Machine Learning and Neural Networks Project

University of Jordan

King Abdullah II School for Information Technology

Machine Learning and Neural Networks Course

Spring Semester 2025

Supervisor: Dr. Ali Alrodan

Number	Student ID	Student Name
1	0224969	Noor Mahmoud Ali Almazaydeh
2	2220728	Ola Ali Naim Alkurdi
3	2221775	Sahar Abdulhay
4	0226202	Ruaa Ali Naim Alkurdi

Introduction:

1. Time Series

Air pollution is a critical environmental issue that poses serious health risks, particularly in industrial and urban regions. Among the harmful pollutants, **benzene (C_6H_6)** is especially dangerous due to its carcinogenic nature and long-term impact on human health. Prolonged exposure to benzene has been associated with respiratory diseases, blood disorders, and increased cancer risks. Predicting benzene concentration in the air is vital for early warning systems, public health interventions, and effective air quality management.

1.1Objectives

The objective of this project is to apply time series prediction techniques to forecast benzene (C_6H_6) levels based on environmental and sensor data. Specifically, we aim to compare the performance of three different models: **Echo State Network (ESN)**, **Long Short-Term Memory (LSTM)**, and **Bidirectional LSTM (Bi-LSTM)**.

We use a multivariate time series dataset collected from an urban monitoring station in Italy, made available through the UCI Machine Learning Repository. The dataset contains hourly measurements of various air quality indicators and meteorological variables, including pollutant concentrations, sensor readings, temperature, and humidity. Our goal is to evaluate how accurately each model can predict benzene concentration over time based on these features.

A. Time series

I. Dataset name: Air Quality UCI Dataset (from UCI Machine Learning Repository)
II. Number of records: 9357
III. Number of columns: 15

Each record includes various environmental and sensor-based measurements. The data includes a `Date` and `Time` column which were combined and converted into a proper datetime format to structure the data as a time series.

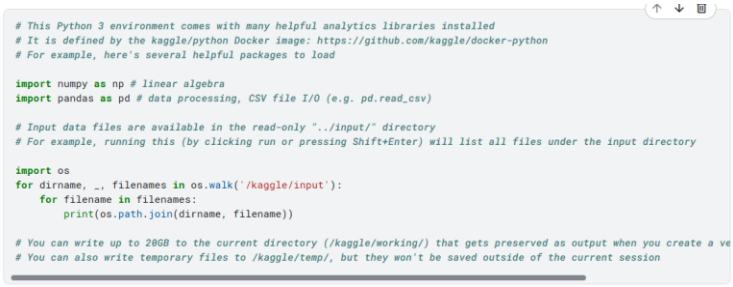
Environmental and Sensor Features:

- **CO(GT):** Carbon monoxide concentration (mg/m^3)
- **NMHC(GT):** Non-methane hydrocarbons (ppb)
- **C6H6(GT):** Benzene concentration (mg/m^3) ← **Target variable**
- **NOx(GT):** Nitrogen oxides (ppb)
- **NO2(GT):** Nitrogen dioxide (µg/m^3)
- **PT08.S1 - PT08.S5:** Sensor responses from metal oxide sensors
- **T:** Temperature (°C)
- **RH:** Relative Humidity (%)
- **AH:** Absolute Humidity

The target variable is **C6H6(GT)**, which represents the actual concentration of benzene in the atmosphere. This dataset enables the exploration of complex temporal relationships between atmospheric conditions and pollutant levels for the purpose of accurate forecasting.

Preprocessing Steps for air quality data

1. **Kaggle starter code:**



```
# This Python 3 environment comes with many helpful analytics libraries installed
# It is defined by the kaggle/python Docker image: https://github.com/kaggle/docker-python
# For example, here's several helpful packages to load

import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)

# Input data files are available in the read-only "../input/" directory
# For example, running this (by clicking run or pressing Shift+Enter) will list all files under the input directory

import os
for dirname, _, filenames in os.walk('/kaggle/input'):
    for filename in filenames:
        print(os.path.join(dirname, filename))

# You can write up to 20GB to the current directory (/kaggle/working/) that gets preserved as output when you create a ve
# You can also write temporary files to /kaggle/temp/, but they won't be saved outside of the current session
```

- Part 1: Importing Core Libraries
  import numpy as np # linear algebra
  import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)
- Part 2: Listing All Input Files

import os

for dirname, _, filenames in os.walk('/kaggle/input'):

  for filename in filenames:

print(os.path.join(dirname, filename))

- Part 3: Notes on File Storage (Just Comments)
  You can write up to 20GB to the current directory (/kaggle/working/)
  You can also write temporary files to /kaggle/temp/ Picture

2. Load required libraries for data processing

```
import pandas as pd
import numpy as np
from sklearn.preprocessing import MinMaxScaler
```

3. Loading the Dataset

```
# Load Excel file
df = pd.read_excel("/kaggle/input/air-quality-new/AirQualityUCI.xlsx", engine='openpyxl')
```

4. Removing Empty or Redundant Column

```
[11]: #Remove empty columns
      df.drop(columns=['Unnamed: 15', 'Unnamed: 16'], inplace=True, errors='ignore')
```

5.Combining Date and Time into a Single Timestamp

```
# Create datetime column as a timestamp index
df['Datetime'] = pd.to_datetime(df['Date'].astype(str) + ' ' + df['Time'].astype(str),
                                errors='coerce', dayfirst=True)
df.set_index('Datetime', inplace=True)
```

6. Feature Selection

```
# Define input features and the target column
input_features = ['PT08.S1(CO)', 'PT08.S2(NMHC)', 'PT08.S3(NOx)',
                  'PT08.S4(NO2)', 'PT08.S5(O3)', 'T', 'RH', 'AH']
target_column = 'C6H6(GT)'
```

- Select only the relevant features and the target
- Focus the model on meaningful sensor and environmental variables
- Drop Nans: Ensures clean input for scaling and modeling.

7. Feature Scaling

```
# Keep only the required inputs and outputs
df = df[input_features + [target_column]].dropna()
#Normalize values between 0 and 1
scaler = MinMaxScaler()
df_scaled = pd.DataFrame(scaler.fit_transform(df),
                         columns=input_features + [target_column],
                         index=df.index)
```

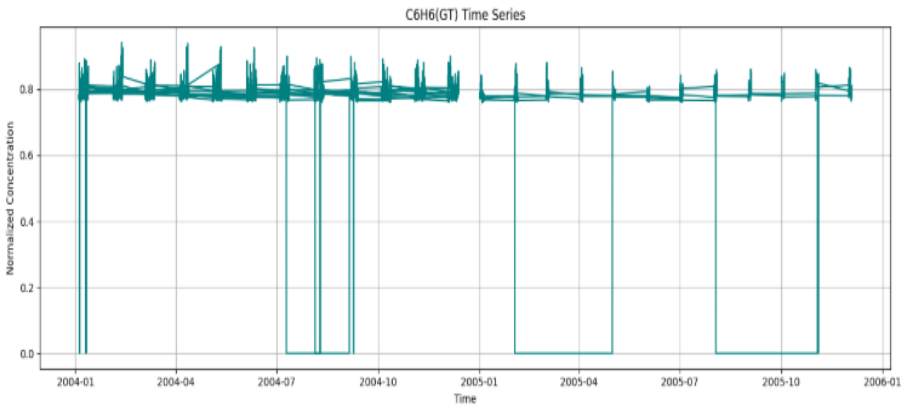
- Normalize features to the range [0, 1]

8. Time Series Plot of the Target (C6H6)

- This plot visualizes the normalized time series of benzene concentration (C6H6) over time to reveal trends and patterns in air pollution levels.

```
import matplotlib.pyplot as plt
import seaborn as sns

#===== 1. Time series plot of C6H6 concentration (Target) =====
plt.figure(figsize=(14, 5))
plt.plot(df_scaled.index, df_scaled['C6H6(GT)'], color='teal')
plt.title('C6H6(GT) Time Series')
plt.xlabel('Time')
plt.ylabel('Normalized Concentration')
plt.grid(True)
plt.tight_layout()
plt.show()
```

