Traffic Congestion Reduction and Management System

COMP 8047 – Major Project

Kartik Verma – A01022059
5-19-2024

Table of Contents

List of Figures

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've taken some courses in database administration and analysis, which I found really interesting. These classes have taught me a lot, and I think they've given me the basic skills I need for this project called 'Traffic Congestion Reduction and Management System.' I'm excited to work on this because traffic is such a big problem, and I hope to learn even more from this project."

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 aims to tackle the ongoing problem of traffic congestion in urban areas with the knowledge and skills of Data Analysis. The project's main objective is to use techniques such as data collection, analysis, and management techniques to improve and optimize traffic flow and reduce congestion in Philadelphia, Pennsylvania. By collecting historical and real-time data from different sources, including public and government websites and crowdsourced data from navigation apps such as Google Maps and Apple Maps, and applying modern 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 practical and innovative solution to address and tackle urban traffic congestion. This project has the potential to benefit various stakeholders. Urban Planning Departments can use the system's data-driven insights to make better decisions about City Planning, and Urban infrastructure development. Local Municipal Authorities can take better decisions and enforce laws insuring the safer and less congested lower maintenance roads. Locals or the Normal Public can make better decisions in planning their trips avoiding congestion and long waiting times in longer commutes. Businesses will benefit from smoother logistics and more consumer output due to lesser congestion of traffic in the city. In summary, this project plans to solve multiple problems for all kinds of people in the city thus boosting the economy and overall development of the city.

1.4 Problem Statement and Background

Traffic congestion is one of those problems that affect both society and economy on a daily basis. In 2019, traffic jams in the United States costed nearly $88 billion due to wasted time and fuel expenses, according to data from the American Transportation Research Institute. The congestion and traffic not just hit financially, but also contributes to excessive carbon emissions and air pollution, leading to negative impacts on public health and the environment. Solving this issue will not only bring easier and smoother travel, but also make our cities more livable and sustainable.

The traffic problem has been around for decades, and it's getting worse by rapid urbanization as more people are moving to cities and more people are driving thus causing traffic. In 2018, the United Nations estimated that by 2050, approximately 68% of the world's population will live in urban areas. In response to this urbanization wave, traditional traffic management systems have adapted by using static traffic signal timings and traffic cameras. However, these systems often lack the real-time adaptability required to effectively address the ever-changing traffic problems and predict the patterns to improve in future.

Traffic congestion is a complicated problem influenced by various factors, like road design, population density, time of day, accidents, and weather conditions. Current and most traffic management strategies mostly depend on historical data and predefined traffic signal timings. While these strategies can be helpful, they fall short when it comes to addressing the ever-changing and evolving nature of congestion. A comprehensive solution is required to utilize real-time data, predictive analytics, and user engagement to effectively manage congestion in urban environments.

Everyday used 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 often lack solving the traffic problem across the whole city and don’t provide long-term solutions. Other technologies include Traffic Management Systems (TMS) which monitor the traffic using traffic light signals. 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 understand the project and the report properly and effectively, it is important to understand the background of the Project. In this section, I will try to provide an objective overview of the factors that led to the idea of the project and the main problems it is trying to solve.

Nature of the Project: The project will focus to develop an interactive app for the City Planners of Philadelphia, PA. The major components of the project will include providing real-time traffic and weather insights and various historical trends to judge congestion and take better decisions in the future. The project also tends to provide predictive analysis using historical data and machine learning algorithms which will provide better mobility and improve transportation efficiency.

Challenges in Urban Mobility: Cities around the world are facing multiple increasing challenges in managing traffic congestion, optimizing transportation networks, and ensuring the safety and convenience of travellers. These issues are made worse by factors like population growth, rapid urbanization, and unpredictable weather patterns. While some apps provide real-time features, and other datasets provide historical information, this app will provide a single place to analyze both features in more detail and interactive way.

Project Statement within the Context of Project Proposal: The idea for this project came from a detailed analysis of current traffic management systems and finding the gaps in dealing with weather-related disruptions. The project proposal set out the main goal of developing a new solution that combines real-time weather data with traffic management algorithms to improve traffic flow and make the system more resilient to bad weather.

2.2 Essential Problems Being Solved

In this section, I will discuss about the main problems and challenges this project will solve. Understanding of these problems will help discuss the solutions this project will focus on and how they are relevant.

Traffic Congestion: One of the biggest issues urban cities and transportation is facing today is the prevalence of traffic congestion, 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: Adverse weather conditions such as rain, snow, fog, and extreme temperatures, greatly affect traffic flow patterns and transportation infrastructure. These harsh weather conditions result in reduced visibility, slippery road surfaces, and increased accident rates, further causing more traffic congestion slowing down movement of vehicles.

By focussing on both weather and traffic management, and their relationship on how different weather conditions affect the traffic situations, this project aims to improve the resilience and efficiency of urban transportation networks. By proactively predicting weather-related disruptions and implementing

adaptive strategies, the proposed solution will 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 discuss about different alternative approaches and solutions that could potentially address the identified challenges. By considering multiple other perspectives, I will discuss about the pros and cons of all these approaches.

Alternative Approaches to Traffic Management: There are many existing methods in the market to deal with traffic congestion. These include traditional methods include optimizing traffic signals, managing lanes, and improving public transportation. While these methods can work, they often lack the flexibility to deal with changing weather conditions.

Weather-Informed Traffic Management Strategies: Another approach involves using weather data in traffic Management systems However, these methods usually rely on the historical data to make predictions but 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 Machine Learning (ML), artificial intelligence (AI), and data analytics, offer new ways to improve traffic management. These innovative technologies use sensor networks, predictive modeling, and cloud computing to deliver real-time traffic insights and adaptive control strategies. However, these technologies are however new and not cost-effective for many cities.

Comparison of Alternative Solutions: Each alternative solution has its own set of advantages, challenges, and trade-offs. A comparative analysis is required to evaluate the effectiveness, scalability, and cost-effectiveness of different approaches. Factors such as data accuracy, ease of use, cheaper in cost, and covers both historical and real-time solutions are important to check while selecting for ideal solutions.

Rationale for the Selected Approach: After carefully considering different alternative solutions, I opted for an integrated approach that combines real-time weather data and insights with advanced machine learning 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 taken in order to provide proactive, and adaptive solution to address the dynamic nature of urban transportation and ever changing weather conditions.

2.4 Solution Chosen for this Project and Rationale Behind

In this section, I will discuss about the selected solution and approach and why it was chosen over other approaches to cover all the project's objectives.

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

The project aims to develop a advanced traffic management system that improves traffic flow in urban areas, especially during adverse weather conditions. This solution integrates advanced data analytics, machine learning algorithms, and data from APIs. 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 to give real-time weather and traffic insights for the city. This project also analyze historical patterns, such as traffic during the certain hours of the day or weather conditions affecting vehicles to create congestion, or the most congested days of the week, etc. Along with Real-time and historical insights, it uses Machine learning models, which include Linear Regression and Random Forest to 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. The system finally integrates to develop a user-friendly Flask-based web dashboard where users input their location, arrival time, and desired destination to get real-time traffic predictions and suggested alternate routes, along with the Public Transit options. 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 using real-time weather data, this solution allows for proactive decision-making and adaptive strategies to deal with weather-related traffic issues such as users can depart early to reach the same destination when it's raining heavily or snowing. This approach helps anticipate and respond to changing conditions, reducing congestion and accidents.

2. Enhanced Accuracy and Precision: Unlike traditional traffic methods that only use historical data or fixed schedules, this system provides accurate and precise information about current weather conditions and their effects on roads, visibility, and driver behavior. This detailed information helps make better traffic management decisions.

3. Comprehensive Solution Coverage: This solution covers several important aspects, such as:

    - Using the Flask web framework to develop a user-friendly dashboard for real-time monitoring and control.
    - Integrating real-time data insights and APIs to capture and process live traffic and weather data.
    - Processing historical data and using predictive analysis to forecast traffic patterns and optimize resources.
    - Using Tableau for data visualization, which helps create interactive and easy-to-understand visuals for better decision-making.

4. Scalability and Flexibility: This solution is designed to be scalable and flexible, so it can be customized and used in different urban environments with varying traffic patterns, infrastructure, and weather conditions. The Modular architecture of this system will help to make future upgrades and integrate with the existing system.

2.5 Design and Development

The Design and Development section of the Project report will cover the entire process of designing the flask app and developing various predictive analysis algorithms. These are some of the steps:

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)
```

Figure 1: Loading and Reading data from csv file using the pandas library.

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

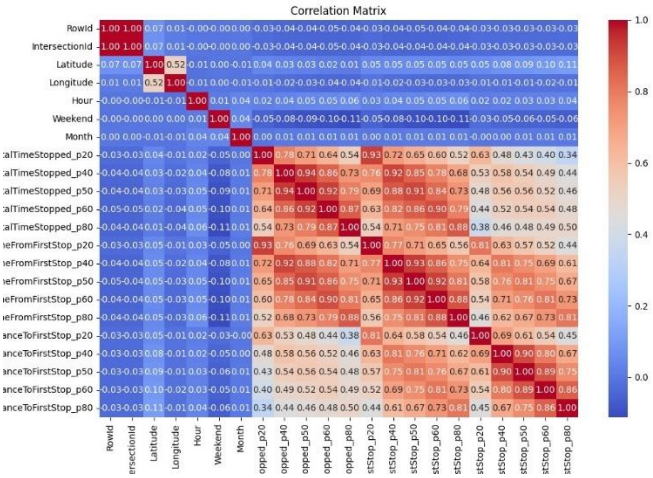


Figure 2: Correlation analysis Heatmap: Relationship between Variables of original dataset

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

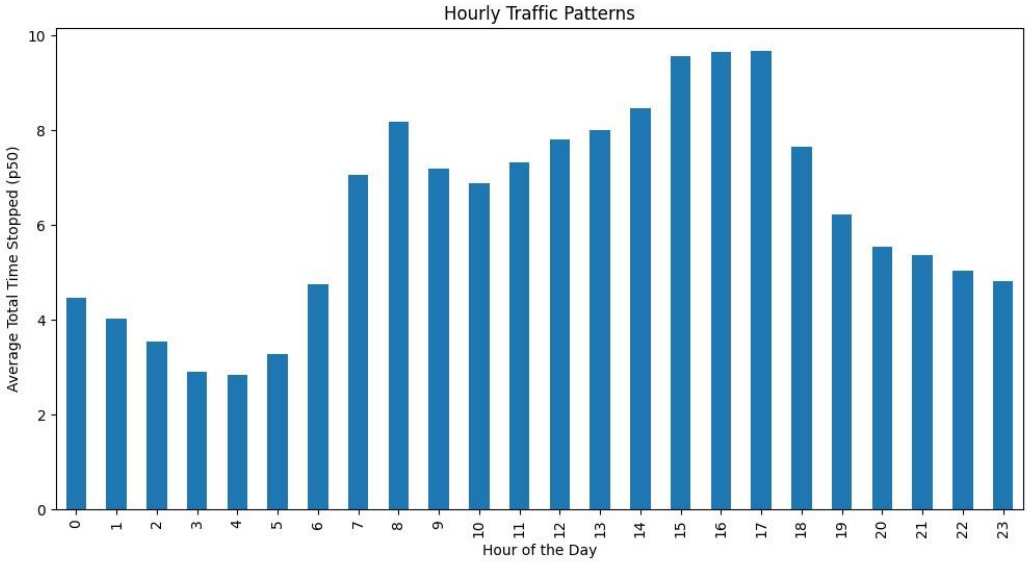


Figure 3: Hourly Traffic Patterns: Average Speed Throughout the Day.

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

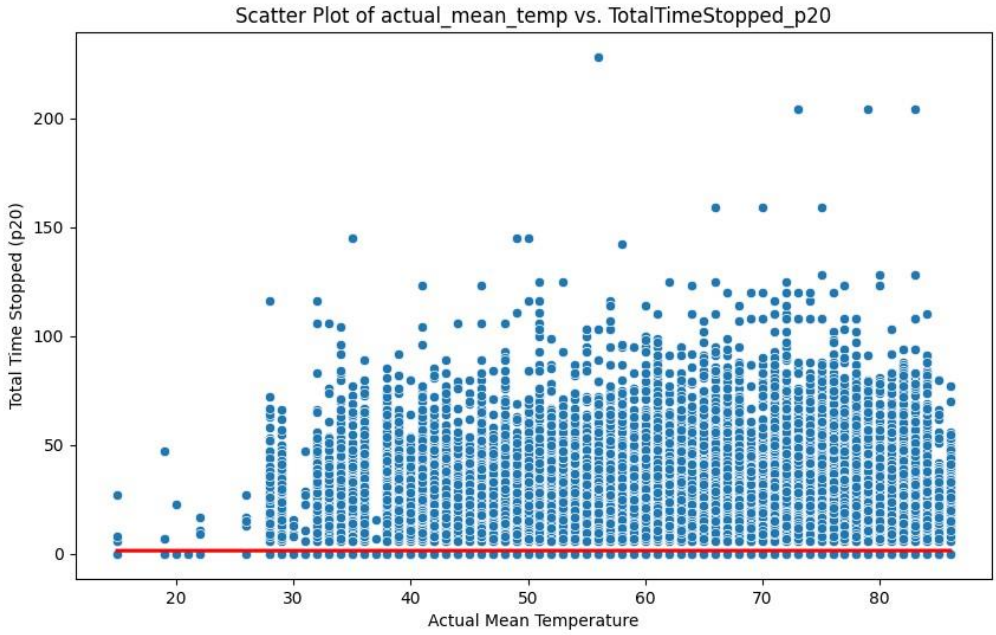


Figure 4: Temperature vs. Traffic Stop Duration – Scatter Plot

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)
```

Figure 5: Cleaning by checking missing values and duplicates in the merged traffic-weather data

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:

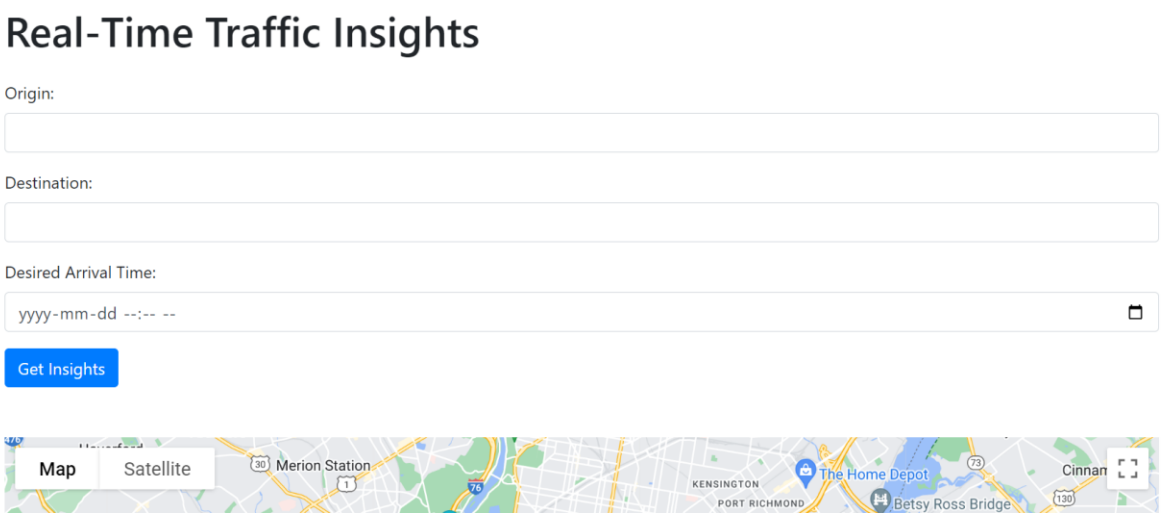


Figure 6: Real-time traffic insights catered on the basis of Origin, destination, and arrival time input by the user.

