, filter data, drill down

into specific details, and explore different perspectives of the data, facilitating better decision-making and understanding of the analyzed insights.



The picture shown above shows the Visualization done in Tableau for the Traffic and Weather Analysis for City of Philadelphia (saved in folder as City-Dashboard.twb)

- Web-Based Dashboard with Flask: Flask, a lightweight and flexible Python web framework, was utilized to develop a web-based dashboard that served as the interface for accessing and interacting with the analyzed insights. The dashboard, powered by Flask, provided stakeholders with a user-friendly platform to explore the visualizations and findings derived from Tableau and other analysis tools. Flask facilitated the integration of Tableau dashboards into the web application, ensuring seamless navigation and accessibility for users across different devices and platforms.
- Python Scripting for UI/UX Enhancement: Python scripting was employed to enhance the user interface (UI) and user experience (UX) of the web-based dashboard. Custom Python scripts were developed to add interactive elements, dynamic content updates, and personalized features to the dashboard, improving usability and engagement for stakeholders. Python's flexibility and versatility allowed for the creation of tailored solutions to meet specific UI/UX requirements and preferences, enhancing the overall effectiveness of the dashboard in conveying insights and facilitating decision-making processes.

- Machine Learning: While not extensively utilized in the current implementation, research into machine learning algorithms and techniques revealed their potential applications for traffic prediction, anomaly detection, and optimization. Exploring supervised and unsupervised learning approaches could enhance the application's capabilities to provide personalized recommendations and predictive insights based on historical traffic patterns and weather conditions.
- Geospatial Analysis: Delving into geospatial analysis techniques and libraries provided valuable insights into spatial data processing, visualization, and clustering. Techniques such as K-means clustering were explored to analyze traffic patterns, identify congestion hotspots, and optimize route planning based on geographic data.
- Model evaluation and optimization: These were pivotal aspects of the project, ensuring that the developed models could accurately predict outcomes and provide valuable insights. Here's an expanded view on the strategies employed for model evaluation and optimization:
- Rigorous Evaluation Metrics: The evaluation of model performance involved the use of robust evaluation metrics such as mean absolute error (MAE), root mean square error (RMSE), and R-squared. These metrics provided comprehensive insights into the accuracy, precision, and goodness-of-fit of the models, enabling stakeholders to assess their effectiveness in predicting target variables.
- Hyperparameter Tuning: To enhance model performance and generalization capabilities, hyperparameter tuning techniques were employed. Grid search and random search methods were utilized to systematically explore the hyperparameter space and identify optimal combinations that minimized error metrics or maximized model performance. By fine-tuning hyperparameters such as learning rates, regularization parameters, and tree depths, the models were optimized to achieve better predictive accuracy and robustness across different datasets.

- Feature Selection: Feature selection played a crucial role in refining the models and improving their interpretability and efficiency. Through feature selection techniques such as recursive feature elimination (RFE), feature importance ranking, and domain knowledge-based selection, irrelevant or redundant features were identified and removed from the input data. This streamlined the model input space, reducing noise and complexity while retaining the most informative features for predictive modeling. As a result, the optimized models could focus on relevant input variables, leading to improved prediction accuracy and reduced overfitting.
- Algorithm Selection: Experimentation with various algorithms and model architectures enabled the identification of the most suitable approach for the problem domain. Different algorithms, including regression models, decision trees, ensemble methods, and neural networks, were evaluated based on their performance, scalability, and interpretability. By comparing the strengths and weaknesses of each algorithm and assessing their performance on validation datasets, the optimal algorithm for the specific prediction task was selected. This process ensured that the chosen model architecture effectively captured the underlying patterns in the data and produced reliable predictions with minimal bias and variance.
- Feature engineering: Feature Engineering played a pivotal role in refining the input data and enhancing the predictive power of machine learning models. Here's an expanded overview of

the feature engineering techniques employed in the project, along with insights into the machine learning model development process:

Feature Engineering Techniques:

One-Hot Encoding: One-hot encoding was applied to categorical variables to transform them into numerical representations suitable for modeling. This technique expanded categorical features into binary vectors, with each category represented by a binary flag. By converting categorical variables into numerical format, the models could effectively incorporate categorical data into predictive algorithms without introducing ordinality.