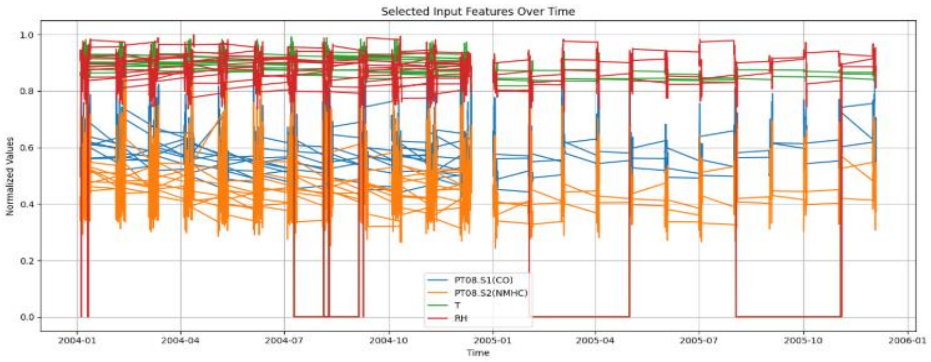


9. Time Series of Selected Inputs

- Understand their correlation with the target

- Observe temporal behavior of selected features

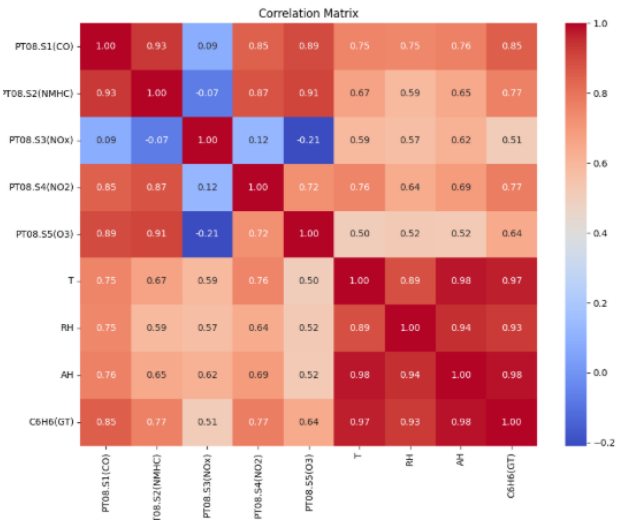
```
#===== 2. Time series plot of some Inputs =====
plt.figure(figsize=(14, 6))
for col in ['PT08.S1(CO)', 'PT08.S2(NMHC)', 'T', 'RH']:
    plt.plot(df_scaled.index, df_scaled[col], label=col)
plt.title('Selected Input Features Over Time')
plt.xlabel('Time')
plt.ylabel('Normalized Values')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```



10. Correlation Heatmap

- Quantify linear relationships between features and the target

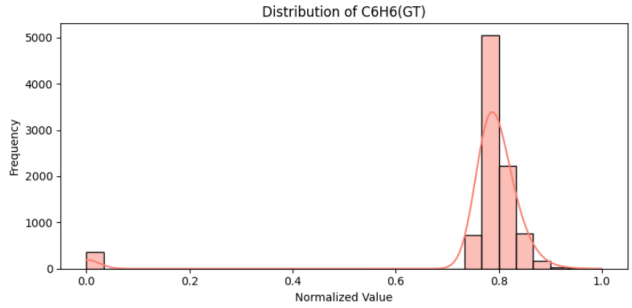
```
#===== 3. Correlation matrix (Heatmap) =====
plt.figure(figsize=(10, 8))
corr_matrix = df_scaled.corr()
sns.heatmap(corr_matrix, annot=True, cmap='coolwarm', fmt=".2f")
plt.title('Correlation Matrix')
plt.tight_layout()
plt.show()
```



11. Target Distribution

- Analyze distribution of the target variable

```
#===== 4. Distribution plot for C6H6 (GT) =====
plt.figure(figsize=(8, 4))
sns.histplot(df_scaled['C6H6(GT)'], bins=30, kde=True, color='salmon')
plt.title('Distribution of C6H6(GT)')
plt.xlabel('Normalized Value')
plt.ylabel('Frequency')
plt.tight_layout()
plt.show()
```



12. Sequence Preparation for Time Series Modeling

- Convert Data into Sequences for Supervised Learning (predict future benzene values)

```python
#Prepare the data sequence
#Number of time steps used

TIME_STEPS = 24

#Convert the data to sequences
def create_sequences(data, target_col__index, time_steps=TIME_STEPS):
    X, y = [], []
    for i in range(len(data) - time_steps):
        X.append(data[i:i+time_steps, :-1])  # جميع الأعمدة ما عدا الـ target
        y.append(data[i+time_steps, target_col_index])  # الهدف في المستقبل
    return np.array(X), np.array(y)

#Convert DataFrame to NumPy array
data_array = df_scaled.values
target_index = df_scaled.columns.get_loc("C6H6(GT)")

X, y = create_sequences(data_array, target_index, TIME_STEPS)

#Final shape of the data
print("X shape:", X.shape)  # (samples, time_steps, features)
print("y shape:", y.shape)  # (samples,)
```

13. Splitting the data into training, validation, and test sets with a ratio of 70%, 15%, and 15% respectively.

```python
#Splitting the data into training, validation, and test sets with a ratio of 70%, 15%, and 15% respectively.
train_size = int(0.7 * len(X))
val_size = int(0.15 * len(X))
test_size = len(X) - train_size - val_size

X_train, X_val, X_test = X[:train_size], X[train_size:train_size+val_size], X[train_size+val_size:]
y_train, y_val, y_test = y[:train_size], y[train_size:train_size+val_size], y[train_size+val_size:]

print("Training samples :", X_train.shape[0])
print("Validation samples:", X_val.shape[0])
print("Testing samples   :", X_test.shape[0])
```

```
Training samples : 6533
Validation samples: 1399
Testing samples   : 1401
```

Methodology:

1)Time Series:

1- LSTM

The LSTM model (Long Short-Term Memory) was used because it is one of the most popular models for handling time series data. It is characterized by its ability to retain long-term information, which makes it suitable for forecasting time series.