- 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.

```python
# 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
```

Figure 7: functions asking user to input 3 values for desired real-time traffic results

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.

```python
@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)
```

Figure 8: Python code showing flask app's structure of 3 pages and how they are interlinked

The code snippet above shows three 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, this project aims to empower users with the information needed to make informed decisions and optimize their commuting experience.

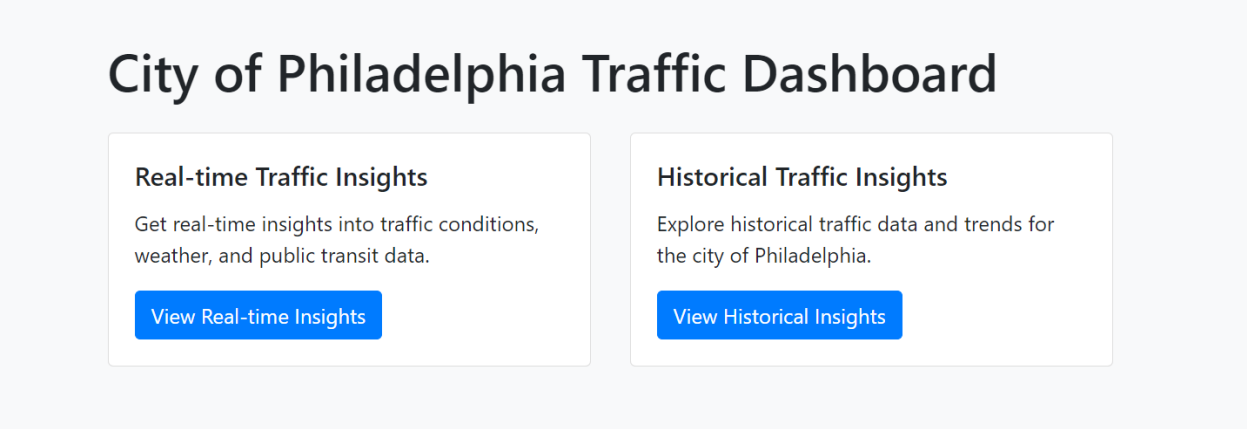


Figure 9: Home page of the Web application

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. Enabling different services shown below from Google Maps API will provide additional information about travel times, congestion levels, and alternative routes between specified locations.

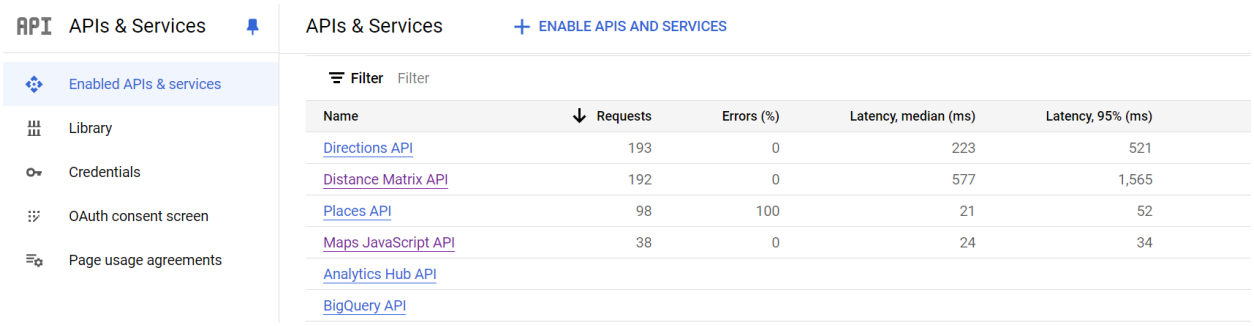


Figure 10: google APIs & Services activated to collect real-time data and insights

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.

```python
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
```

Figure 11: Conversion of JSON output from APIs to human readable format for Web app

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 the app to obtain real-time weather information for a specified location.

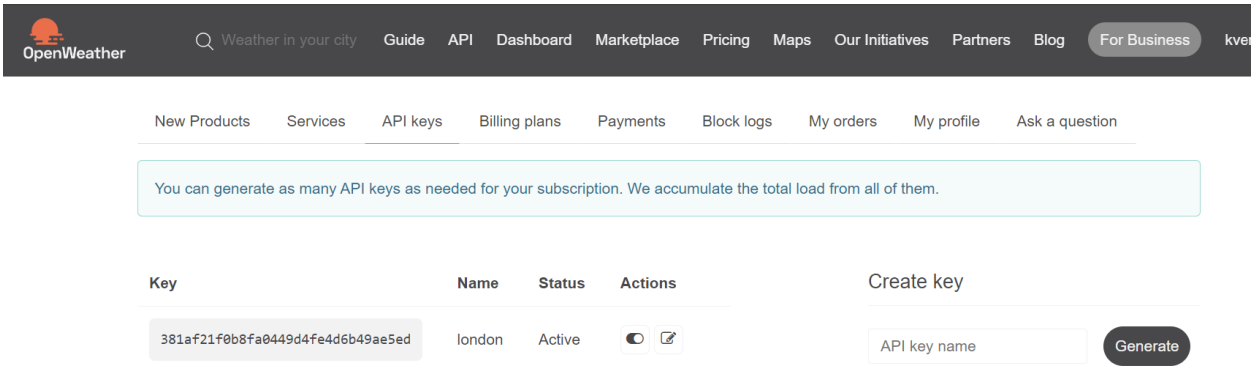


Figure 12: API keys for the OpenWeatherMap API

The screenshot above shows the API from OpenWeatherMap is active.

```python
# 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
```

Figure 13: Different weather insights pulled from the API for the app

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: All these Charts and graphs are obtained from Manipulating and visualizing the historical data

1. Traffic Congestion Trends: A bar chart depicting the top 10 congested routes in the city. This chart helps users plan their travel routes more effectively and not choose the most congested routes.

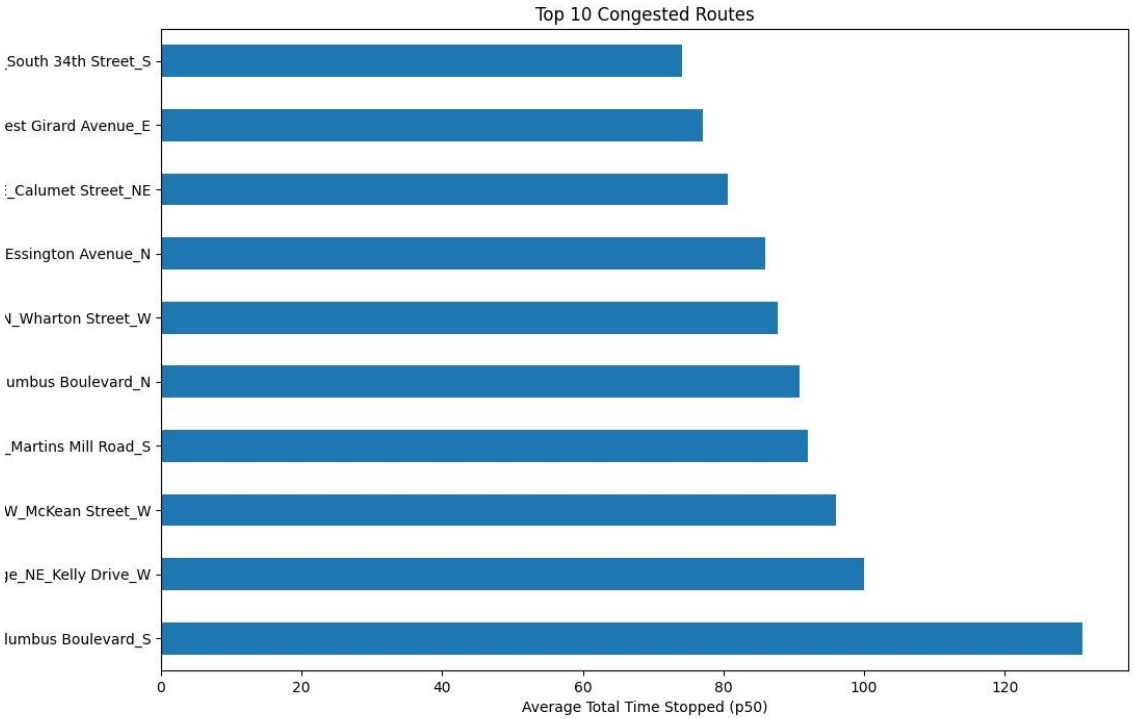


Figure 14: Top 10 Congested routes in the city

The graph in the above screenshot shows the top ten congested Routes with their names and how much average time is stopped.
This Route analysis is achieved by grouping the “Path” Column, and calculating the mean total time stopped for each route and then the routes are sorted into top 10 order.

Using the following code:

```
# Route Analysis
route_traffic = train_df.groupby('Path')['TotalTimeStopped_p50'].mean().sort_values(ascending=False).head(
route_traffic.plot(kind='barh', figsize=(12, 8))
plt.xlabel('Average Total Time Stopped (p50)')
plt.ylabel('Route')
plt.title('Top 10 Congested Routes')
plt.savefig('route_analysis.png')
plt.show()
```

Figure 15: Code for Route Analysis

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.

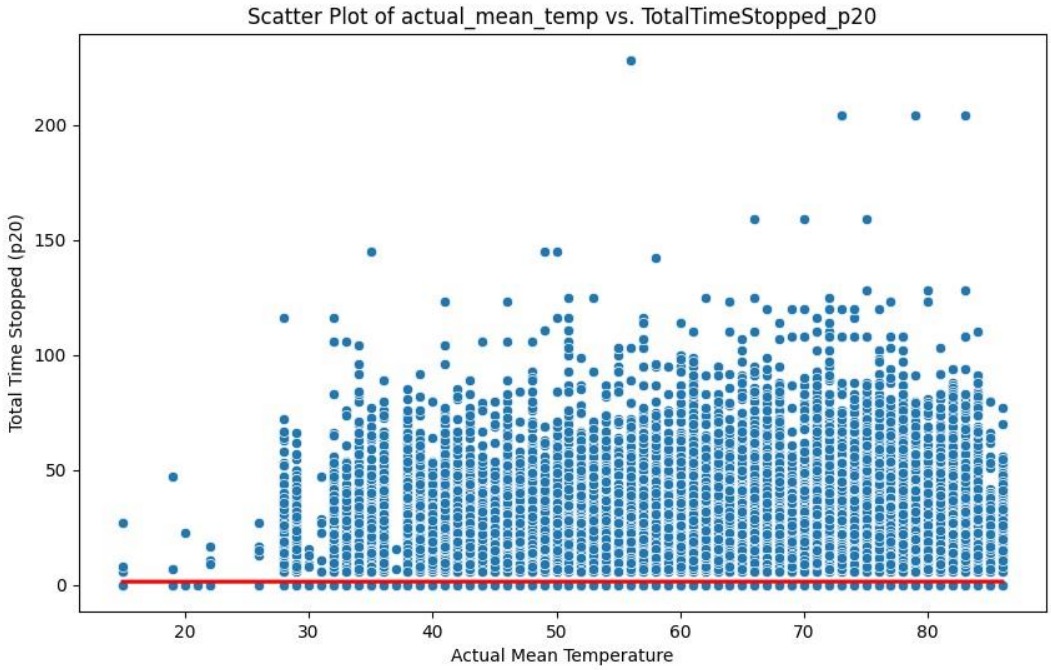


Figure 16: Scatter plot showing relationship between Temperature and traffic

The scatter plot above shows the Traffic or total time stopped affected by the Actual Mean temperature. This plot visualizes the relationship between the 'actual_mean_temp' (actual mean temperature) and 'TotalTimeStopped_p20' (total time stopped, p20 percentile). The scatter plot shows individual data points, while the red regression line represents the linear fit between the two variables. The x-axis is labeled as "Actual Mean Temperature," and the y-axis represents "Total Time Stopped (p20).