Feature Scaling: Feature scaling techniques such as standardization or normalization were employed to ensure uniformity in the scale of numerical features. Standardization rescaled features to have a mean of zero and a standard deviation of one, while normalization scaled features to a range between zero and one. By standardizing or normalizing numerical features, biases towards features with larger magnitudes were mitigated, leading to improved model convergence and performance.

Polynomial Features: Polynomial feature generation was utilized to capture non-linear relationships between variables. This technique involved creating new features by taking polynomial combinations of the original features, up to a specified degree. By introducing polynomial features, the feature space was enriched, allowing models to capture complex interactions and non-linear patterns in the data, thereby enhancing model flexibility and predictive accuracy.

Machine Learning Model Development: The project involved the development of machine learning models to predict traffic patterns and analyze the impact of weather on traffic conditions. Various algorithms and techniques provided by the scikit-learn (sklearn) library were explored to build predictive models:

Linear Regression:

Linear regression was employed for predicting continuous variables such as traffic volume based on weather conditions and other factors. Serving as a baseline model, linear regression provided simple yet interpretable predictions by modeling linear relationships between input features and target variables.

Random Forest:

Random Forest was utilized for both regression and classification tasks, offering robust predictions and the ability to handle non-linear relationships in the data. By constructing multiple decision trees and aggregating their predictions, Random Forest provided reliable forecasts, making it suitable for complex traffic prediction scenarios where non-linear interactions exist among features.

Gradient Boosting:

Gradient Boosting techniques were adopted to improve model performance through ensemble learning. By sequentially training weak learners to correct the errors of previous models, Gradient Boosting combined multiple weak learners to create a strong predictive model. This approach enhanced model accuracy and generalization capabilities, making it effective for capturing complex relationships and producing accurate forecasts.

By leveraging feature engineering techniques and exploring diverse machine learning algorithms, the project successfully developed predictive models capable of capturing intricate patterns in traffic and weather data, thereby enabling stakeholders to make informed decisions based on reliable forecasts and analyses.

2.12 Future Enhancements

In the future, several enhancements could be made to further improve the functionality and performance of the project. Some potential future enhancements include:

- Real-Time Traffic Updates: Integrate additional data sources like traffic cameras, social media feeds, or crowd-sourced information to provide more accurate and real-time traffic updates. This would enhance the system's responsiveness to changing traffic conditions.
- Advanced Predictive Models: Develop more sophisticated machine learning models, including deep learning algorithms or ensemble methods, to improve the accuracy of traffic predictions. These models can capture complex patterns in traffic data for better forecasting.
- Personalized Route Recommendations: Implement a feature that offers personalized route recommendations based on individual user preferences and historical travel data. This would enhance the user experience by providing tailored suggestions for the most efficient routes.
- Dynamic Incident Handling: Enhance the system's ability to respond dynamically to traffic incidents in real-time. Integration with emergency response systems could provide timely and accurate information to users affected by accidents or road closures.
- Enhanced User Interface: Continuously improve the user interface to make it more intuitive, visually appealing, and user-friendly. Incorporating user feedback, conducting usability testing, and implementing design best practices would contribute to this enhancement.
- Optimization for Scalability: Architect the system to be more scalable and resilient to handle increased user traffic and data volume. This involves optimizing database queries, implementing caching mechanisms, and leveraging cloud-based infrastructure for scalability.
- Integration with Public Transportation: Expand the project's scope to include integration with public transportation systems, offering users comprehensive multi-modal route planning options. This would include buses, trains, and ride-sharing services for a seamless travel experience.
- Weather Impact Analysis: Incorporate weather data into the traffic prediction models to analyze its impact on traffic patterns. This enhancement would improve prediction accuracy and help users better plan their travel routes based on weather conditions.
- Enhanced Data Visualization: Utilize advanced data visualization techniques like Plotly, D3.js, or WebGL to present insights from the data in a more interactive and insightful manner. Dynamic and engaging visualizations would enhance user understanding of traffic trends.