```python
model = Sequential()
model.add(LSTM(64, input_shape=(X_train.shape[1], X_train.shape[2]),
return_sequences=False))
model.add(Dense(1))
model.compile(optimizer='adam', loss='mse')
```

Input Layer: Sequential data input.

LSTM Layer : Captures temporal dependencies and patterns in the input features.

Dense Output Layer: A single neuron that outputs the predicted C6H6 value.

Loss Function: Mean Squared Error (MSE) was used, suitable for regression tasks.

Optimizer: Adam optimizer for faster and more stable convergence.

Hyperparameter

Hyperparameter	Value
LSTM Units	64
Optimizer	Adam
Loss Function	MSE
Epochs	50
Batch Size	32
Return Sequences	False

Experiment Setup

Tools & Libraries:

- **Python 3.11**
- **TensorFlow / Keras** for model creation
- **scikit-learn** for preprocessing and evaluation metrics
- **Matplotlib** for visualizations
- **Google Colab** environment

Justification:

- LSTM is ideal for capturing temporal relationships in air quality data.
- Normalization helped the model converge faster and avoid gradient instability.
- C6H6 was selected as the target due to its relevance as a toxic air pollutant.

Results & Analysis

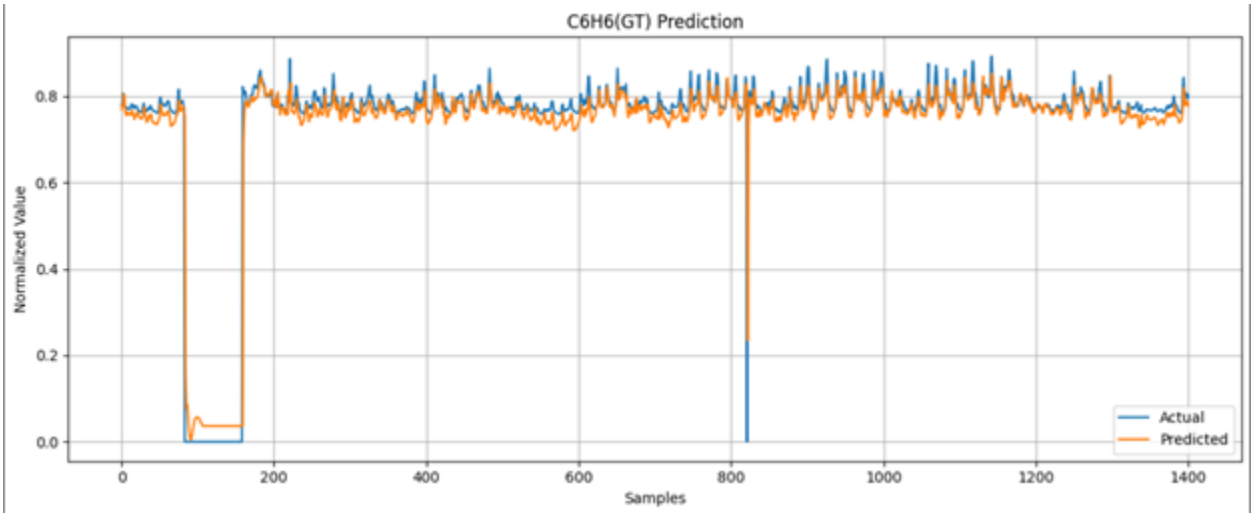
Evaluation Metrics

Metric	Value(validation, testing)
MSE	0.0044, 0.0019
RMSE	0.0667, 0.0443
MAE	0.0222, 0.0153
R^2 Score	0.9226, 0.9398

The **R^2 score of 0.9398** indicates that the model explains about **94% of the variance** in the true benzene concentrations, which reflects strong predictive performance.

Figure Analysis

The figure below shows the predicted vs. actual C6H6(GT) plot.

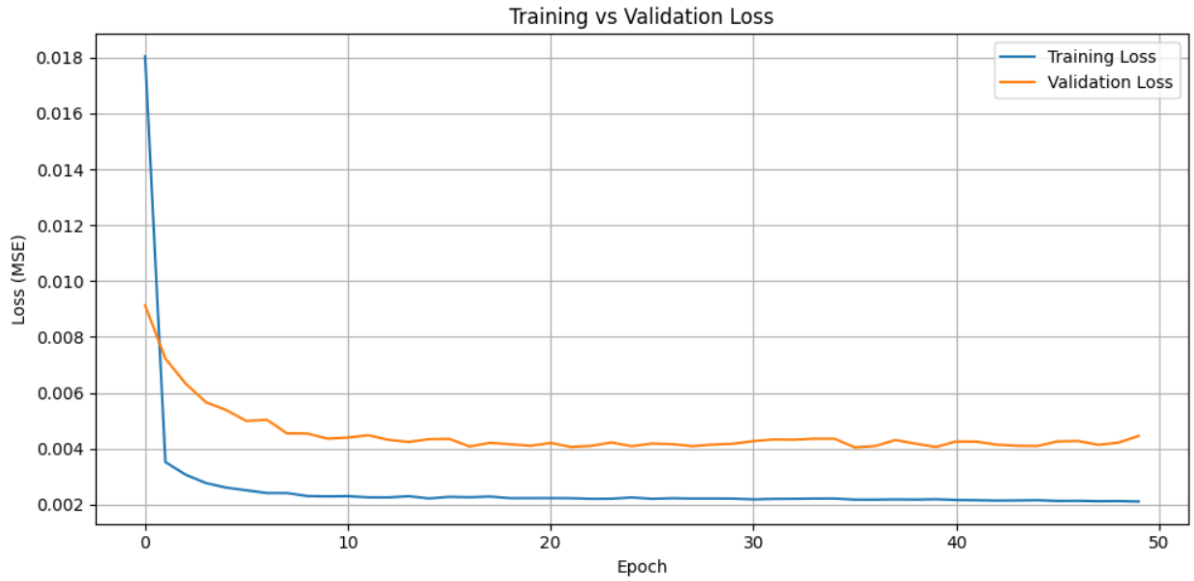


Interpretation of the Plot:

- The **blue line** represents the actual C6H6(GT) values.
- The **orange line** shows the predicted values by the LSTM model.
- The two curves closely follow each other, indicating high prediction accuracy.
- Some sharp drops in the true values (outliers or rare events) were not captured perfectly, which is expected.

- Overall, the model shows strong performance with minimal deviation between actual and predicted values across time.

The figure below shows the Training and validation loss plot.



The graph above presents the **training loss** and **validation loss** curves over 50 epochs. Both losses are measured using the Mean Squared Error (MSE), which is suitable for regression tasks like air quality prediction.

- The **blue line** represents the training loss.
- The **orange line** shows the validation loss.
- **Rapid Decrease Initially**: The training loss drops sharply in the first few epochs, indicating that the LSTM model quickly captured essential patterns from the training data.
- **Stability**: After approximately the 10th epoch, the training loss stabilizes and continues to improve gradually

- **Validation Loss Behavior**: The validation loss also decreases early and then stabilizes in the range of 0.004 to 0.005, showing consistent performance on unseen data.
- **No Overfitting**: There is no significant gap between training and validation loss curves. This suggests that the model does not overfit and generalizes well to the validation set.

Conclusion & Future Work

In this project, I built a deep learning model using LSTM to predict air quality, focusing on the concentration of Benzene (C6H6(GT)), based on time-series data. The model gave good results, as the error values like MSE, RMSE, and MAE were low, and the R^2 score was high (0.93). This means the model was able to capture most of the patterns in the data.

When I compared the predicted values with the actual ones, they were very close, especially when the values were stable. Some differences appeared when there were sudden changes, which is expected in time-series predictions. Also, the training and validation loss curves showed that the model didn't overfiting and was stable after a few epochs.

suggestions for improvement

- Hyperparameter tuning: I can try different LSTM structures, like using more layers or bidirectional LSTM, and test different dropout rates or learning rates to improve performance.
- Feature engineering: Adding more input features like weather info, time of day, or interactions between pollutants might help the model learn better.
- Trying other models: I would like to test GRU, Transformer-based models, or even combine CNN with LSTM to see if they perform better.
- Using ensemble methods: Combining several models together could help make predictions more accurate and stable.

Methodology:

2- Echo State Network (ESN)

The Echo State Network (ESN) is a type of Recurrent Neural Network (RNN) based on the reservoir computing paradigm. Unlike traditional RNNs, only the output layer of an ESN is trained; the recurrent layer (reservoir) remains fixed after random initialization.

This significantly reduces training complexity and allows the model to handle time-series data efficiently.

Input Layer: Sequential data (PT08.S1(CO), PT08.S2(NMHC), PT08.S3(NOx), PT08.S4(NO2), PT08.S5(O3))

Reservoir Layer: Sparsely connected recurrent network capturing dynamic temporal dependencies.

Output Layer: Linear regression to produce predictions for the next time step.

Hyperparameter

Hyperparameter	Value
Reservoir size	500 neurons
Spectral radius	0.95
Input scaling	0.5
Ridge alpha	1e-6
Time steps (window size)	24

Reservoir states were computed by iterating over each 24-hour sequence, updating the internal dynamics, and collecting the final reservoir state as the representation for that sample.

Experiment Setup

Tools & Libraries:

- **Language:** Python 3.11
- **Notebook:** Google Colab
- **Libraries:** NumPy, pandas, scikit-learn, matplotlib, seaborn
- **Data Source:** Air Quality UCI Dataset from the UCI Machine Learning Repository

Justification:

- **ESN** was selected for its speed and ability to handle temporal dependencies without backpropagation through time.
- A 24-hour window was used to simulate daily cycles in pollutant behavior.
- Ridge regression provides a stable and fast solution for readout training.

Results & Analysis

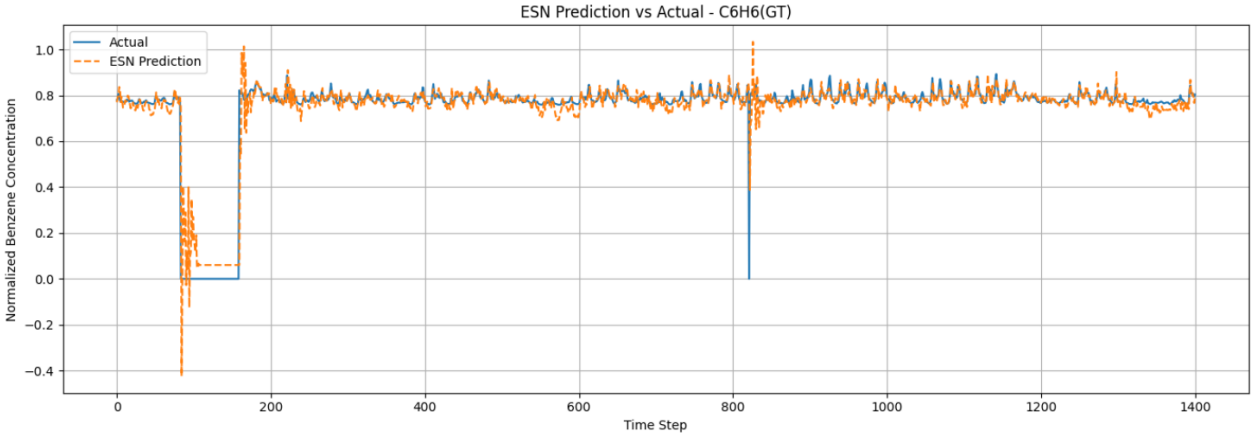
Evaluation Metrics

Metric	Value
MSE	0.00309
RMSE	0.05565
MAE	0.02494
R^2 Score	0.905

The **R^2 score of 0.91** indicates that the model explains about **91% of the variance** in the true benzene concentrations, which reflects strong predictive performance.

Figure Analysis:

The image below shows the predicted vs. actual C6H6(GT).



Interpretation of the Plot:

- **Overall Fit is Good**

The ESN predictions (**orange**) generally follow the actual values (**blue**), showing that the model captures the main trends well.

- **Captures Fluctuations**

The ESN tracks fast changes in benzene levels accurately, showing it learned the short-term dynamics effectively.

- **Some Errors on Sharp Peaks**: which is common in time series

- **No Strong Bias**

Predictions don't consistently overestimate or underestimate, indicating balanced performance.

- **Stable Across Time**

The model performs consistently throughout the entire test set, suggesting it generalizes well.

Conclusion & Future Work

ESN is capable of accurately forecasting benzene concentration in urban air using environmental and sensor features. The **ESN** achieved a high R^2 score (0.91) and low error metrics on the test set, showing its effectiveness and efficiency in handling temporal patterns.

performance could be improved by:

- Tuning hyperparameters such as spectral radius, reservoir size, and leak rate
- Using multiple reservoir states per sequence (e.g., averaging last few steps)
- Incorporating additional relevant features or external data (e.g., wind speed, traffic)
- Comparing with LSTM and Bi-LSTM models to benchmark temporal learning capabilities

Methodology

3- Bi-LSTM

A **Bidirectional Long Short-Term Memory (Bi-LSTM)** model was developed to predict the concentration of benzene gas C6H6(GT) using time series data. The architecture includes:

```
model = Sequential()
model.add(Bidirectional(LSTM(64, return_sequences=False),
input_shape=(X_train.shape[1], X_train.shape[2])))
model.add(Dense(1))
model.compile(optimizer='adam', loss='mse')
```

- **Input Layer**: Sequential data input
- One **Bidirectional LSTM** layer with **64 units**.
- return_sequences=False :

as the task is to predict a single value at the end of the sequence.

- **Dense Output Layer**: A single neuron that outputs the predicted C6H6 value.
- **Loss Function**: Mean Squared Error (MSE) was used, suitable for regression tasks.
- **Optimizer**: Adam optimizer for faster and more stable convergence.

Hyperparameter

Hyperparameter	Value
LSTM Units	64
Optimizer	Adam
Loss Function	MSE
Epochs	50
Batch Size	32
Return Sequences	False

Experiment Setup

Tools & Libraries

- **Language:** Python
- **TensorFlow / Keras** for model creation
- **scikit-learn** for preprocessing and evaluation metrics
- **Matplotlib** for visualizations
- **Google Colab** environment

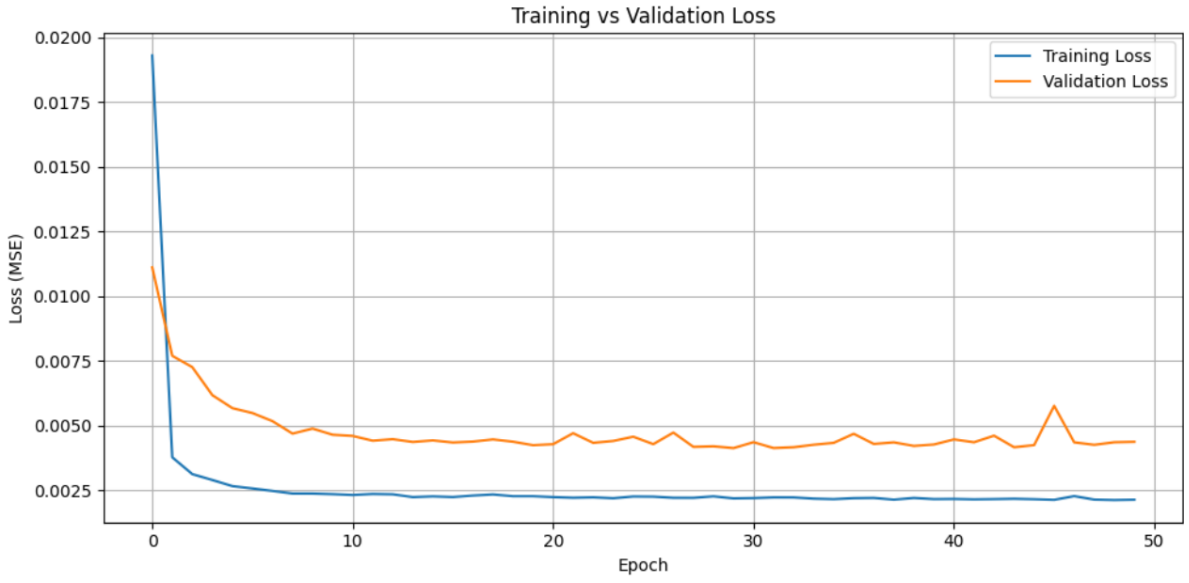
Justification

- Bi-LSTM is it processes the sequence in both forward and backward directions, improving the model's ability to capture temporal dependencies.
- The Adam optimizer was used due to its adaptive learning rate and strong performance in deep learning tasks.
- Normalization helped the model converge faster and avoid gradient instability.
- C6H6 was selected as the target due to its relevance as a toxic air pollutant.

Results & Analysis

Figure Analysis

The image below shows the Training and validation loss plot.



- **Blue Line – Training Loss:**
- Represents how well the model is learning from the training data.
- Starts high (around **0.019 MSE**) at **epoch 0**, indicating large initial error.
- Drops rapidly during the first 10 epochs, showing effective learning.
- After epoch 10, the curve flattens, indicating that the model has **converged** and is no longer improving significantly with further training.

- **Orange Line – Validation Loss:**

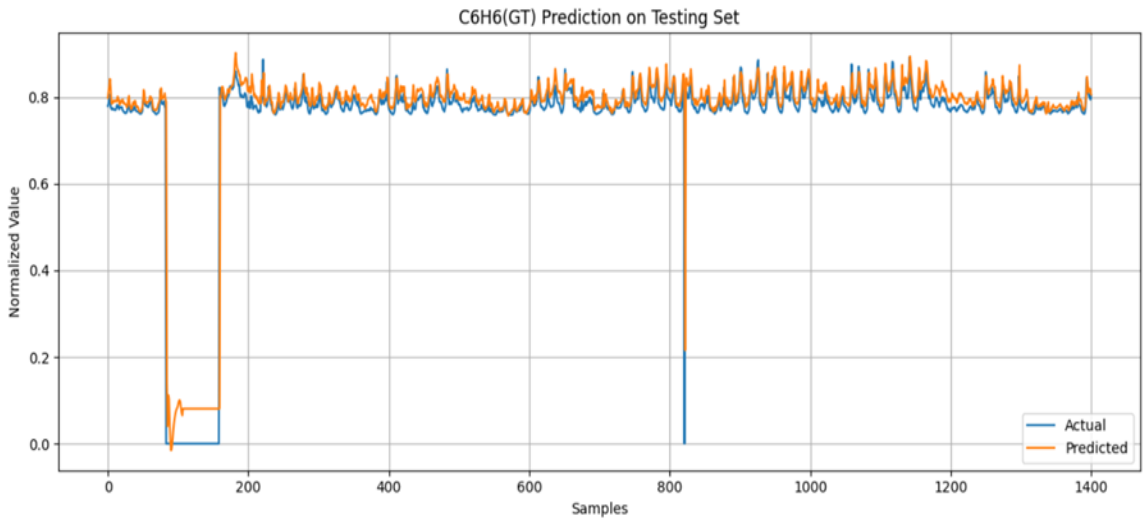
➢ Reflects how well the model performs on **unseen (validation) data**.

➢ Follows a similar pattern to the training loss: initial drop followed by stabilization.

➢ The gap between training and validation loss remains **small and consistent**, suggesting that the model is **not overfitting**.

➢ The validation loss stays low (around **0.004 MSE**), which is a **strong indicator of good generalization**.

The image below shows the predicted vs. actual C6H6(GT) plot.



➢ The **orange line** (**Predicted**) closely follows the **blue line** (**Actual**), indicating that the model **generalizes well.**

➢ Minor prediction errors are noticeable during sharp drops or spikes (e.g., around sample **800**), likely due to rare or extreme values in the data.

➢ In most regions, especially where values are stable, the prediction accuracy is **excellent.**

key Observations

- **No signs of overfitting**: The validation loss does not increase while training loss continues to decrease. Both remain low and close together.
- **Stable training**: The loss curves are smooth and don't show erratic spikes or oscillations, which means training was stable and successful.

Evaluation Metrics

Dataset	**MSE**	**RMSE**	**MAE**	**R^2 Score**
Validation	0.00436	0.06605	0.02239	0.92427
Testing	0.00197	0.04444	0.01723	0.93961

These results demonstrate strong predictive performance, with **R^2 scores** above **0.93**, indicating the model explains over **93%** of the variance in the target variable.

Conclusion & Future Work

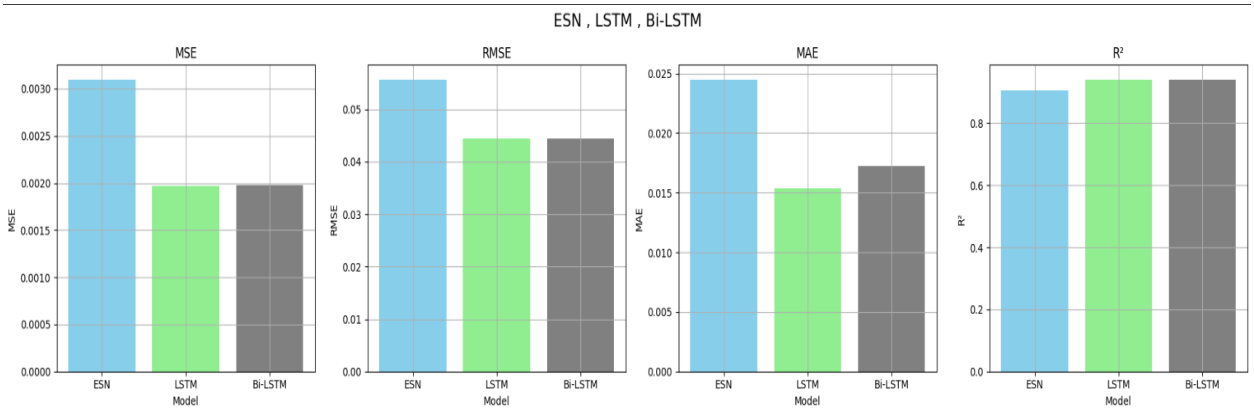
a Bidirectional Long Short-Term Memory (**Bi-LSTM)** model was successfully implemented to predict the concentration of Benzene (**C6H6**) based on time-series sensor data. The model achieved strong performance on both validation and testing sets, with an R^2 score of **0.9262** and **0.9376**, respectively. These results demonstrate the model's ability to effectively capture temporal dependencies in the data and generalize to unseen samples.

The loss curve analysis showed that the model converged quickly and maintained low and stable training and validation losses, indicating well-tuned hyperparameters and a lack of overfitting.