Using the following code:

```
# Scatter plot with Regression Line
plt.figure(figsize=(10, 6))
sns.scatterplot(data=merged_df_sample, x='actual_mean_temp', y='TotalTimeStopped_p20')
sns.regplot(data=merged_df_sample, x='actual_mean_temp', y='TotalTimeStopped_p20', scatter=False, color='r
plt.title('Scatter Plot of actual_mean_temp vs. TotalTimeStopped_p20')
plt.xlabel('Actual Mean Temperature')
plt.ylabel('Total Time Stopped (p20)')
plt.savefig('traffic-weather3.png')
plt.show()

# Histograms of Weather Variables
```

Figure 17: Code for Scatter Plot

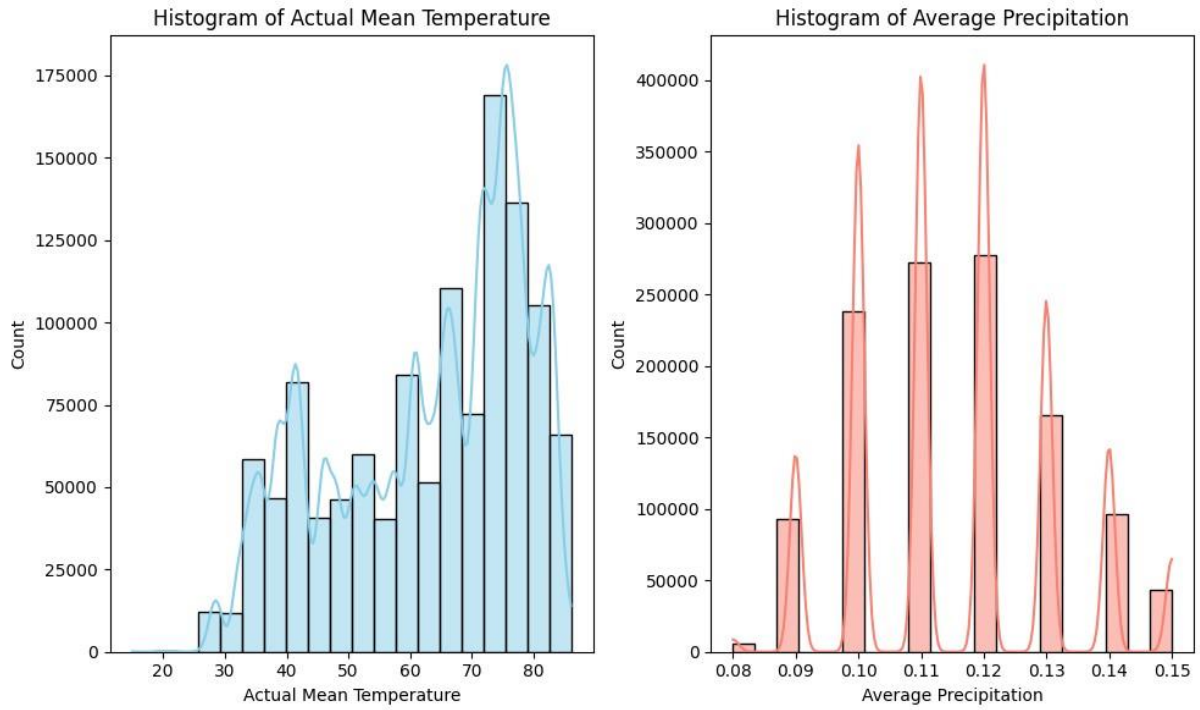


Figure 18: Climate Data Distribution: Temperature and Precipitation Histograms

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

The left subplot displays the distribution of 'actual_mean_temp' (actual mean temperature), while the right subplot shows the distribution of 'average_precipitation' (average precipitation) showing weather conditions in Philadelphia, PA

Using the following code:

```
# Histograms of Weather Variables
plt.figure(figsize=(10, 6))
plt.subplot( *args: 1, 2, 1)
sns.histplot(merged_df_sample['actual_mean_temp'], bins=20, kde=True, color='skyblue')
plt.title('Histogram of Actual Mean Temperature')
plt.xlabel('Actual Mean Temperature')

plt.subplot( *args: 1, 2, 2)
sns.histplot(merged_df_sample['average_precipitation'], bins=20, kde=True, color='salmon')
plt.title('Histogram of Average Precipitation')
plt.xlabel('Average Precipitation')
plt.tight_layout()
plt.savefig('traffic-weather4.png')
plt.show()
```

Figure 19: Code for Histograms showing Weather variables

3. Route Optimization: One feature of the project is showing alternative routes with different travel times and levels of traffic. This can help people choose the best route by looking at real-time traffic data. It also lets users see different route options on a map.

```
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
```

Figure 20: Real-time traffic insights such as Distance, Duration, Traffic Incidents, and Alternative Routes

```
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
```

Figure 21: Real-time traffic insight for Public Transit data from Origin to destination

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 using these insights and visualizations, users can make informed decisions to travel more efficiently and avoid 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.

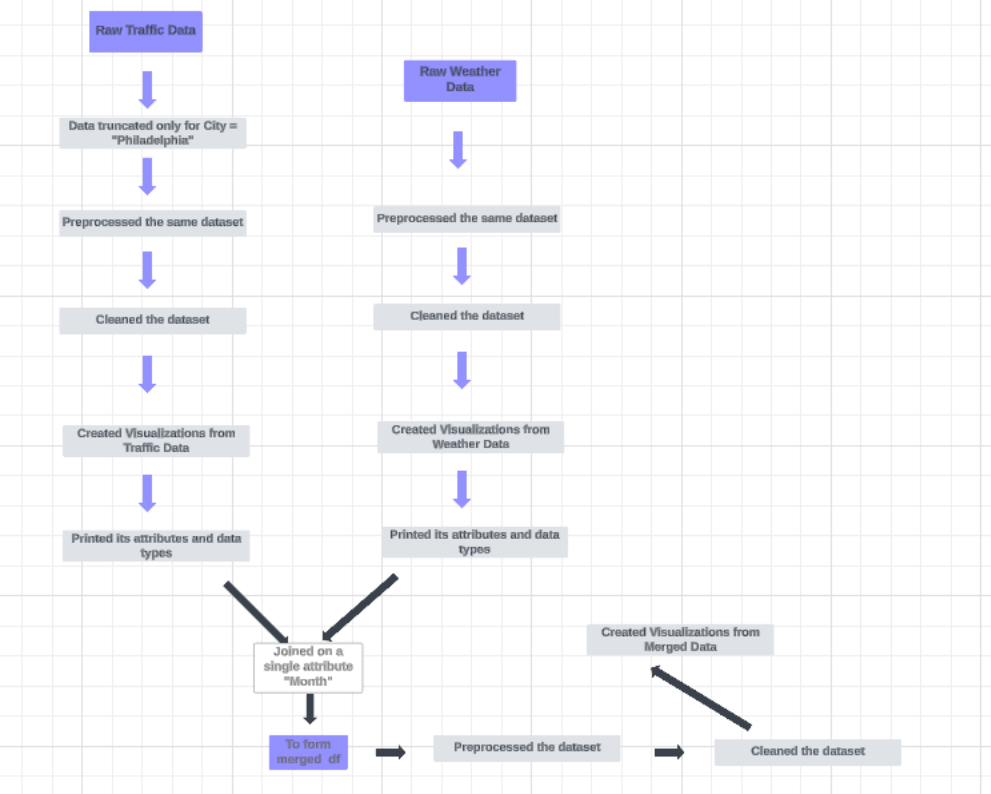


Figure 22: Flowchart Sequence Diagram

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.

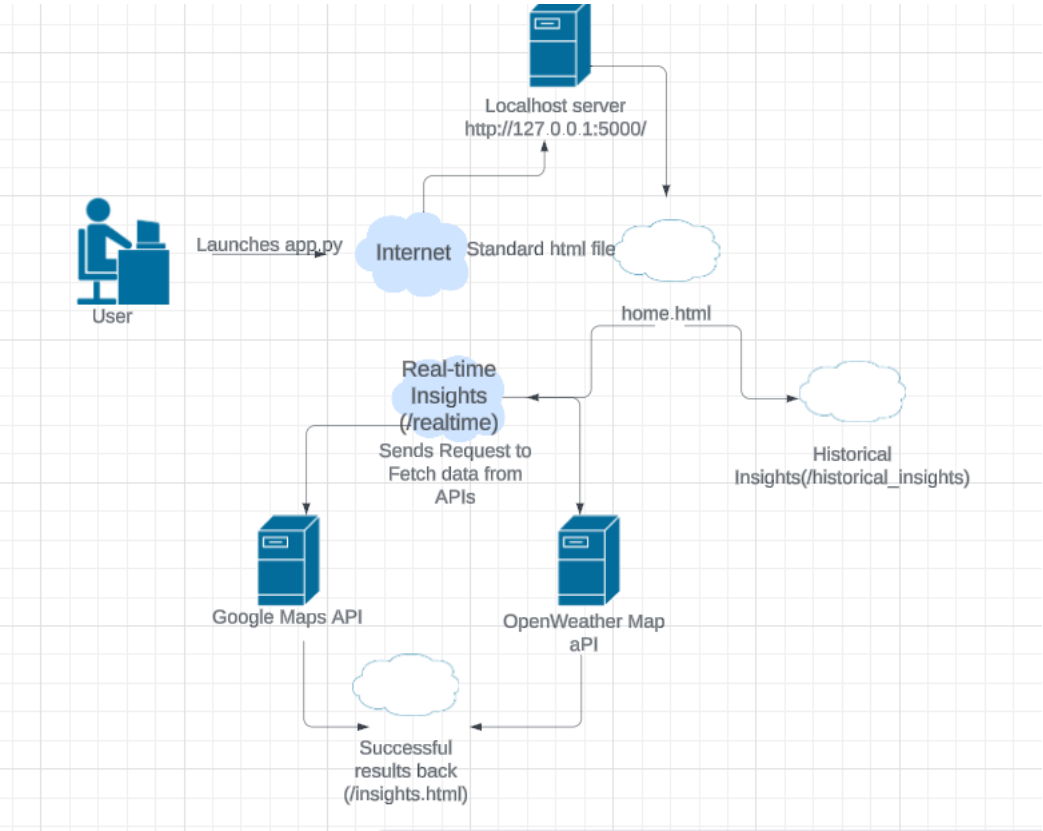


Figure 23: Network diagram of requests flowing in the system

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.

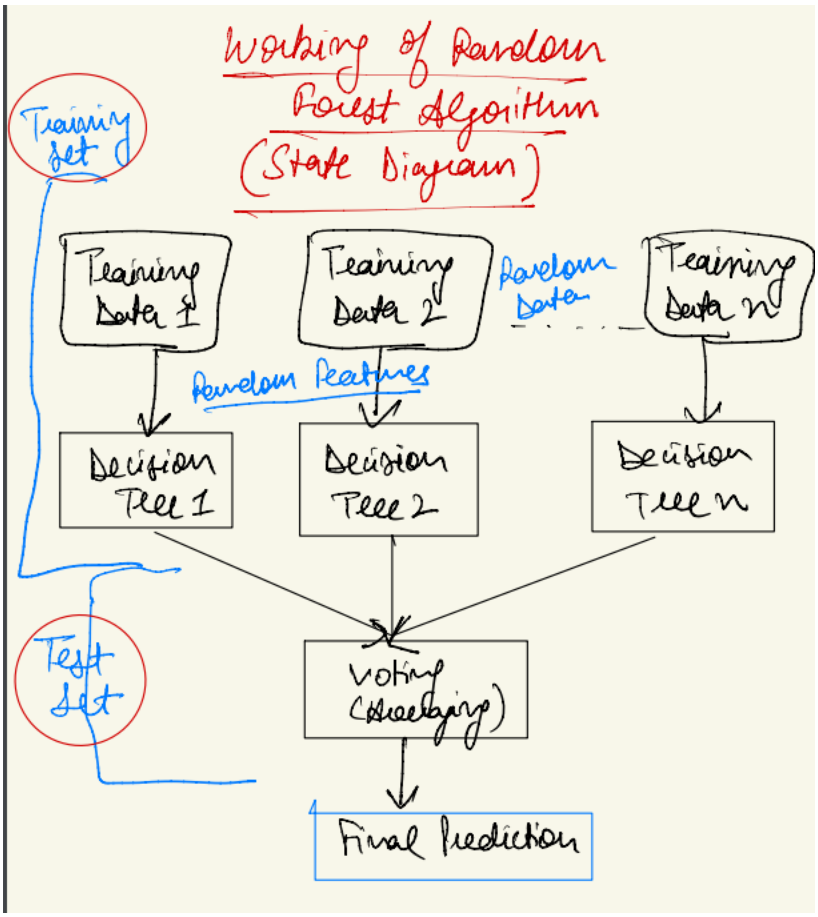


Figure 24: State Diagram for Random Forest Model for Predictive Modelling

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 cover different aspects to ensure the robustness, reliability, and usability of the web-based dashboard application. The types of tests to be conducted include are:

1. Functionality Testing:

This type of testing checks if all the different parts of the app work the way they are supposed to according to the 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.

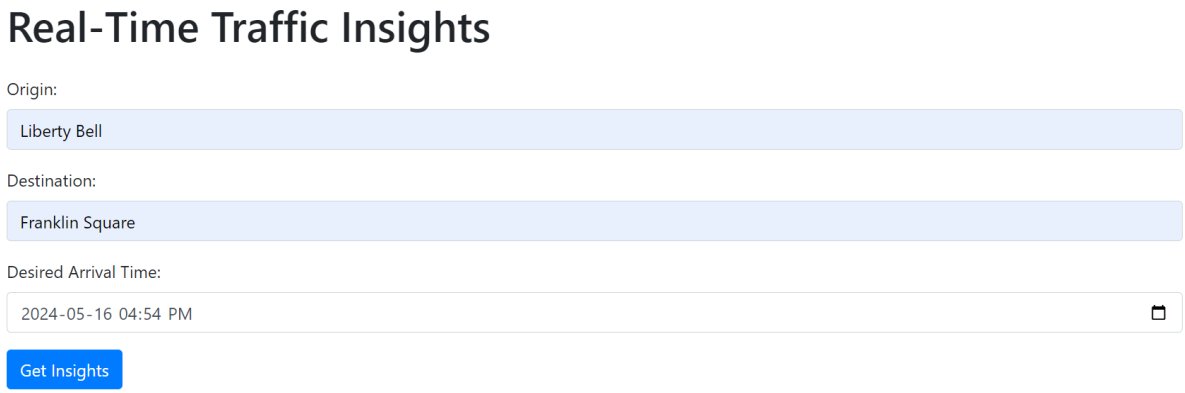


Figure 25: Test Case 1 – Entering all valid inputs

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.