- Mobile Application Development: Develop a mobile application version of the project to provide users with on-the-go access to traffic information and route planning capabilities. Features such as push notifications, voice navigation, and offline mode support would enhance the mobile user experience.
- Integration with Food Delivery and Ridesharing Apps: In the future, consider collecting and integrating data from food delivery drivers and ridesharing apps to cater to their needs. Insights such as nearby restaurants, food delivery hotspots, popular pickup and drop-off locations, as well as points of interest like parks and recreational areas, can be valuable for these users. By incorporating this data, the system can provide tailored recommendations and route optimizations for food delivery drivers and rideshare operators, enhancing their efficiency and overall experience. This expansion would diversify the application's user base and broaden its utility in urban mobility and delivery services.

These future enhancements aim to elevate the project's capabilities and provide users with more valuable and personalized experiences. Prioritization of these enhancements will depend on factors such as user feedback, technological advancements, and resource availability.

2.13 Development Schedule and Milestones

The Table shown below depicts the schedule and milestones for the "Traffic Congestion Reduction and Management System" project:

Major Task	Sub-Task	Estimated Duration
Real-Time Data Collection and Analysis		**100 hours**
Data Sources Research and Selection		**25 hours**
	Research available data sources	**15 hours**
	Select appropriate data sources	**10 hours**
Data Collection Implementation		**35 hours**
	Implement data collection scripts for Google Maps APIs, and OpenWeatherMap API	**20 hours**
	Implement data collection scripts for navigation apps	**15 hours**
Data Collection Testing and Documentation		**40 hours**
	Test data collection procedures for government websites	**15 hours**
	Test data collection procedures for navigation apps	**15 hours**
	Document data collection processes	**10 hours**

User-Friendly Web Dashboard			**70 hours**
UI Design and Development			35 hours
	Design the user interface (UI) for the web-based dashboard	20 hours	
	Develop the UI using Flask	15 hours	
Real-Time Data Visualization			20 hours
	Implement real-time data visualization	20 hours	
User Testing and Refinement			15 hours
	Conduct user testing and gather feedback	15 hours	
Incident Handling			**50 hours**
Incident Detection Algorithm Implementation			30 hours
	Implement incident detection algorithms based on pattern recognition	30 hours	
Incident Handling Module Development			20 hours
	Develop and integrate the incident handling module	20 hours	
Algorithms and Development Process			**50 hours**
Machine Learning Algorithm Implementation			25 hours
	Implement selected machine learning algorithms for predictive modeling	25 hours	
Development Process			25 hours
	Agile and Waterfall process adaptation, including iterative development cycles	15 hours	
	Testing and quality assurance checks	10 hours	
Design Patterns			**20 hours**
MVC Pattern Implementation			10 hours
	Apply the Model-View-Controller (MVC) pattern to web application development	10 hours	
Observer Pattern Implementation			10 hours
	Implement the Observer pattern for real-time data updates and incident reporting	10 hours	
Database			**30 hours**

Database Setup and Configuration		10 hours
	Set up and configure the selected database system	10 hours
Schema Development and Implementation		10 hours
	Develop database schemas	10 hours
Data Storage and Retrieval		10 hours
	Implement data storage and retrieval functionality	10 hours
Test Plan and Verification		40 hours
Requirements Testing		10 hours
	Review project requirements and compare them with the implemented system	10 hours
Scenarios Testing		10 hours
	Simulate real-world scenarios to validate system responses	10 hours
Functions Testing		10 hours
	Test each system function individually to ensure they perform as intended	10 hours
Edge Cases Testing		5 hours
	Push the system to its limits to evaluate its behavior in extreme situations	5 hours
Unit Testing		5 hours
	Test individual components and modules to ensure they function correctly	5 hours
User Testing and Feedback		10 hours
	Involve users to evaluate the usability and effectiveness of the user interface	10 hours
Feasibility and Documentation		40 hours
Feasibility Assessment		20 hours
	Assess the project's feasibility with regard to capabilities, resources, and time	20 hours
Project Documentation		20 hours
	Generate project reports and documentation	20 hours
Total		370 hours

The updated schedule totals 370 hours, ensuring the project remained within a feasible time frame while accounting for the added sections and tasks related to project scope, testing, and documentation. Regular tracking and adjustments should be made to ensure the project stays on track.

2.14 Technical Challenges

Managing and synchronizing real-time data from diverse sources, including APIs, and integrating them with web interface along with historical and predictive insights is a substantial technical challenge. This necessitates the development of efficient data pipelines and robust data processing techniques to ensure that the system operates with minimal latency and maintains data accuracy.