suggestions for improvement

- **Tuning Hyperparameters**: Adjusting the number of LSTM units, batch size, and learning rate to find the optimal configuration.
- **Adding Dropout**: To reduce overfitting and improve generalization.
- **Using More Features**: Including time-related or external features to provide more context.
- **Trying GRU or CNN-LSTM**: Alternative models may better capture temporal patterns.

Comparison



The comparison shows that the **LSTM** model outperforms both **ESN** and **Bi-LSTM**:

- LSTM has the lowest errors (**MSE**, **RMSE**, **MAE**) and the highest **R^2 (0.94)**, meaning it's the most accurate.
- Bi-LSTM comes close, with slightly higher error rates and an **R^2 of 0.94**.
- ESN performs the worst, with the highest errors and the lowest **R^2 (0.91)**.

References

- **Air Quality Dataset**
  UCI Machine Learning Repository. *Air Quality Dataset*.
  Available at: https://archive.ics.uci.edu/ml/datasets/Air+Quality
- Some usage of codes in Kaggle

Libraries Used

- **NumPy** – For numerical operations and array handling.
- **Pandas** – For data loading, cleaning, and exploration.
- **Matplotlib** – For plotting results, including loss curves and predictions vs actual values.
- **Scikit-learn (sklearn)**
  For evaluation metrics such as MSE, RMSE, MAE, and R^2 Score.
  For splitting the dataset into training, testing, and validation sets.
  For data normalization using `MinMaxScaler`.
- **TensorFlow / Keras** – For building, training, and evaluating the models.

2. Classification

Traffic congestion and related problems are a common concern in urban areas. Understanding traffic patterns and analyzing data can provide valuable insights for transportation planning, infrastructure development, and congestion management.

Objectives

- Classify traffic conditions into categories: Heavy, High, Normal, and Low.
- Automate traffic analysis and reduce reliance on manual monitoring.

- Provide real-time insights to assist in traffic management and signal control.
- Help authorities respond quickly to congestion and traffic-related events.
- Identify traffic flow patterns across different times and days.
- Support the development of smart traffic systems in modern urban environments.

About Datasets:

i. **Dataset name:** Traffic and TrafficTwoMonth
ii. **Number of records:** 8928
iii. **Number of columns:** 10

Dataset Features:

- **CarCount**: Number of cars detected.
- **BikeCount**: Number of bikes or motorcycles detected.
- **BusCount**: Number of buses detected.
- **TruckCount**: Number of trucks detected.
- **Total**: Total number of vehicles (sum of all vehicle types).
- **Hour**: Hour extracted from the time column.
- **Day**: Day extracted from the date.
- **Month**: Month extracted from the date.
- **Day of the week**: Day name (e.g., Sunday, Monday).
- **Source**: Indicates if the data is from one-month or two-month source.
- **Weekend**: Indicates if the day falls on a weekend (Friday or Saturday).

Chosen datasets is a valuable resource for studying traffic conditions as it contains information collected by a computer vision model. The model detects four classes of vehicles: cars, bikes, buses, and trucks. The dataset is stored in a CSV file and includes additional columns such as time in hours, date, days of the week, and counts for each vehicle type (CarCount, BikeCount, BusCount, TruckCount). The "Total" column represents the total count of all vehicle types detected within a 15-minute duration.

The dataset is updated every 15 minutes, providing a comprehensive view of traffic patterns over the course of one month. Additionally, the dataset includes a column indicating the traffic situation categorized into four classes: **1-Heavy**, **2-High**, **3-Normal**, and **4-Low**. This information can help assess the severity of congestion and monitor traffic conditions at different times and days of the week.

Data Preprocessing:

1. Data Collection & Loading:

Two datasets were collected; each dataset includes traffic counts detected using computer vision models. Datasets were loaded and combined into a single DataFrame for analysis.