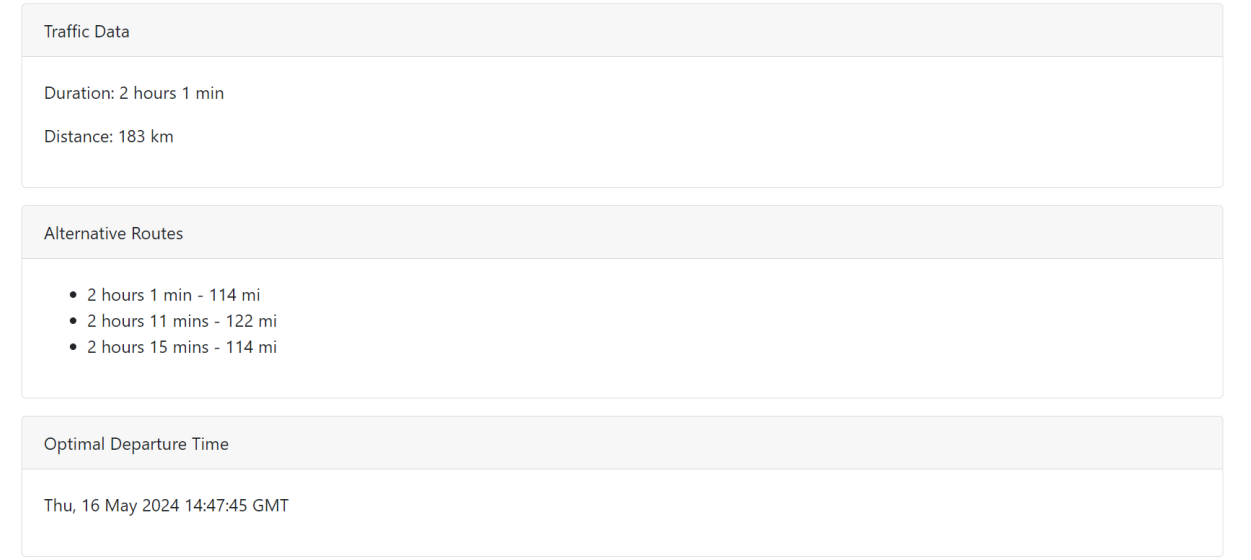


Figure 26: Test Case 2 – Traffic Data Retrieval (Success)

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.

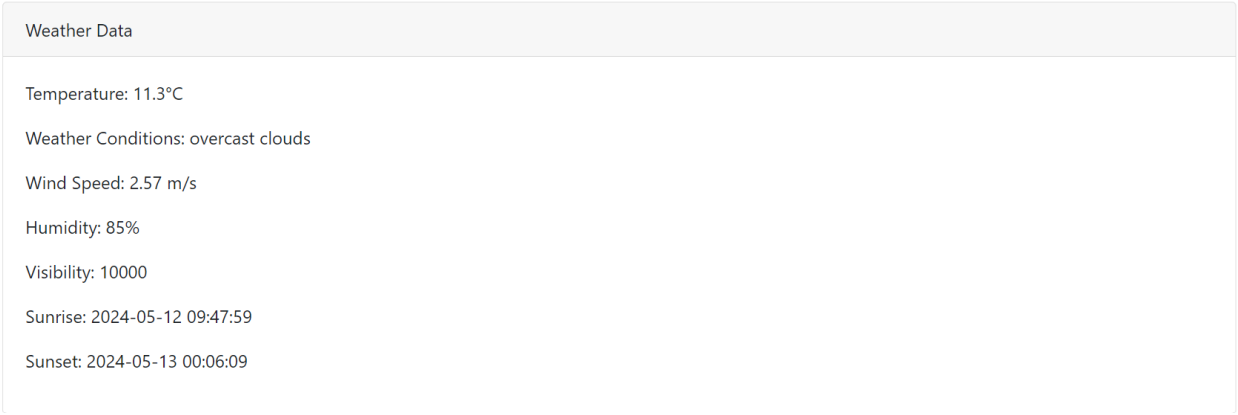


Figure 27: Test Case 3 – Weather Data retrieval (Success)

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.

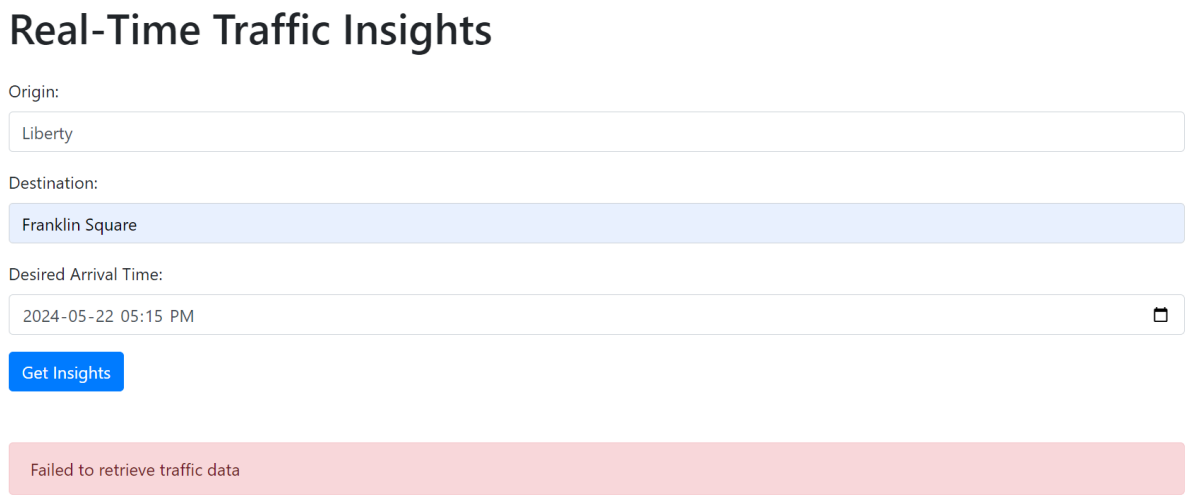


Figure 28: Test Case 4 – Adding Invalid Input (for Origin)

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.

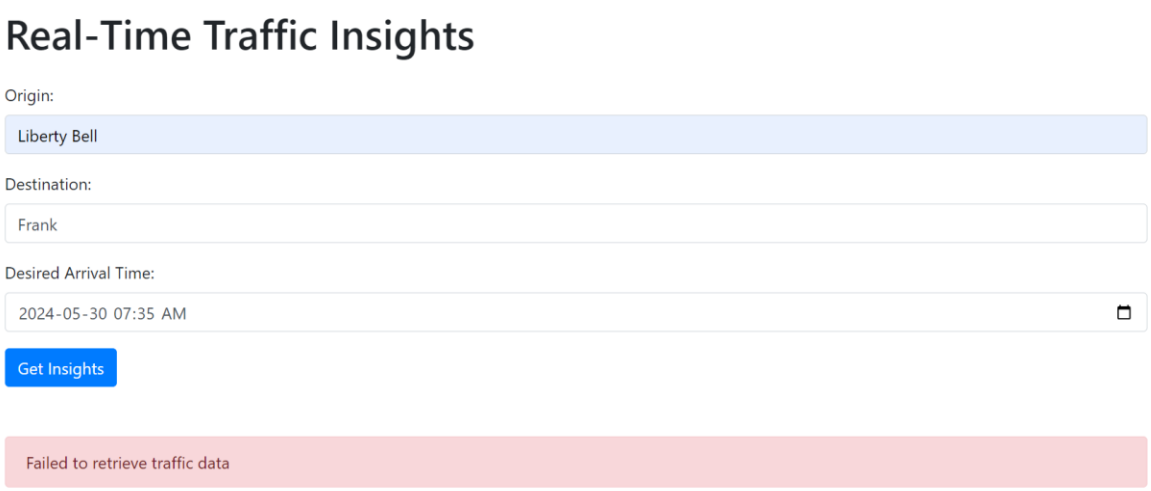


Figure 29: Test Case 5 – Adding Invalid Input (Destination)

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.

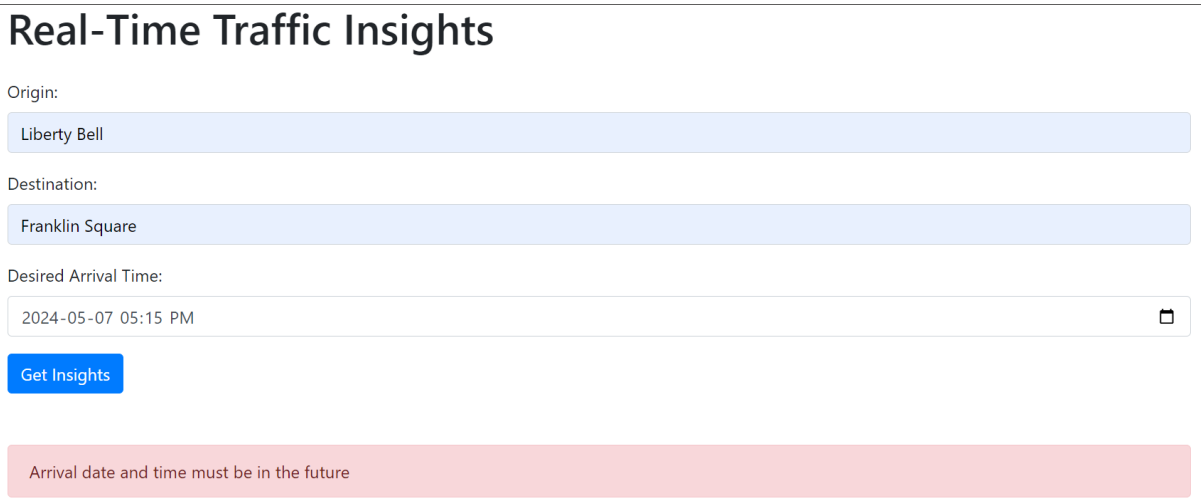


Figure 30: Test Case 6 – Invalid Input (Desired Arrival Time- past date added)

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.

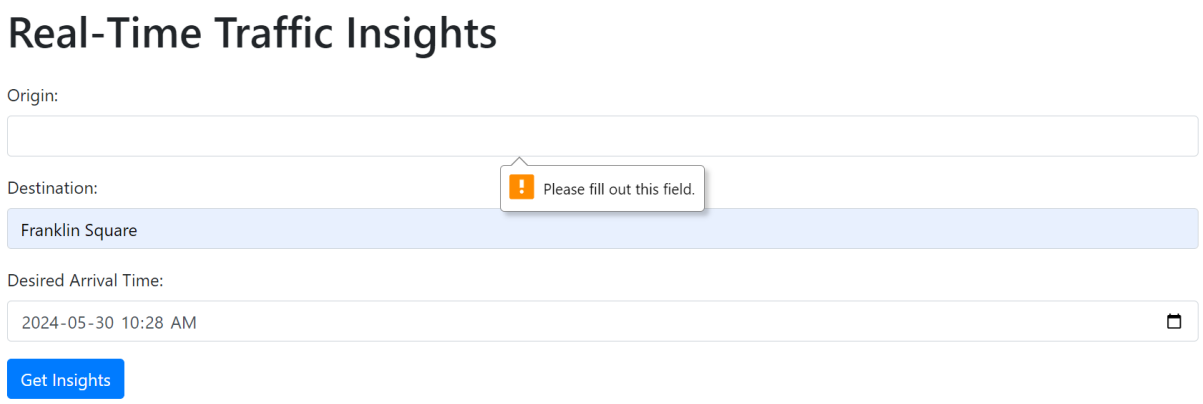


Figure 31: Test Case 7 – Missing Input (Origin)

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.

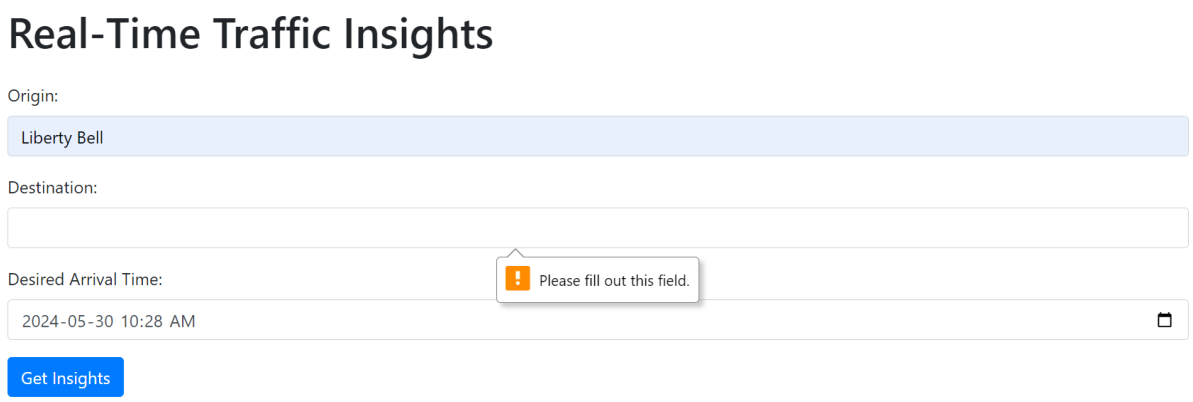


Figure 32: Test Case 8 – Missing Input (Destination)

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.