Handling high volumes of data in real-time, especially in a dynamic urban environment, is a significant scalability challenge. The system must be designed to accommodate data growth while maintaining performance, which involves optimizing algorithms, database structures, and server configurations.

Integrating data from various sources while maintaining data consistency and accuracy is a complex task. Students will need to develop data transformation processes and reconciliation strategies to merge data from different formats and standards.

Creating accurate predictive models for traffic patterns and congestion management requires a deep understanding of machine learning algorithms, data feature engineering, and model validation techniques. It requires extensive research to identify the most suitable models and fine-tune them for real-time prediction.

Developing an effective system for real-time incident detection and response is challenging. This involves researching incident detection algorithms, incident prioritization strategies, and integration with emergency services. Optimizing algorithms for real-time data analysis, traffic prediction, and congestion management will be a time-consuming task.

Addressing these challenges will require extensive research, problem-solving, and experimentation, ultimately stretching the students' technical capabilities and preparing them for real-world scenarios in data analysis, machine learning, web development, and traffic management which requires beyond standard classroom teachings.

3. Conclusion

The fusion of predictive analysis with real-time data integration via a Flask web application marks a significant stride towards enhancing urban mobility management and decision-making processes. This comprehensive project amalgamated data science methodologies with web development techniques, yielding valuable insights and functionalities.

Predictive Analysis:

The predictive analysis segment of the project delved into the intricate realm of traffic forecasting, aiming to estimate Total Time Stopped at intersections within the City of Philadelphia. Leveraging machine learning models like Linear Regression and Random Forest Regressor, the project accurately predicted traffic congestion based on factors such as time, weather conditions, and historical traffic patterns.

The inclusion of feature selection, model evaluation, and visualization techniques facilitated a deeper understanding of the underlying relationships between various parameters and traffic congestion levels. Notably, the Linear Regression model showcased a Mean Absolute Error (MAE) of 6.61 and a Root Mean Squared Error (RMSE) of 10.87, indicating its moderate predictive capability. Conversely, the Random Forest Regressor model exhibited superior performance with an MAE of 0.0012 and an RMSE of 0.0134, underscoring its efficacy in capturing complex traffic patterns.

Visualizations such as correlation matrices, feature distributions, and temporal traffic patterns elucidated crucial insights, enabling stakeholders to discern underlying trends and make informed decisions. Additionally, the generation of interactive plots using Plotly and Folium further enriched the analysis, providing intuitive visual representations of traffic data and spatial clusters.

Historical Insights:

Preprocessing and cleaning of historical traffic data laid the foundation for deriving actionable insights. Through meticulous data wrangling techniques, outliers, missing values, and inconsistencies were addressed, ensuring the integrity and reliability of the dataset. Exploratory data analysis (EDA) techniques such as correlation analysis, feature distribution visualization, and temporal traffic pattern analysis unveiled valuable historical insights.

Correlation matrices revealed the interplay between various traffic metrics, weather variables, and temporal factors, shedding light on significant correlations and dependencies. Feature distribution plots provided insights into the distributional characteristics of key variables, identifying potential outliers and anomalies. Temporal traffic pattern analysis, including hourly and monthly traffic patterns, unearthed recurring trends and seasonality effects, guiding strategic planning and resource allocation.

Flask Web Application:

The Flask web application served as the bridge between predictive insights and real-time data integration, empowering users with dynamic traffic information and actionable insights. Through seamless API integrations with services like Google Maps and OpenWeatherMap, the application facilitated the retrieval of real-time weather updates, traffic conditions, alternative routes, traffic incidents, and public transit data.

The user interface, designed with HTML templates and enhanced with JavaScript for interactivity, offered an intuitive platform for users to input origin-destination pairs and desired arrival times. The backend logic, orchestrated through Python functions and Flask routes, orchestrated data retrieval, processing, and presentation, ensuring a smooth user experience.