```
# Display settings
pd.set_option('display.max_columns', None)
sns.set(style="whitegrid")
```

```
[4] # Load datasets
traffic_df = pd.read_csv('/content/Traffic.csv')
traffic_two_month_df = pd.read_csv('/content/TrafficTwoMonth.csv')
```

```
[7] traffic_df.head(3)
```

Show hidden output

```
[8] traffic_df['Traffic Situation'].value_counts()
```

Show hidden output

```
[9] # Combine datasets
traffic_df['Source'] = 'OneMonth'
traffic_two_month_df['Source'] = 'TwoMonth'
combined_df = pd.concat([traffic_df, traffic_two_month_df], ignore_index=True)
```

2. Data Cleaning

Checked for and removed duplicate rows, missing values were verified, and the dataset was confirmed to be complete. Outliers in vehicle count columns (CarCount, BikeCount, BusCount, TruckCount) were removed using the IQR method to reduce noise

```
[24] # Remove outliers using IQR method
     def remove_outliers(df, columns):
         for col in columns:
             Q1 = df[col].quantile(0.25)
             Q3 = df[col].quantile(0.75)
             IQR = Q3 - Q1
             lower = Q1 - 1.5 * IQR
             upper = Q3 + 1.5 * IQR
             df = df[(df[col] >= lower) & (df[col] <= upper)]
         return df
     vehicle_cols = ['CarCount', 'BikeCount', 'BusCount', 'TruckCount']
     combined_df = remove_outliers(combined_df, vehicle_cols)
```

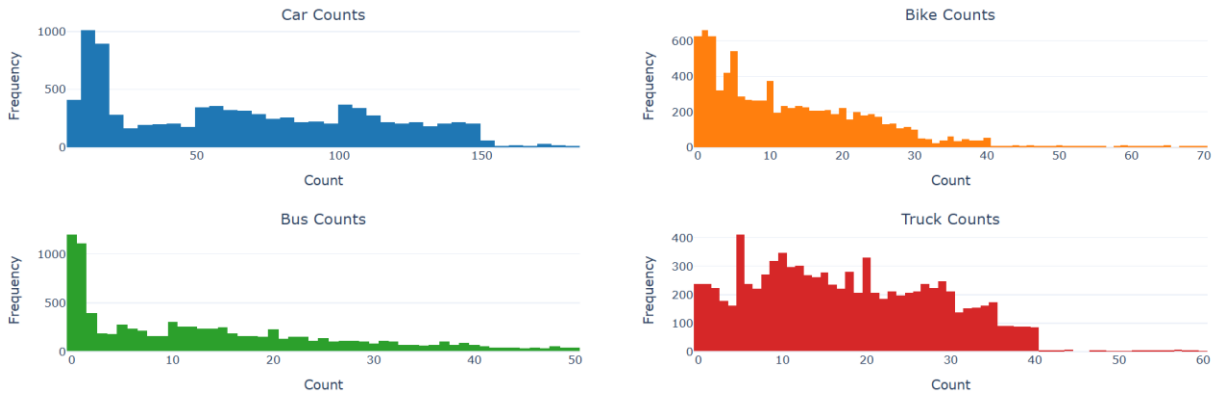
```
 #Check for missing values and duplicates
print("Missing values in each column:")
print(combined_df.isnull().sum())

print(f"Number of duplicate rows: {combined_df.duplicated().sum()}")
```

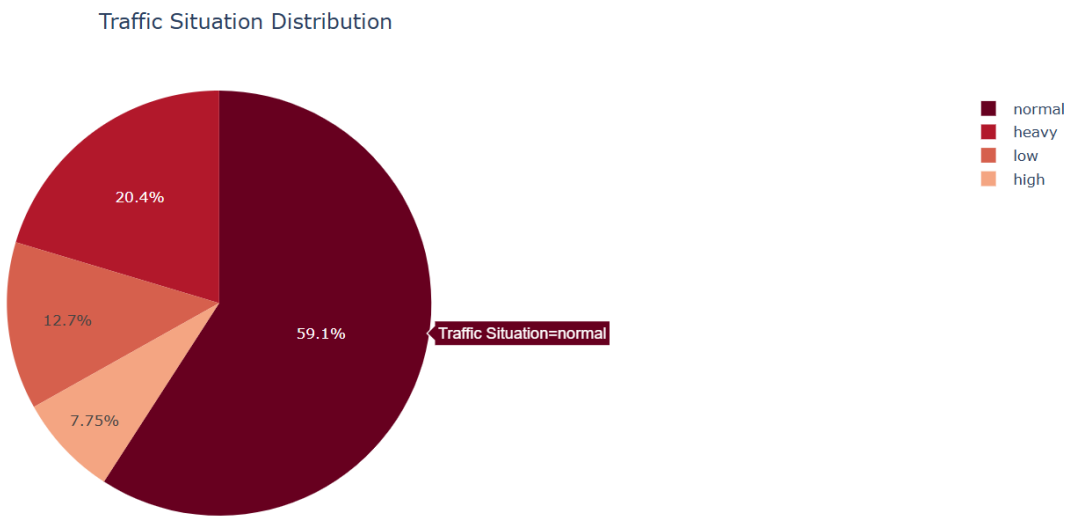
3. Exploratory Data Analysis (EDA)

Histograms plotted for each vehicle type to observe distribution. A pie chart displayed the proportion of each traffic situation class (Heavy, High, Normal, Long). A heatmap was used to visualize correlations between different vehicle types and total counts.

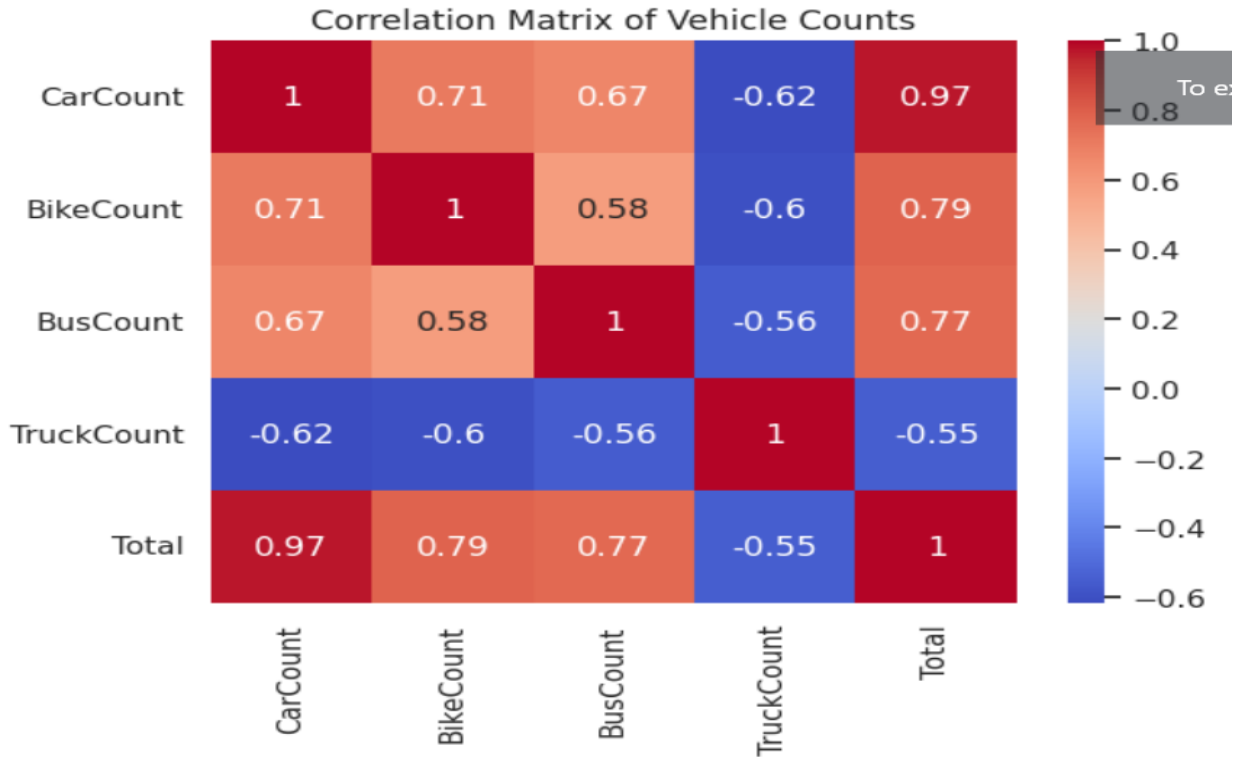
A- Histograms



B- pie char


Traffic Situation Distribution
20.4%
12.7%
59.1%
7.75%
Traffic Situation=normal
normal
heavy
low
high

C- heatmap


Correlation Matrix of Vehicle Counts
CarCount
BikeCount
BusCount
TruckCount
Total
1
0.71
0.67
-0.62
0.97
0.71
1
0.58
-0.6
0.79
0.67
0.58
1
-0.56
0.77
-0.62
-0.6
-0.56
1
-0.55
0.97
0.79
0.77
-0.55
1
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6

4. Feature Engineering

Extracted new time-based features: Hour from Time Day , Month from Date , and created a Weekend feature to identify Friday/Saturday. Combined all numeric and categorical features for further processing.