Figure 33: Test Case 9 – Missing Input (Desired Arrival Time)

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:

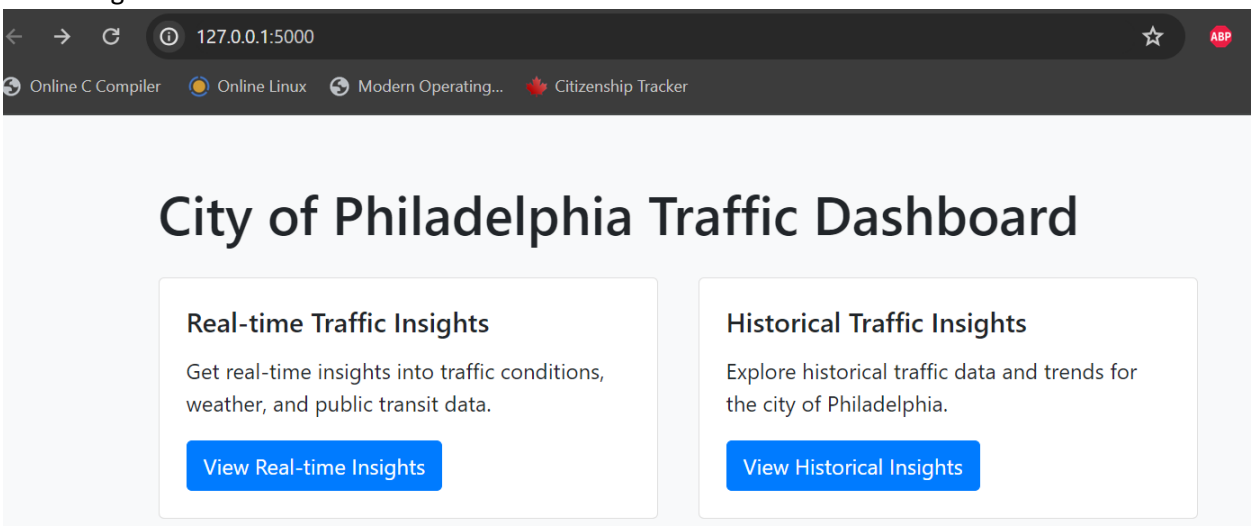
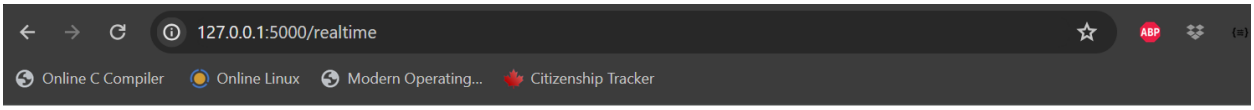


Figure 34: Test Case 10: Testing Navigation for Home Page (/home)

Clicking on “View Real-time Insights” button:



Real-Time Traffic Insights

Origin:

Destination:

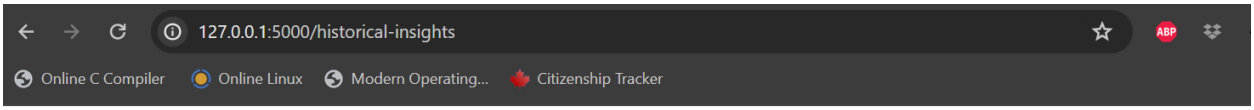
Desired Arrival Time:

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

Get Insights

Figure 35: Testing Navigation for Real-time Traffic Insights Page (/realtime)

Clicking on “View Historical Insights" button:



Historical Traffic Insights

Correlation Matrix

Explore the relationship between different traffic features.

View Correlation Matrix

Feature Distribution

View histograms showing the distribution of selected traffic features.

View Feature Distribution

Hourly Traffic Patterns

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

View Hourly Traffic Patterns

Monthly Traffic Patterns

Explore the average total time stopped during different months.

View Monthly Traffic Patterns

Figure 36: Testing Navigation for Historical Traffic Insights Page (/historical-insights)

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:

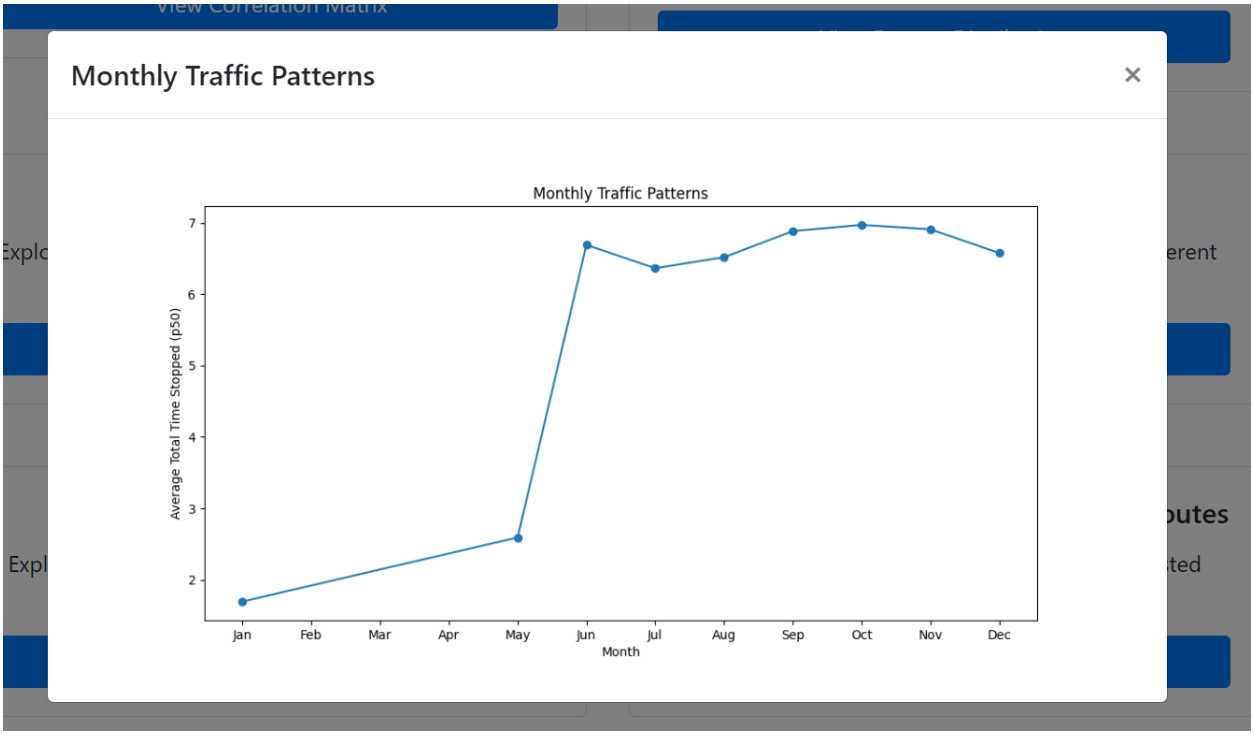


Figure 37: Test Case 11 – Testing links to open Graph of Monthly Traffic Patterns

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.

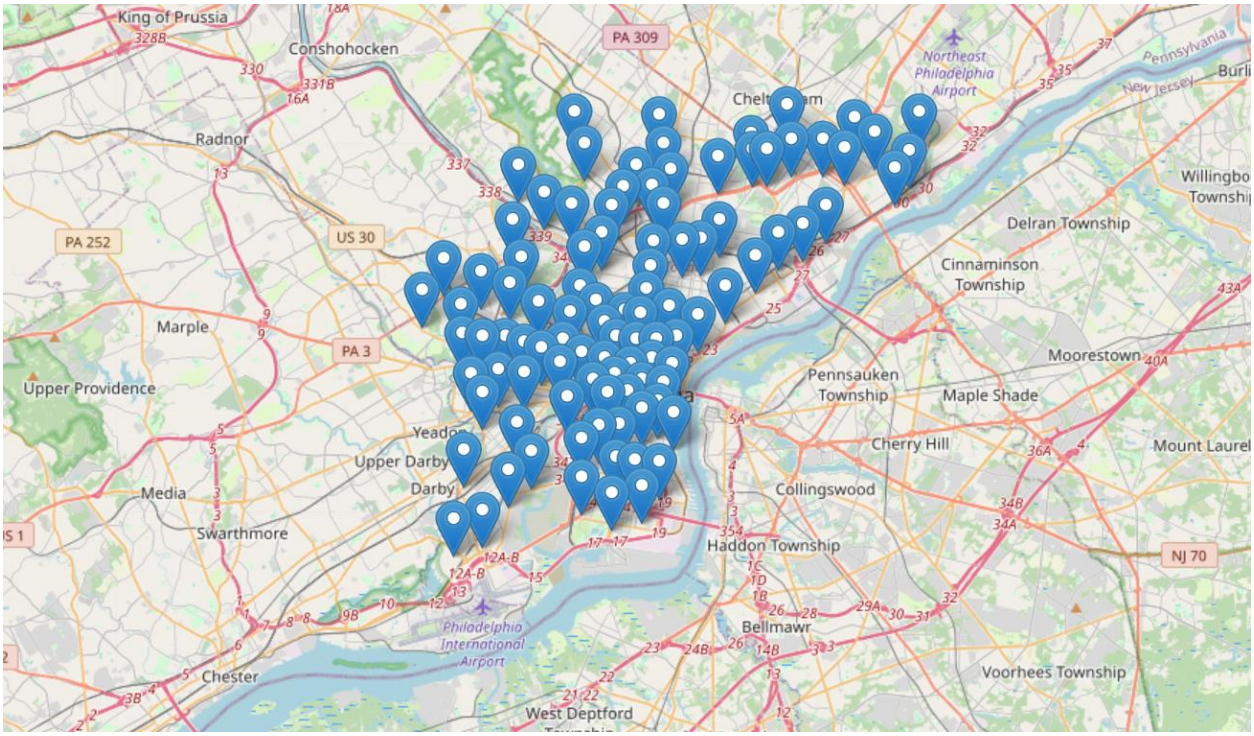


Figure 38: Test Case 12 – Testing Clustered Traffic Map showing congestion on the city map

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 this project, Unit testing is employed 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.

```python
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: [])
```

Figure 39: Test Case 13 – Unit test function to get Alternative Routes

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')
```

Figure 40: Test Case 14 – Unit test function to get Public Transit Data

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.

```python
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: [])
```

Figure 41: Test case 15 – Unit test function to get Traffic Incidents around the area

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)
```

Figure 42: Test Case 16 – Unit Test Function to get 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)
```

Figure 43: Test Case 17 – Unit test function to get traffic data (both valid, and invalid inputs)

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)
```

Figure 44: Test Case 18 – Unit Test function to suggest 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)
```

Figure 45: Test Case 19 – Unit test function to get optimal departure time with future arrival 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.

```
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)
```

Figure 46: Test case 20 – Unit test function to get no Alternative routes for Straight path

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.

```
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)
```

Figure 47: Test Case 21 – Unit test function to get no Traffic incidents in remote location

All the tests were successful:

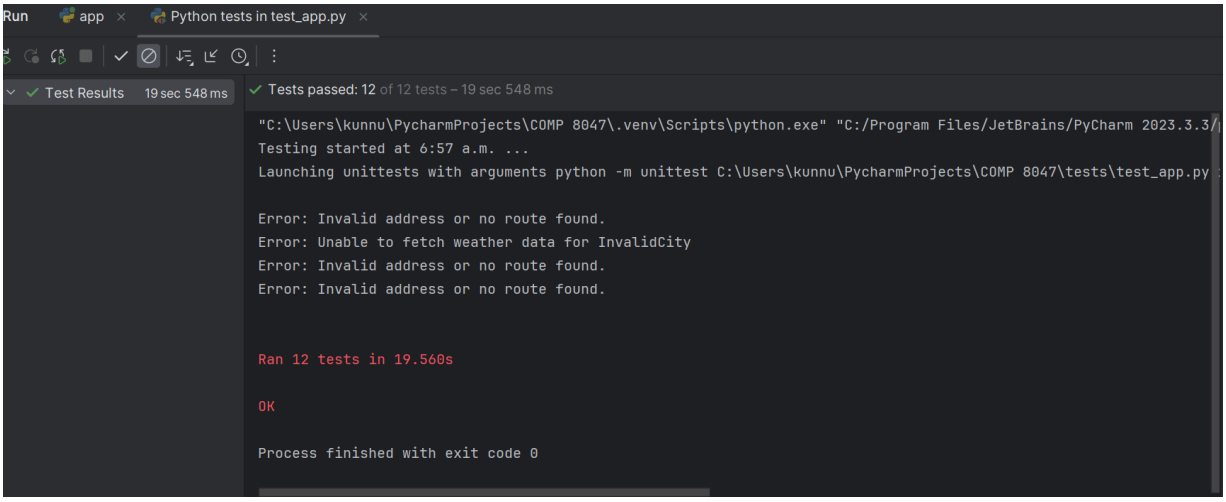


Figure 48: All 12/12 unit test cases were successful

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)
```

Figure 49: Test Case 22 – Integration testing function for home page (successful request)

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)
```

Figure 50: Test Case 23 – Integration testing function for Real-time insights page (successful request)

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)
```

Figure 51: Test Case 24 – Integration testing function for Historical Insights page (successful request)

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)
```

Figure 52: Test Case 25 – Integration testing function for insights - POST page (successful request)

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:

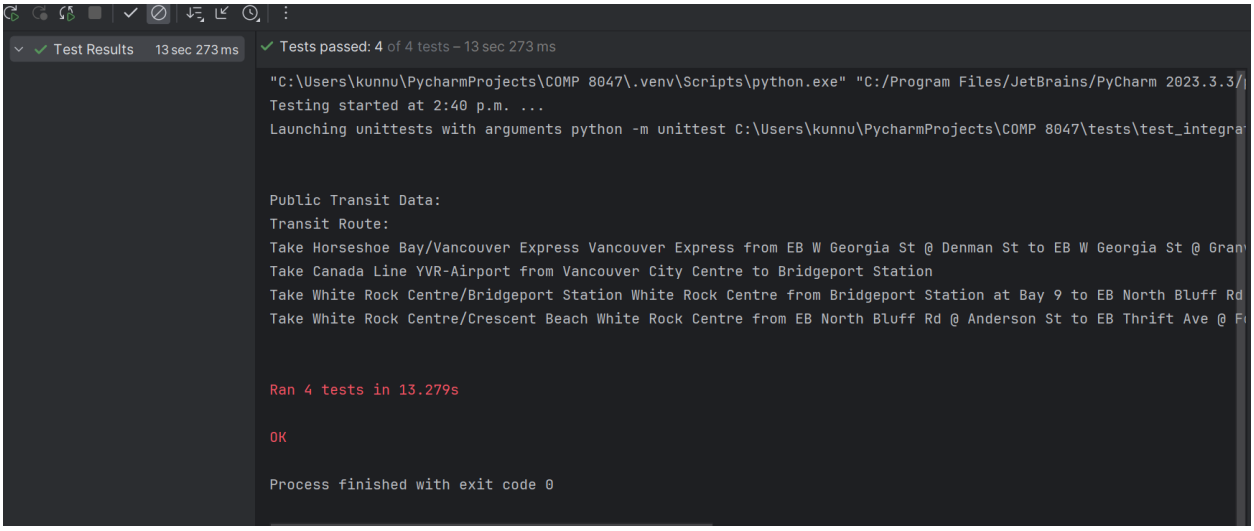


Figure 53: All 4/4 Integration tests passed

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
```