Conclusion and Future Directions:

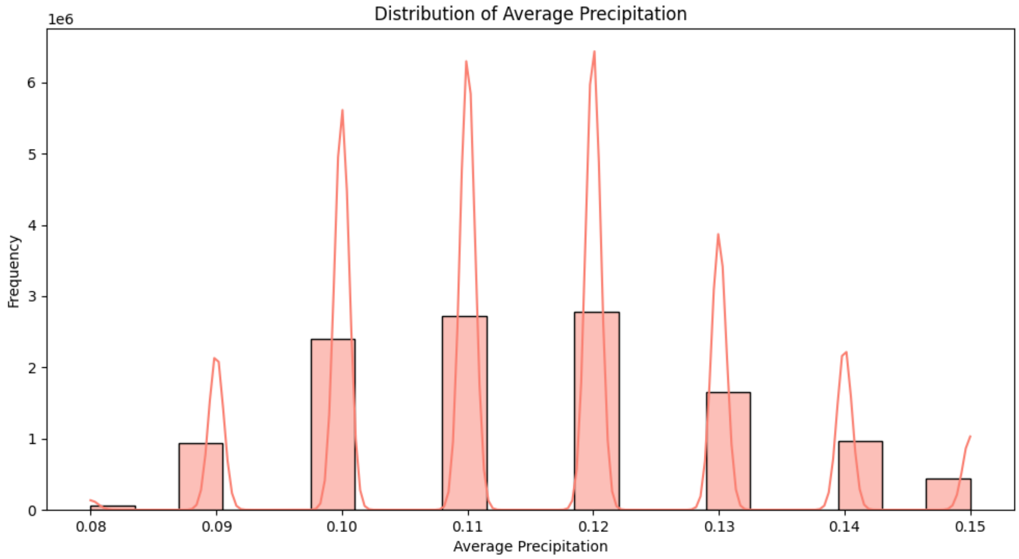
In conclusion, this project has not only augmented expertise in data science and web development but also showcased the potential for synergizing predictive analytics with real-time data integration for urban mobility management. The insights gleaned from predictive models, real-time data feeds, and historical traffic analysis can inform strategic decisions pertaining to traffic management, infrastructure development, and public transit planning.

Moving forward, several avenues for improvement and expansion present themselves. Firstly, refining predictive models by incorporating additional features such as road conditions, special events, and social factors could enhance accuracy and robustness. Furthermore, augmenting the web application with personalized user profiles, route optimization algorithms, and proactive traffic alerts could enrich the user experience and utility.

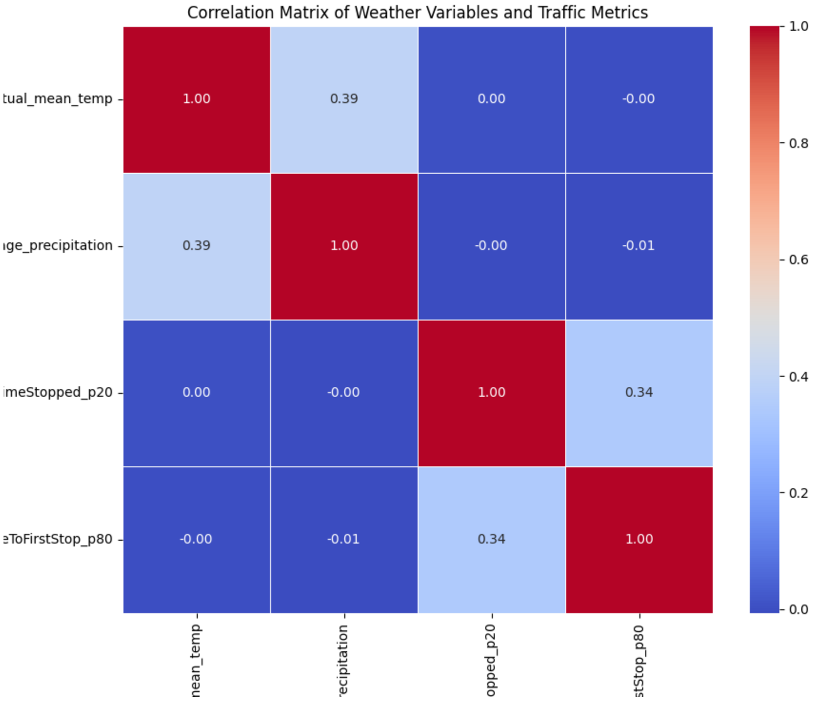
In essence, this project serves as a testament to the transformative power of data-driven decision-making in optimizing urban mobility and fostering smarter, more sustainable cities. By harnessing the synergy between predictive analytics, real-time data integration, and historical insights, this project pave the way towards a future where traffic congestion becomes a relic of the past, and efficient, equitable transportation systems thrive.