```
# Feature engineering
combined_df['Hour'] = pd.to_datetime(combined_df['Time'], errors='coerce').dt.hour
combined_df['Day'] = pd.to_datetime(combined_df['Date'], errors='coerce').dt.day
combined_df['Month'] = pd.to_datetime(combined_df['Date'], errors='coerce').dt.month
combined_df['Weekend'] = combined_df['Day of the week'].isin(['Friday', 'Saturday']).astype(int)
```

Show hidden output

```
# Features
numeric_features = ['CarCount', 'BikeCount', 'BusCount', 'TruckCount', 'Total', 'Hour', 'Day', 'Month']
categorical_features = ['Day of the week', 'Source']
```

5. Encoding Categorical Features

One-hot encoding was applied to Day of the week and Source. This transformed categorical data into a machine-readable numeric format without introducing bias.

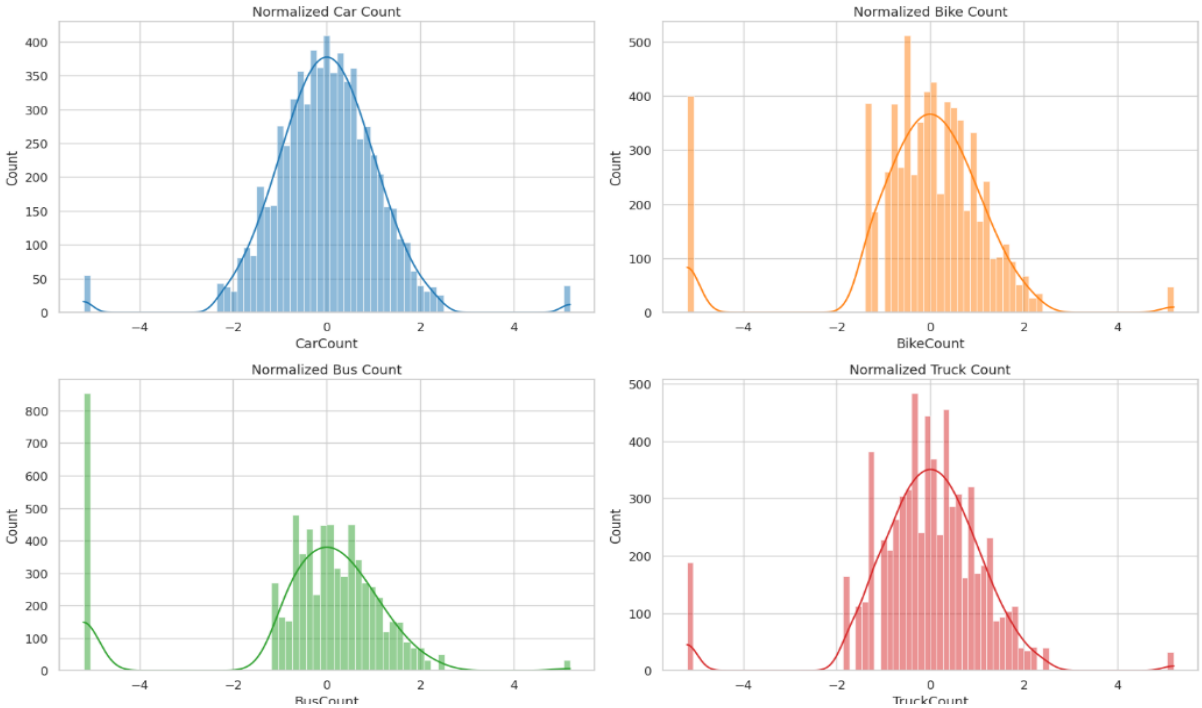
```
# Preprocessor
preprocessor = ColumnTransformer(
    transformers=[
        ('num', StandardScaler(), numeric_features),
        ('cat', OneHotEncoder(handle_unknown='ignore'), categorical_features)
    ])
```

6. Data Normalization

Applied Quantile Transformer to normalize vehicle count features to a normal distribution.

This helped improve model learning by reducing skewness and bringing values to a consistent scale.

```python
# Normalize vehicle count columns
scaler = QuantileTransformer(output_distribution='normal')
combined_df[vehicle_cols] = scaler.fit_transform(combined_df[vehicle_cols])
```



7. Data Splitting

The dataset was split into: (80% for training,20% for testing), this approach ensures the model trains on most of the data while being evaluated on unseen samples to measure real-world performance.

```python
2] # Split the data into training and testing sets
   X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

Methodology

2) Classification:

1-SVM

Support Vector Machine (SVM) was utilized for multi-class classification to predict four categories: Heavy, High, Normal, and Low.

```python
# 0 = Heavy, 1 = High, 2 = Normal, 3 = Low
```

```
class_names = ['Heavy', 'High', 'Normal', 'Low']
kernels = ['linear', 'poly', 'rbf']
for kernel_type in kernels:
    print(f" Kernel: {kernel_type.upper()} ")
    svm_clf = SVC(kernel=kernel_type, random_state=42)
    svm_clf.fit(X_train_scaled, y_train)
    y_pred = svm_clf.predict(X_test_scaled)
```

Kernels Explored:

- **Linear Kernel**: Suitable when the data is linearly separable.
- **Polynomial Kernel**: Captures interactions of features up to a certain degree.
- **RBF (Radial Basis Function) Kernel**: Handles non-linear relationships effectively.

```
# Hyperparameter Tuning
print("\n Hyperparameter Tuning for RBF Kernel ")
param_grid = {
    'C': [0.1, 1, 10],
    'gamma': [0.001, 0.01, 0.1],
    'kernel': ['rbf']
}

svm = SVC(random_state=42)
grid_search = GridSearchCV(svm, param_grid, cv=5, n_jobs=-1, verbose=1)
grid_search.fit(X_train_scaled, y_train)

print("Best Parameters:", grid_search.best_params_)
print("Best CV Score:", grid_search.best_score_)
```

Hyperparameter Tuning (for RBF):

- C: [0.1, 1, 10] — controls regularization strength.
- gamma: [0.001, 0.01, 0.1] — defines influence of a single training example.
- kernel: fixed as 'rbf' for tuning.

Hyperparameters were tuned using **GridSearchCV** with 5-fold cross-validation to identify the best parameter combination.