Figure 54: Stopping the application from running (exit code 0)

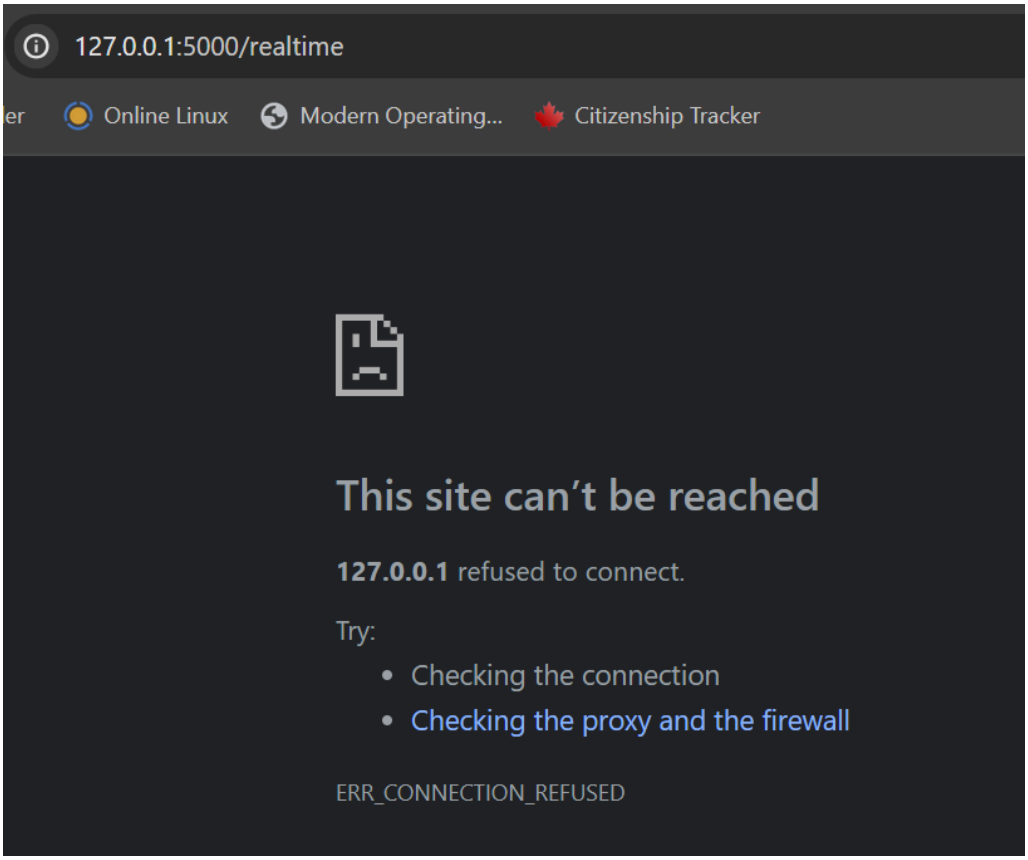


Figure 55: Site cannot be reached error message when the network is broken by pausing the app.py

2.8 Predictive Analysis

Data Overview (Source: https://www.kaggle.com/code/fayzaalmukharreq/traffic-congestion/notebook)

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 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).

```python
# 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])
```

Figure 56: Loading, Sampling, and Data Splitting

Loading dataset (merged_df), sampling that dataset (to reduce the actual size and increase the training speed of the model), defining the target feature, and printing the size of training and test set

- **Linear Regression Model**
  A Linear Regression Model is a machine learning statistical model/tool which is used to develop a relationship between target variable (dependent variable) and one or more independent variables. This model develops a linear equation showing how the dependent variable changes when the independent variables are changed and thus predicting that target/dependent variable.
  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)
```

Figure 57: Training Linear Regression Model on the training data and calculating error

- **Model Evaluation (Linear Regression Model)**
  MAE (Mean Absolute Error): This measures the average absolute difference between the predicted values and the actual values in a dataset, therefore the more the mae the more the error rate, and the poorer the prediction.
  RMSE (Root Mean Squared Error): The RMSE is the square root of the average squared difference between predicted values and actual values. This is easier to optimize, the more this error, the worse the prediction.

  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
```

Figure 58: Results of training Linear Regression Model

- **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.

Figure 59: Feature Importance in Linear Regression

Using the following code:

```
# 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)

# Visualize feature importance for Linear Regression
feature_importance_linear = linear_model.coef_
plt.figure(figsize=(10, 6))
plt.barh(features, feature_importance_linear)
plt.xlabel('Feature Importance')
plt.ylabel('Feature')
plt.title('Feature Importance for Linear Regression')
plt.show()
plt.savefig('linear-regression.png')
```

Figure 60: Bar chart to visualize feature importance for Linear Regression

- **Random Forest Regressor Model**
  As the name suggests, Random Forest regressor model is a machine learning algorithm used to predict the target variable in which the results from the multiple different decision trees combined are averaged to provide better and accurate results.
  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.

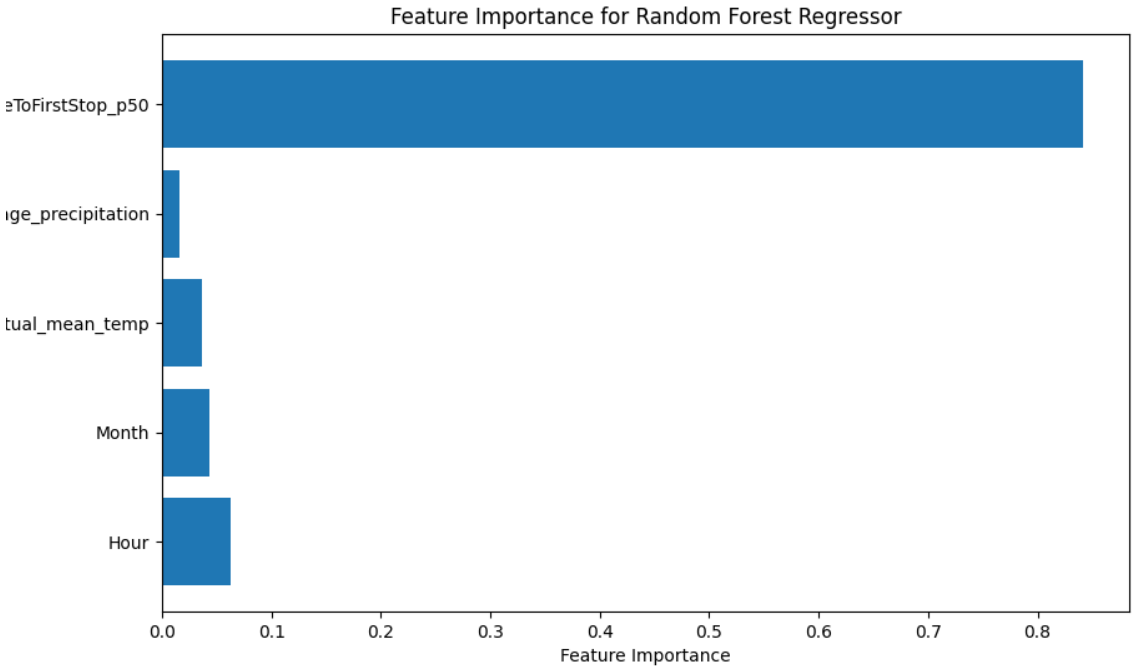


Figure 61: Feature importance in Random Forest Model

```python
# 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)
```

Figure 62: Training Random Forest Regressor Model on training data and calculating error

- **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
```

Figure 63: Output/results of training Random Forest regressor Model

- **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

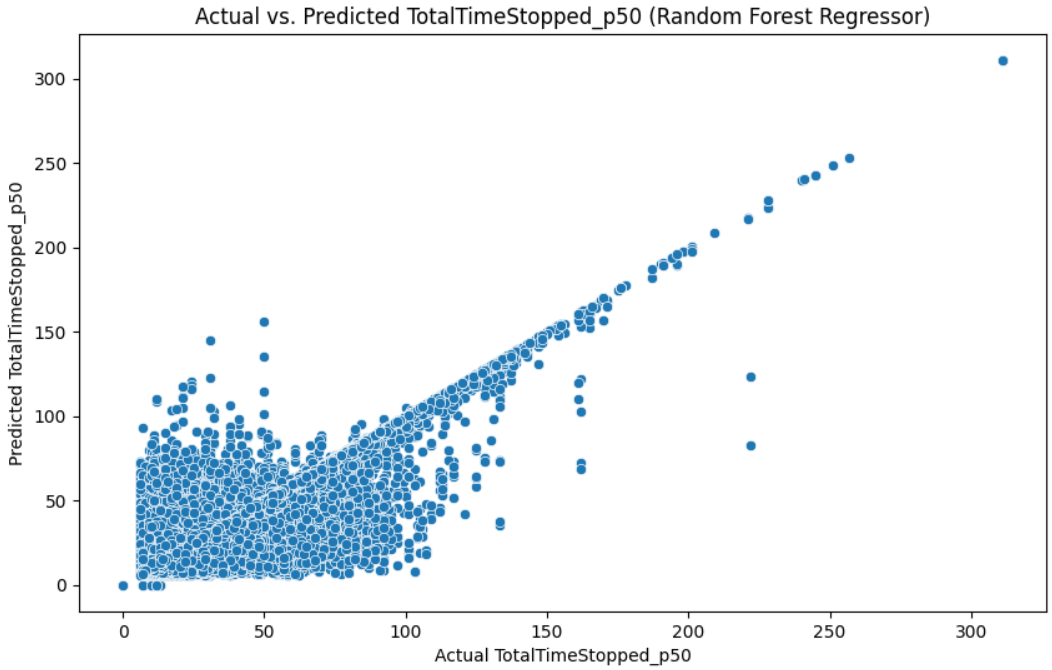


Figure 64: Actual vs Predicted target variable in Random Forest Model

Using the following code:

```
# Visualize predicted versus actual TotalTimeStopped_p50 values using scatter plot for Random Forest
plt.figure(figsize=(10, 6))
sns.scatterplot(x=y_test, y=y_pred_rf)
plt.xlabel('Actual TotalTimeStopped_p50')
plt.ylabel('Predicted TotalTimeStopped_p50')
plt.title('Actual vs. Predicted TotalTimeStopped_p50 (Random Forest Regressor)')
plt.show()
plt.savefig('actualVsPredicted.png')
```

Figure 65: Code to visualize Scatter plot of Actual vs Predicted variable in Random Forest Model

2.9 Methodology

The methodology adopted for the development of the "Traffic Congestion Reduction and Management System" includes various processes such as 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 uses 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 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 deletion of the entire rows, taking mean/average to fill in the missing values is used to handle the missing values
- Normalization and Standardization: Numerical features are scaled to a common range or standardized to ensure consistent data representation and ensure model convergence.
- Avoiding Duplication: The dataset is checked for duplication and the duplicates are removed from the data to reduce memory and inconsistencies.

2. Exploratory Data Analysis (EDA):

Descriptive Statistics:

Tools of Statistics such as Mean, variance, Standard Deviation, central tendency are used to explore and understand the data for further analysis.

Data Visualization:

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

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 created based on based on the initial information about the topic and exploring the data further to create insights. Models are further trained to predict the target variable based on the independent variables in the data.

Dimensionality Reduction:

Due to large number of features and heavy amount of data, various techniques such as Principal Component Analysis (PCA) or feature selection algorithms like Recursive Feature Elimination (RFE), are used to reduce the number of features and avoid problems making the data smoother and cleaner to work. Both these techniques remove the unnecessary and irrelevant features or dimensions without losing important information to reduce training time and increase efficiency.

Text and Categorical Data Processing:

It is important to work with numerical data to create visualizations and better understanding, thus the textual and categorical values are converted to numerical format using techniques such as one-hot encoding, label encoding, or embedding. One hot encoding is a pre-processing technique used to encode all the categorical values of the data into binary format in which value of 1 indicates the presence while 0 indicates the absence of that dimension or value

4. Model Development and Evaluation:

Model Selection:

A diverse range of machine learning algorithms such as regression models, decision trees, ensemble methods (e.g., Random Forest, Gradient Boosting), and deep learning architectures were considered for predictive analysis and model development and Linear Regression and Random Forest Model were chosen 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., 80:20 ratio for training and test sets respectively). The training set was further divided into small samples due to the very large size of the dataset and to improve training speed. Models were then trained on the training data and evaluated using cross-validation techniques to evaluate their generalization performance.

Performance Metrics:

Various performance metrics, such as accuracy, precision, recall, F1-score, ROC-AUC, and Mean Absolute Error (MAE), were calculated to evaluate the models' predictive performance and error rate in order to find out which model predicts the target variable better and accurate.

5. 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 for Public Transit data.

6. Technologies to Be Used:

Languages:

The project mainly used Python for real-time data analysis and web development. Python is an object oriented language which covers wide variety of tasks such as data analysis, web development, machine learning, and artificial intelligence.

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 very important roles in different aspects of the project, including data retrieval, preprocessing, visualization, and machine learning model development. By using these tools, the project can efficiently analyze traffic and weather data, develop predictive models, and visualize insights for effective decision-making and traffic management.

7. Web Development:

Frameworks like Flask are used 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.

8. 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.

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.

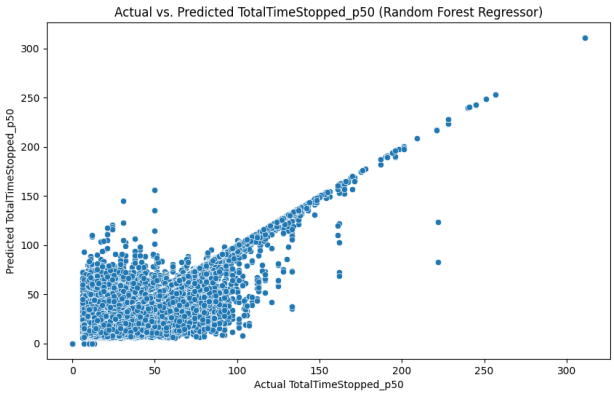


Figure 66: Scatter Plot of Actual vs Predicted TotalTimeStopped_p50 values when trained with Random Forest Model

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

9. Predictive Analysis

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

Why the model was trained:

- 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.

- Having knowledge of 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.

Overall, the trained model help 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.

10. Algorithms used:

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

- Linear regression is a statistical method used to develop 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 discuss about different aspects of the implementation, including performance, functionality, security, usability, scalability, maintainability, and associated limitations or challenges. Here's a detailed breakdown:

Performance:

By extensively testing the performance evaluations of API integration, data processing, and graph generation, better understanding of efficiency, scalability, and potential bottlenecks of the application can be obtained. This will help improve the overall speed and user experience of the app.

- API Integration: The app uses 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 how the API calls to the third party affects the response time in the app.
- 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. Measure the memory usage and processing time required to aggregate, filter, and analyze large volumes of data collected over time.
- 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 checking how well the app handles weather data, traffic info, and public transit details, it can be made sure that it meets user needs and helps them plan their trips better.

- Weather Data Retrieval: Checking how well the app provides the weather data of the particular city using the OpenWeatherMap API. The data needs to be checked for accuracy and reliability. Verify the accuracy and reliability of weather information, including temperature, weather conditions, wind speed, humidity, visibility, sunrise time, and sunset time.
- Error Handling: Checking how well and gracefully the app handles the errors such as missing inputs by the users, or using wrong inputs to get information related to traffic or weather. The app should provide detailed and informative error messages and guide users to enter valid inputs.
- Traffic Insights: Checking how accurate the traffic data is obtained when the valid inputs are entered by the user to get the real time traffic data, alternative routes from origin to

destination, traffic incidents, and Public transport information. Alternative routes should provide additional information such as travel time, distance, and traffic conditions. The Public Transport route should be clear and concise with the directions and step by step instructions, which makes it easier for users to follow.

Security:

By focusing on these security measures, the app can be more secure and reduce the chances of unauthorized access, data breaches, and other security related problems

- API Key Management: The app should securely manage API keys for services like OpenWeatherMap and Google Maps. It's important to keep these keys safe to prevent unauthorized access or misuse. Adding Restrictions to these APIs from the source is a great first step to follow.
- Secure Storage: Verify that sensitive information, such as API keys, is securely stored and not directly exposed in the source code or configuration files
- 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.

Usability:

By focusing on usability aspects such as user interface design, accessibility, and error handling, it 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: Check how easy the web app interface is to use with Flask templates. Make sure users can easily get traffic and weather info.
- Clarity and Intuitiveness: Look at how clear and easy the interface is. Check the layout, design, and organization of forms, buttons, and menus. The interface should be simple, easy to follow and should work well on different devices and screen sizes..
- Error Handling: Make sure the app handles errors well and tells users when something goes wrong
- Input Validation: Check if the app properly validates user inputs like city names and addresses. Show clear error messages to help users fix their mistakes.
- API Error Handling: Ensure that the app handles API errors smoothly, like server errors or network issues. Give users clear error messages and log details for entering valid and correct inputs for desired results.
- Feedback Mechanisms: Provide real-time feedback to users, like loading indicators or success/error messages, to show the status of their requests. Make sure the feedback is clear and helpful.

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.

- Code Structure: Look at how the Flask app's code is organized. Make sure it's easy to read, update, and add new features without breaking anything.
- Modularity: Check if the code is broken down into reusable parts like functions and modules. Different parts of the app (like getting data, processing it, and showing it) should be in separate modules.
- Readability: Make sure the code is easy to read. Use good variable names, comments, and documentation so other developers can understand it easily.
- Maintainability: See how easy it is to update and add new features. The code should be flexible to support changes without causing problems.
- Data Handling: Make sure the app can handle more data without slowing down.
- Caching: Use caching to store frequently accessed data so the app doesn't have to fetch it repeatedly. This can make the app faster and more efficient.
- Filtering and Aggregation: Optimize how the app processes large amounts of data. Use techniques like database indexing and query optimization to handle data better and faster.

Limitations and Challenges:

By looking at the limitations and challenges associated with API rate limits, data accuracy, and handling heavy historical data, stakeholders can get better understanding of the issues that might come up using the app. This helps in making smart decisions about improving performance, reducing risks, and using resources well to keep the app effective and reliable

- API Rate Limits: Discuss any limitations or challenges imposed by API rate limits, or usage policies of external service providers. Think about ways of improving API usage and minimizing the impact of rate limiting on application functionality.