4. Appendices

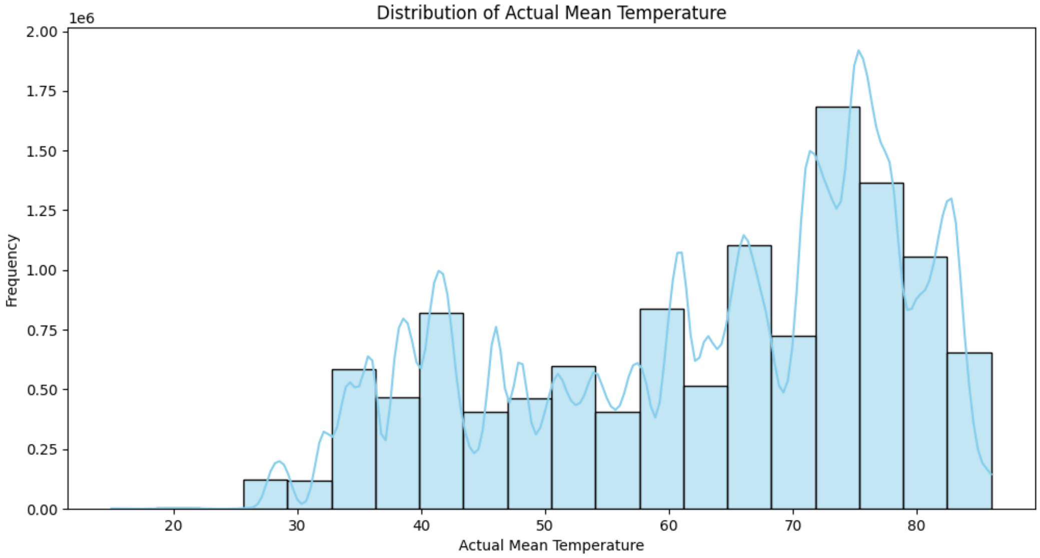
Graphs and Plots Missing from the main Body:



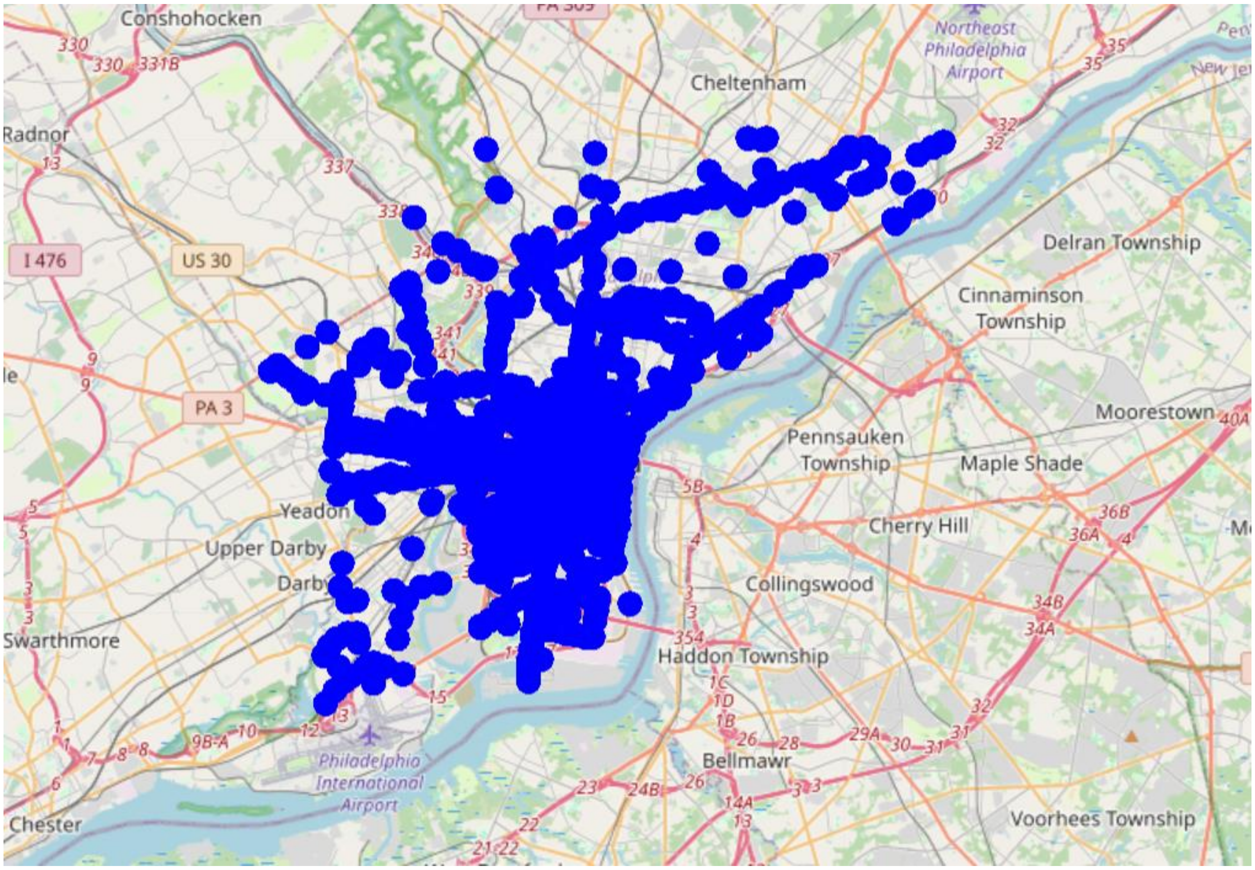
The graph shows Distribution of Average Precipitation from Weather dataset.