- Reliability of Third-Party Data: Mention that data from APIs like OpenWeatherMap and Google Maps might not always be accurate or complete, which can affect the app's insights.
- Implications of Inaccurate Data: Explain how wrong data can lead to bad recommendations or decisions, reducing user trust and satisfaction.
- Handling Heavy data: Handling heavy historical data presents several challenges, particularly in terms of processing, cleaning, visualization, and sampling. This involves using 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: Handling large amounts of historical data needs a lot of computational power and memory usage. Things like parsing data, transforming it, and making sure it's good quality take up a lot of processing power and memory.
- Time-Intensive Operations: Cleaning and processing big datasets can take a lot of time. Tasks like normalizing data, removing duplicates, and fixing errors can be slow and might delay getting the data ready.
- Trade-offs: When working with historical data, there's a trade-off between how clean the data is and how fast you can get it done. It's important to find a good balance to make sure the data is accurate and ready for analysis without taking too long.

Visualization and Sampling:

- Resource-Intensive Visualization: Visualizing large amounts of historical data can be tough on computing resources and can slow down rendering times. Making graphs, charts, and maps from big datasets needs a lot of processing power and memory, which can affect how responsive and easy-to-use the visualization interface is.
- Sampling Techniques: To make visualization faster without losing much accuracy, sampling techniques can be used to increase speed. Methods like random sampling can help shrink the dataset size while keeping the overall data pattern.
- Importance of Representative Samples: It's really important to make sure the samples picked represent the entire data set. If the samples are biased or not representative, the insights and conclusions from the visualization could be wrong or misleading.
- Consideration of Implications: There are implications on accuracy when the data is sampled because the whole dataset is not used to create those visualizations. While sampling makes visualization and analysis faster, it can also cause sampling errors or miss key patterns and outliers in the full dataset. It's important to balance data reduction and insight accuracy to make smart decisions about visualization strategies.

In summary, a full review of the implications of the implementation provides valuable insights into the application's strengths, weaknesses, and areas for improvement. Using this information, stakeholders can make informed and smart decisions to gather the useful and reliable 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 provided valuable insights into new trends 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: Using data visualization libraries such as Matplotlib, Seaborn, Plotly, and Folium provided insights into their pros and cons for generating graphs, charts, and maps.

Each library offers unique features and functionalities, allowing 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 was a bonus to the Project proposal and brought a significant enhancement in data visualization capabilities, enabling stakeholders to gain deeper insights from the analyzed data through interactive dashboards and visualizations.
- 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.

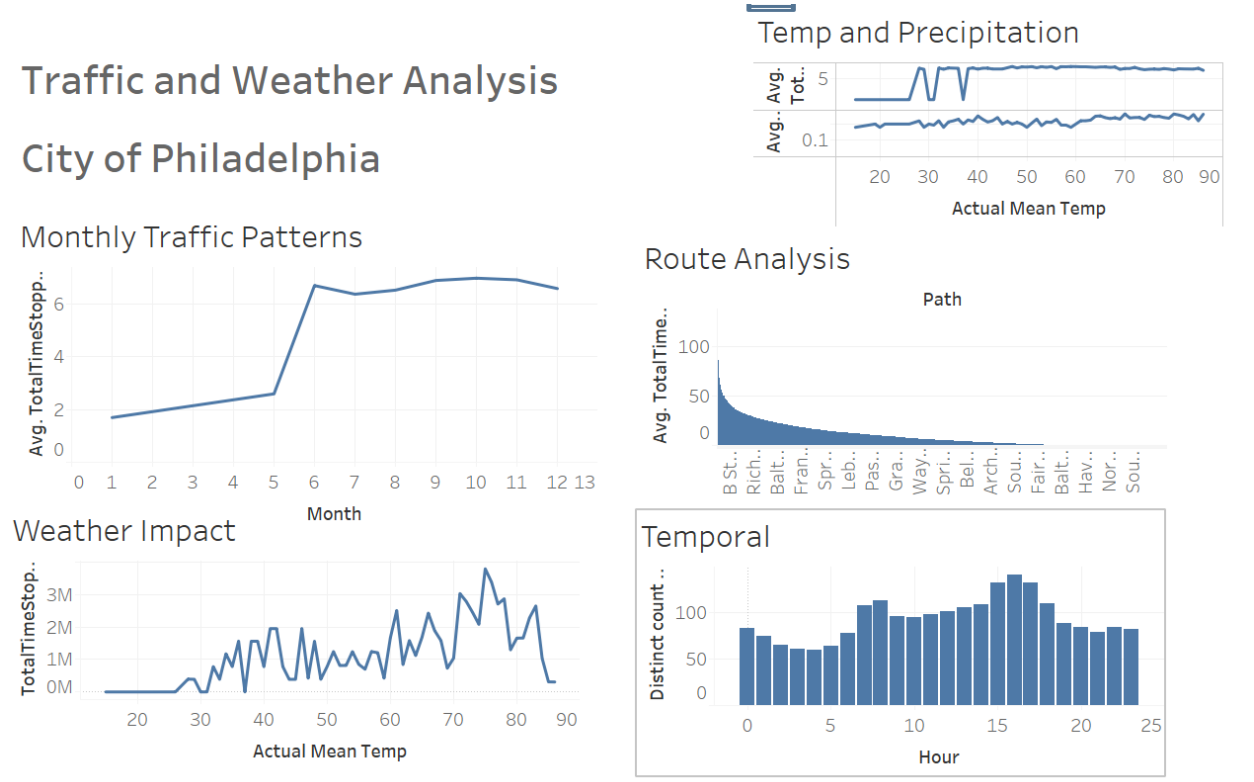


Figure 67: Tableau Dashboard showing 5 Traffic and Weather visual Insights

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.

- Python Scripting for UI/UX Enhancement: Python scripting was used to enhance the user interface (UI) and user experience (UX) of the web-based dashboard and its data science features. 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 customized solutions to enhance the overall effectiveness of the dashboard in appealing insights and improving decision-making processes.
- Machine Learning: Using Machine learning models such as Linear Regression Model, and Random Forest Regression model to train the data to predict the target variable and then checking the error rate with MAE and RMSE was a part of great search and use of new technology.
- Geospatial Analysis: Dealing with 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.
- Feature Selection: Feature selection played an important 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 led to improved model accuracy and prevented overfitting.

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: Adding and combining 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.
- 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.
- 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 plan to improve the project's capabilities and provide users with more valuable and personalized experiences.

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 real-time data from diverse sources, including APIs, and combining them with web interface along with historical and predictive insights is a big 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.

Merging data from various sources and keeping it consistent and accurate is tough. One has to come up with data transformation and reconciliation processes to combine 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.

Tackling these challenges will need a lot of research, problem-solving, and experimentation, pushing one's technical skills and getting ready for real-world tasks in data analysis, machine learning, web development, and traffic management, beyond is learnt in class.

3. Conclusion

Combining predictive analysis with real-time data integration in a Flask web application is a big step for improving urban mobility and decision-making. This project mixed data science with web development to give useful insights and features.

Predictive Analysis:

The predictive part of the project focused on traffic forecasting in Philadelphia, aiming to estimate Total Time Stopped at intersections. Using machine learning models like Linear Regression and Random Forest Regressor, the project predicted traffic congestion based on time, weather, and historical traffic patterns.

The project included feature selection, model evaluation, and visualization techniques to understand the relationships between different factors and traffic congestion levels. The Linear Regression model had a Mean Absolute Error (MAE) of 6.61 and a Root Mean Squared Error (RMSE) of 10.87, showing moderate predictive ability. The Random Forest Regressor performed better, with an MAE of 0.0012 and an RMSE of 0.0134, showing its effectiveness in capturing complex traffic patterns. Visualizations like correlation matrices, feature distributions, and traffic patterns helped stakeholders see trends and make informed decisions. Interactive plots with Plotly and Folium added to the analysis by showing traffic data and spatial clusters.

Historical Insights:

Cleaning and processing historical traffic data was the first step in getting useful insights. By carefully handling outliers, missing values, and inconsistencies, the dataset's integrity and reliability were ensured. Exploratory data analysis (EDA) techniques like correlation analysis, feature distribution visualization, and temporal traffic pattern analysis provided valuable historical insights. Correlation matrices showed the relationships between different traffic metrics, weather variables, and time factors, highlighting important correlations and dependencies. Feature distribution plots showed the distribution characteristics of key variables, spotting potential outliers and anomalies. Temporal traffic pattern analysis, including hourly and monthly traffic patterns, revealed recurring trends and seasonality, helping with strategic planning and resource allocation.

Flask Web Application:

The Flask web application connected predictive insights with real-time data integration, giving users dynamic traffic information and actionable insights. API integrations with services like Google Maps and OpenWeatherMap allowed the application to get real-time weather updates, traffic conditions, alternative routes, traffic incidents, and public transit data. The user interface, built with HTML templates and JavaScript for interactivity, provided an easy platform for users to input origin-destination pairs and desired arrival times. The backend, managed with Python functions and Flask routes, handled data retrieval, processing, and presentation, ensuring a smooth user experience.

Conclusion and Future Directions:

This project has enhanced skills in data science and web development while showing the potential of combining predictive analytics with real-time data integration for urban mobility management. The insights from predictive models, real-time data, and historical traffic analysis can guide decisions in traffic management, infrastructure development, and public transit planning.

Looking ahead, there are several ways to improve and expand the project. Adding more features like road conditions, special events, and social factors to the predictive models could improve accuracy and robustness. Enhancing the web application with personalized user profiles, route optimization algorithms, and proactive traffic alerts could make the user experience better and more useful.

This project demonstrates the power of data-driven decision-making in improving urban mobility and creating smarter, more sustainable cities. By combining predictive analytics, real-time data integration, and historical insights, this project sets the stage for a future with less traffic congestion and better transportation systems.

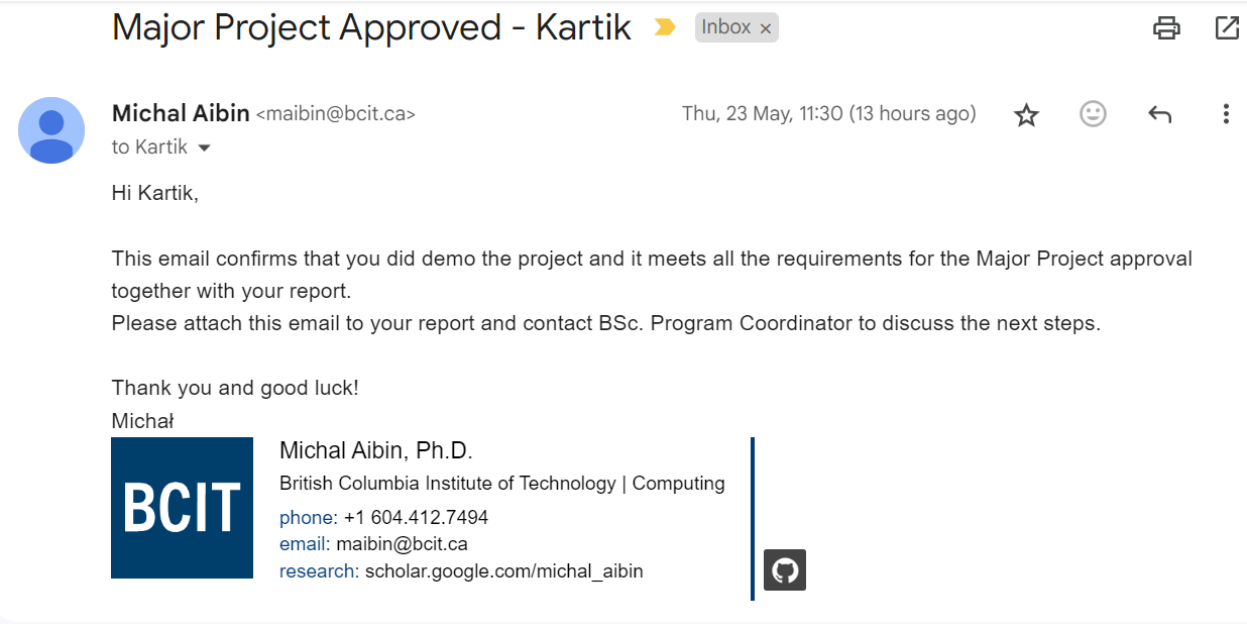
4. Appendices

SME's Approval

Please find the Attached Approval from SME in the screenshot shown below:

SME: Michal Aibin

Date Approved: 2024-05-23



Major Project Approved - Kartik Inbox ×

Michal Aibin <maibin@bcit.ca> Thu, 23 May, 11:30 (13 hours ago)

to Kartik

Hi Kartik,

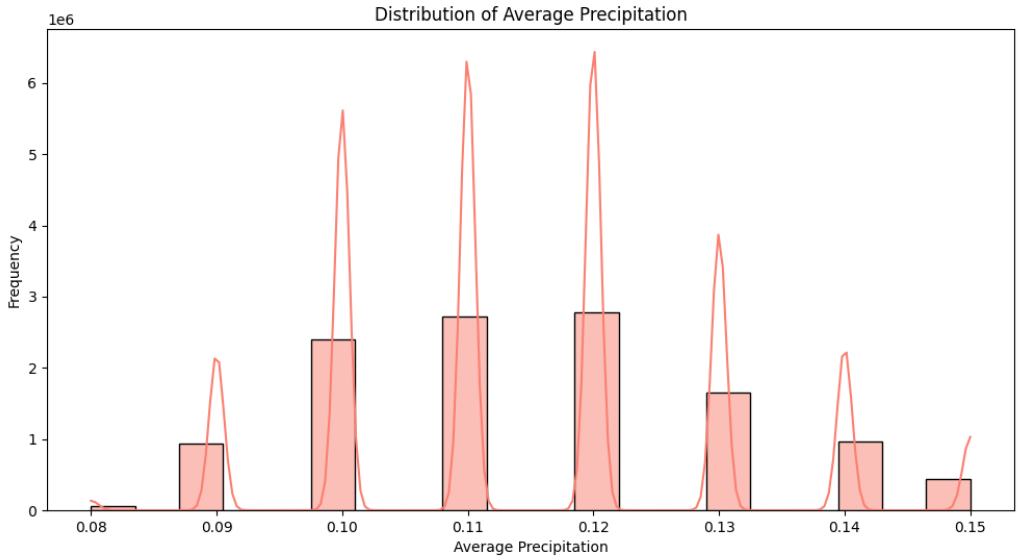
This email confirms that you did demo the project and it meets all the requirements for the Major Project approval together with your report.
Please attach this email to your report and contact BSc. Program Coordinator to discuss the next steps.

Thank you and good luck!
Michał

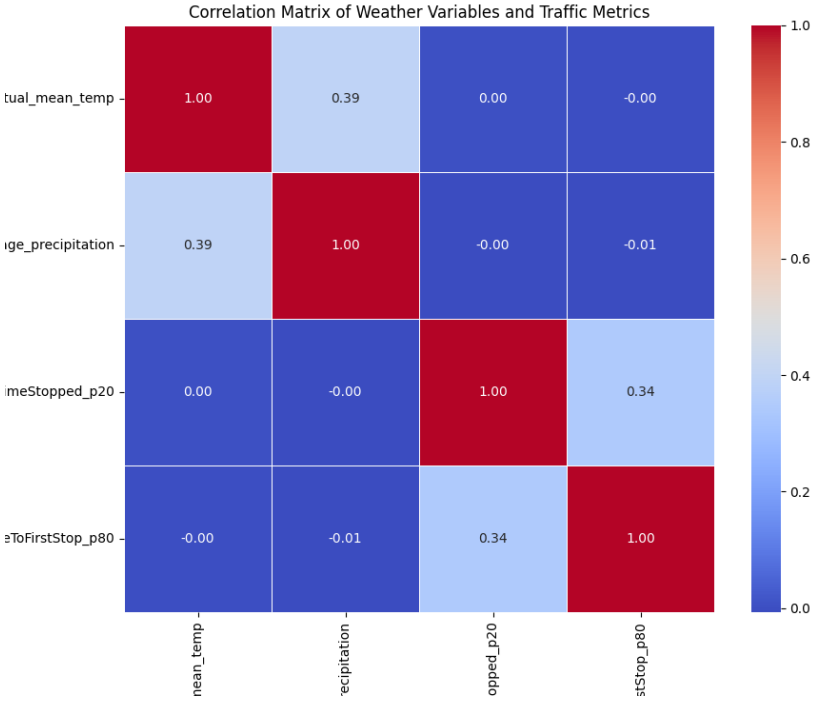
BCIT

Michal Aibin, Ph.D.
British Columbia Institute of Technology | Computing
phone: +1 604.412.7494
email: maibin@bcit.ca
research: scholar.google.com/michal_aibin

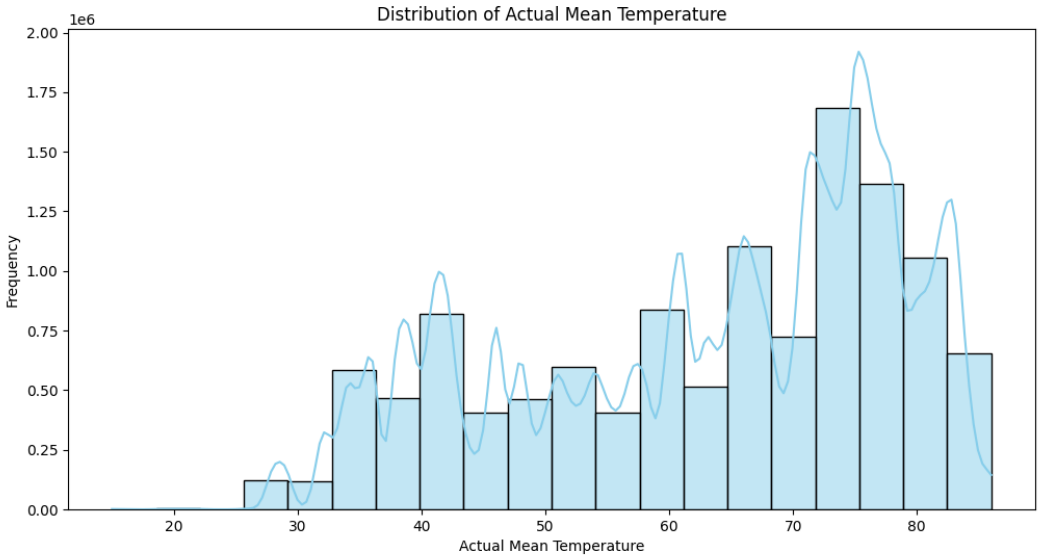
Graphs and Plots Missing from the main Body:



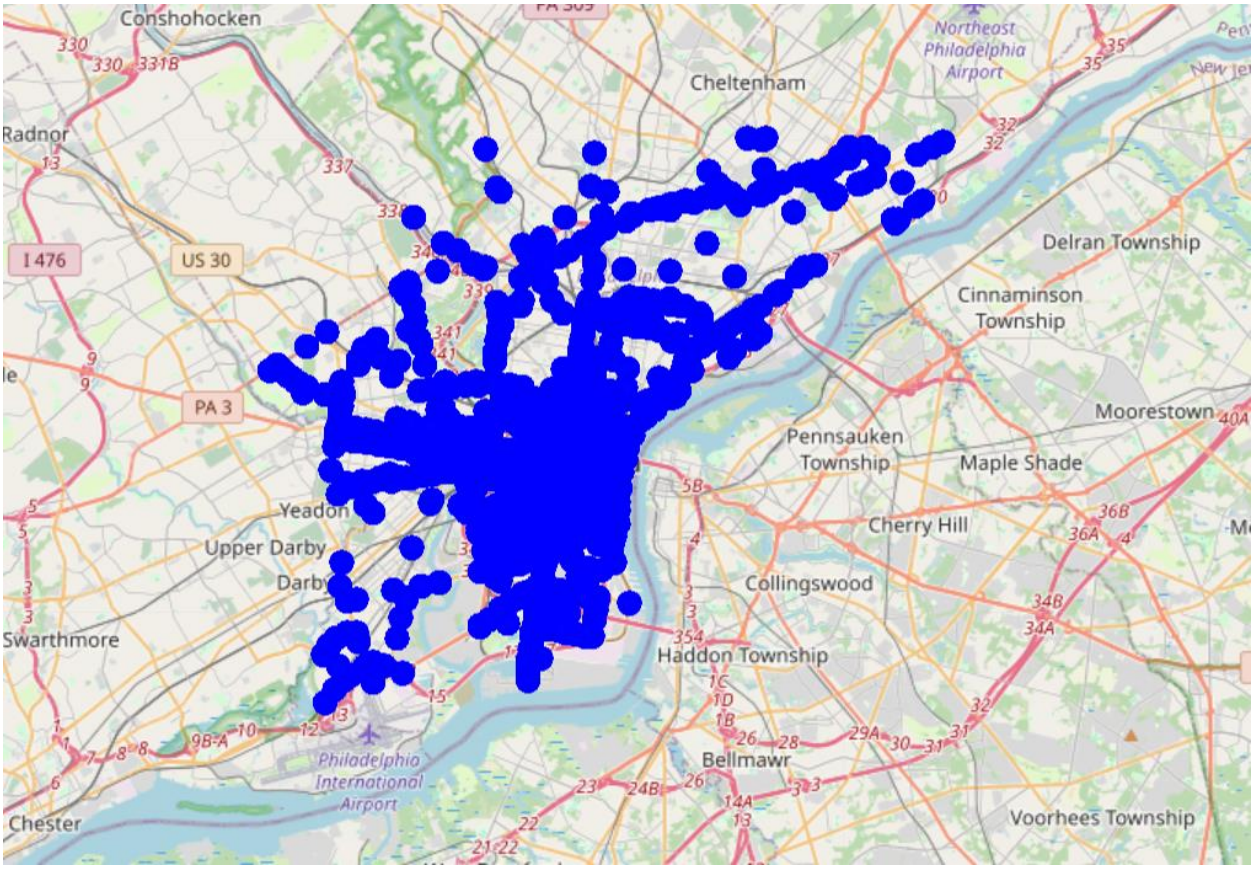
The graph shows Distribution of Average Precipitation from Weather dataset.



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.
- City-Dashboard.twb: Dashboard made in Tableau provides 5 different insights using historical traffic and weather datasets.

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
- 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/**

6. Change Log

Revision 1 (2024-05-19)

These are some of the Changes made to the whole Project Report Document for the Revision of the Project Report based on the suggestions from the Project SME:

- Removal of Markup Language symbols
- Adding detailed descriptions
- Addition of titles to the figures
- Added List of Figures section in the report

Initial Version (2024-05-14)