The graph above shows the traffic and weather analysis.



The graph above shows the Distribution of Actual Mean Temperatures



The graph above shows very congested clusters of traffic on the Map of Philadelphia, PA

App Structure and Setup Guide

Files Overview:

- app.py: The main Python file containing the Flask application setup, routes, and logic for rendering templates and handling user requests.
- templates/: A directory containing HTML templates for different pages of the web application.
- static/: A directory containing static files such as CSS stylesheets, JavaScript files, images, or other resources used by the HTML templates.
- data/: A directory containing any data files, such as CSV or JSON files, used by the application for analysis or visualization.
- requirements.txt: A text file listing all the Python packages and their versions required to run the application. This file is used by package managers like pip to install dependencies.

File Structure:

app.py:

- Imports necessary modules and libraries.
- Defines routes for different URL endpoints and their corresponding functions.
- Renders HTML templates and passes data to them using Flask's template engine.
- Handles form submissions and user interactions.

Predictive-analysis.py:

- Imports necessary libraries
- Imports necessary datasets, merges them.
- Perform data analysis, deep learning, and train models to predict traffic using two models Linear Regression, and Random Forest model.
- Provides results of RME, and RMSE, the accuracy of the models in Prediction, and graphs, and charts for better visualization.

tests/:

- Contains test files created to generate test cases and check functionality.
- Contains test_app.py which contain unit tests and contain test_integration.py which contain integration tests.

templates/:

- Contains HTML files for different pages of the web application, such as home page, results page, about page, etc.
- Contains home.html (home page of the App), index.html (Real-Time Insights), insights.html (for POST method of real-time), historical_insights.html (Historical Insights)
- Uses Jinja2 templating syntax to dynamically render data received from the server.

static/:

- Stores static files like CSS (styles.css), JavaScript (scripts.js), images (All the graph images created dynamically), and other resources (clustered_traffic_map.html)
- CSS files control the styling of HTML elements.
- JavaScript files handle client-side interactions and dynamic behavior.

data/:

- Houses any data files required for analysis or visualization.
- May include CSV, JSON, or other structured data files.

Basic Requirements:

- Python 3.x installed on your system.
- The necessary Python packages installed.
- A web browser to view and interact with the application.

Running the App:

- Clone or download the project repository from the source, or directly run the app.py in Pycharm
- Navigate to the project directory in your terminal or command prompt.
- Install the required Python packages by running:
  pip install -r requirements.txt
- Once the dependencies are installed, run the Flask application by executing:
  python app.py
- Open a web browser and navigate to http://localhost:5000 to access the web application.

Navigating the App:

- Upon accessing the application in your web browser, you will be presented with the home page.
- Use the navigation menu or links provided to explore different sections or functionalities of the app.
- Follow on-screen instructions or prompts to input data, make selections, or interact with the app's features.
- View results, insights, or visualizations presented on the respective pages.

5. References

- B. S. Meghana, S. Kumari and T. P. Pushphavathi, "Comprehensive traffic management system: Real-time traffic data analysis using RFID," 2017 International conference of Electronics, Communication and Aerospace Technology (ICECA), Coimbatore, India, 2017, pp. 168-171, doi: 10.1109/ICECA.2017.8212787.
- Google API Keys: https://console.cloud.google.com/apis/dashboard?project=quiet-axon-420321
- Sharif, J. Li, M. Khalil, R. Kumar, M. I. Sharif and A. Sharif, "Internet of things — smart traffic management system for smart cities using big data analytics," 2017 14th International Computer Conference on Wavelet Active Media Technology and Information Processing (ICCWAMTIP), Chengdu, China, 2017, pp. 281-284, doi: 10.1109/ICCWAMTIP.2017.8301496.
- OpenWeather App API key: https://home.openweathermap.org/api_keys

- Z. Zamani, M. Pourmand and M. H. Saraee, "Application of data mining in traffic management: Case of city of Isfahan," 2010 2nd International Conference on Electronic Computer Technology, Kuala Lumpur, Malaysia, 2010, pp. 102-106, doi: 10.1109/ICECTECH.2010.5479977.
- P. Rizwan, K. Suresh and M. R. Babu, "Real-time smart traffic management system for smart cities by using Internet of Things and big data," 2016 International Conference on Emerging Technological Trends (ICETT), Kollam, India, 2016, pp. 1-7, doi: 10.1109/ICETT.2016.7873660.
- Dataset Source for Historical Insights: https://www.kaggle.com/code/fayzaalmukharreq/traffic-congestion/notebook
- Shapefile for Philadelphia - TIGER/Line Shapefile, 2019, county, Philadelphia County, PA, All Roads County-based Shapefile - Catalog (data.gov)
- Google Book for Data Collection using Python: https://books.google.ca/books?id=DN4SEAAAQBAJ&lpg=PP1&ots=P3DcfC197e&dq=data%20collection%20using%20python&lr&pg=PA18#v=onepage&q=data%20collection%20using%20python&f=false
- Miller, B., & Mick, S. (2019). Real-Time Data Processing using Python in DigitalMicrograph. *Microscopy and Microanalysis, 25*(S2), 234-235. doi:10.1017/S1431927619001909
- Verma, C. Kapoor, A. Sharma and B. Mishra, "Web Application Implementation with Machine Learning," 2021 2nd International Conference on Intelligent Engineering and Management (ICIEM), London, United Kingdom, 2021, pp. 423-428, doi: 10.1109/ICIEM51511.2021.9445368.
- Q. Lin, H. Kuang and Z. Xilin, "Design and Implementation of Big Data Forecasting System Based on Intelligent Transportation," in IEEE Transactions on Consumer Electronics, doi: 10.1109/TCE.2023.3319639.
- Weather Dataset source: import-json-weather-data (kaggle.com)
- Dataset sample: https://opendataphilly.org/categories/transportation/
- Ma X, Yu H, Wang Y, Wang Y (2015) Large-Scale Transportation Network Congestion Evolution Prediction Using Deep Learning Theory. PLoS ONE 10(3): e0119044. https://doi.org/10.1371/journal.pone.0119044
- **"Random Forest Algorithm," Simplilearn, 2023. [Online]. Available: https://www.simplilearn.com/tutorials/machine-learning-tutorial/random-forest-algorithm. [Accessed: Dec. 02, 2023].**
- **"Random forest Algorithm in Machine learning: An Overview," MyGreatLearning, 2023. [Online]. Available: https://www.mygreatlearning.com/blog/random-forest-algorithm/. [Accessed: Dec. 02, 2023].**
- **"Understanding Random Forest Algorithms with Examples," Analytics Vidhya, Jun. 2021. [Online]. Available: https://www.analyticsvidhya.com/blog/2021/06/understanding-random-forest/**