The Result:

Hyperparameter Tuning for RBF Kernel
Fitting 5 folds for each of 9 candidates, totalling 45 fits
Best Parameters: {'C': 10, 'gamma': 0.1, 'kernel': 'rbf'}

Best CV Score: 0.9286135502356965

Experiment Setup

Environment & Tools:

- **Language**: Python
- **Google Colab** environment
- **Libraries**: scikit-learn, pandas, numpy, matplotlib, seaborn, plotly
- **Data Splitting**: 80% for training, 20% for testing
- **Feature Scaling**: StandardScaler was used to normalize the input features and improved performance with SVM
- **Label Binarization**: Applied for AUC-ROC score calculation using label_binarize

Justification of Approach:

- SVM is effective for classification tasks, especially when combined with kernels.
- Multiple kernel functions were tested to determine which fits the data best.
- Grid Search ensures systematic hyperparameter exploration

Results & Analysis

Classification Report:

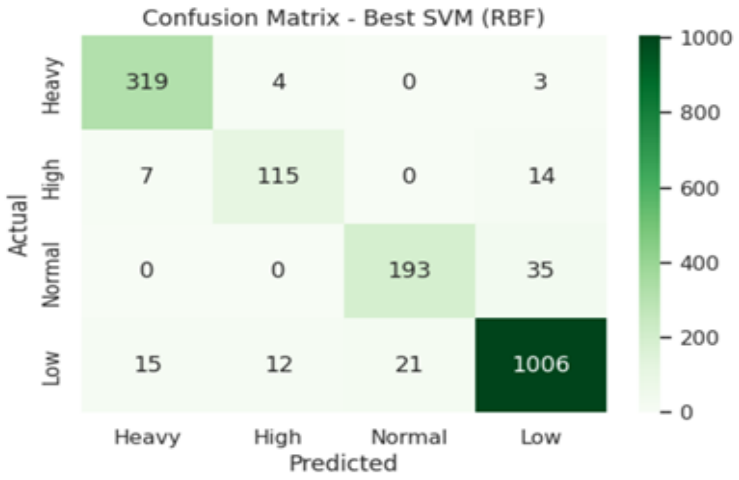
Class	Precision	Recall	F1-score	Support
Heavy	0.94	0.98	0,96	326
High	0.88	0.85	0.86	136
Low	0.95	0.95	0.95	1054
Normal	0.90	0.85	0.87	228

Evaluation Metrics

Metric	Accuracy	Precision	Recall	F1-Score	AUC-ROC
value	0.9364	0.9359	0.9364	0.9359	0.9666

Confusion Matrix

The figure below shows the confusion matrix for the best SVM model using an RBF kernel, we displayed in the heatmap generated using seaborn



Observations:

- Excellent performance overall, especially for the Heavy and Low classes.
- RBF kernel outperformed linear and polynomial kernels.

Conclusion & Future Work

The Support Vector Machine (SVM) model using the Radial Basis Function (RBF) kernel achieved the highest accuracy and overall classification performance compared to other kernels. Additionally, proper data preprocessing particularly feature scaling using **StandardScaler** played a crucial role in ensuring the model's convergence and optimal performance. The application of hyperparameter tuning through **Grid Search** further enhanced the results by systematically identifying the best combination of model parameters, leading to improved generalization on the test set.

suggestions for improvement

- Evaluate additional models like Random Forest, or deep learning approaches.
- Perform feature selection or dimensionality reduction (e.g., PCA) to enhance speed and reduce complexity.

Methodology

2.**Multilayer perceptron (MLP)**

This model is a feedforward neural network trained using backpropagation. It is suitable for multi-class classification problems

Tuned Hyperparameters using GridSearchCV:

- `hidden_layer_sizes`: Number of neurons in the hidden layers ([(50,), (100,), (50, 50)])
- `activation`: Activation functions to introduce non-linearity (["relu", "tanh"])
- `solver`: Optimizer used for weight optimization (["adam"])
- `alpha`: L2 regularization term ([0.0001, 0.001])
- `learning_rate`: Learning rate schedule (["constant", "adaptive"])

```
Best parameters found:
{'classifier__activation': 'tanh', 'classifier__alpha': 0.001, 'classifier__hidden_la
yer_sizes': (50, 50), 'classifier__learning_rate': 'constant', 'classifier__solver':
'adam'}
```

Experiment Setup

- Environment: Python using `scikit-learn library`
- Tools Used:
  `Pipeline,GridSearchCV,classification_report,confusion_matrix,accuracy_score`

Preprocessing

- Numerical features scaled using `StandardScaler`
- Categorical features encoded using `OneHotEncoder`
- Combined using `ColumnTransformer`

Modeling

- MLP Classifier within a pipeline

Hyperparameter Tuning

- ridSearchCV with 3-fold cross-validation

Justification of Approach:

- MLP can learn complex patterns in data, Integrated pipeline ensures reproducibility and cleaner code and GridSearchCV explores parameter combinations to optimize performance

Results & Analysis

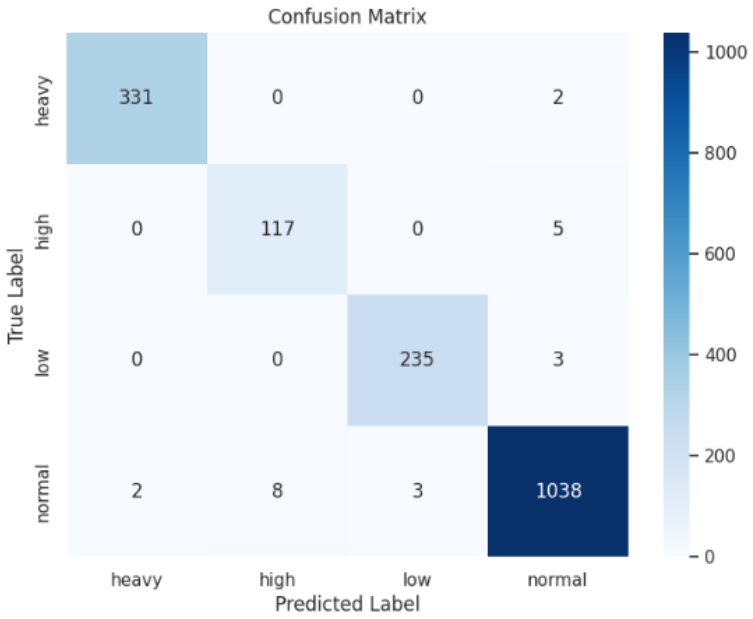
Classification Report:

Class	Precision	Recall	F1-score	Support
0	0.99	0.99	0.99	333
1	0.94	.0.96	0.95	122
2	0.99	0.99	0.99	238
3	0.99	0.99	0.99	1051

Evaluation Metrics

Metric	Accuracy	Precision	Recall	F1-Score	AUC-ROC (multi-class)
value	0.9868	0.9869	0.9868	0.9868	0.9992

Confusion Matrix



Conclusion & Future Work

Conclusion:

- The MLP Classifier achieved excellent performance across all classes
- The best parameter configuration was selected through exhaustive search
- Pipeline approach ensured modularity and reusability

Future Work Suggestions:

- Try other neural network architectures (e.g., deeper networks or CNNs)
- Benchmark against other classifiers such as Random Forest, SVM, XGBoost
- Use `RandomizedSearchCV` for faster hyperparameter search
- Apply SHAP or LIME for explainability
- Validate on external data or unseen test set for generalization

Methodology

3- Random Forest Classifier

This ensemble method builds multiple decision trees and merges them to produce more accurate and stable predictions.

Tuned Hyperparameters using GridSearchCV:

- **`n_estimators`**: Number of trees in the forest [100, 200]
- **`max_depth`**: Maximum depth of each tree [None, 10, 20]
- **`min_samples_split`**: Minimum number of samples required to split an internal node [2, 5]
- **`min_samples_leaf`**: Minimum number of samples required to be at a leaf node [1, 2]
- **`bootstrap`**: Whether bootstrap samples are used when building trees [True, False]

```
Best Parameters:

{'classifier__bootstrap': False, 'classifier__max_depth': None,
'classifier__min_samples_leaf': 1,
'classifier__min_samples_split': 2, 'classifier__n_estimators':
100}
```

Experiment Setup

- **Environment:** Python using the Scikit-learn library.
- **Tools Used:** `Pipeline`, `GridSearchCV`, `classification_report`, `confusion_matrix`, `roc_auc_score`.
- **Training Setup:** A pipeline was used to connect preprocessing and modeling. GridSearchCV was applied with 5-fold cross-validation.
- **Evaluation Strategy:** The dataset was split into training and testing sets, and various evaluation metrics were used to assess performance.

Justification of Approach:

- Handles non-linear relationships effectively.
- Resistant to overfitting due to ensemble nature.
- Performs well even with unbalanced data.

Results & Analysis

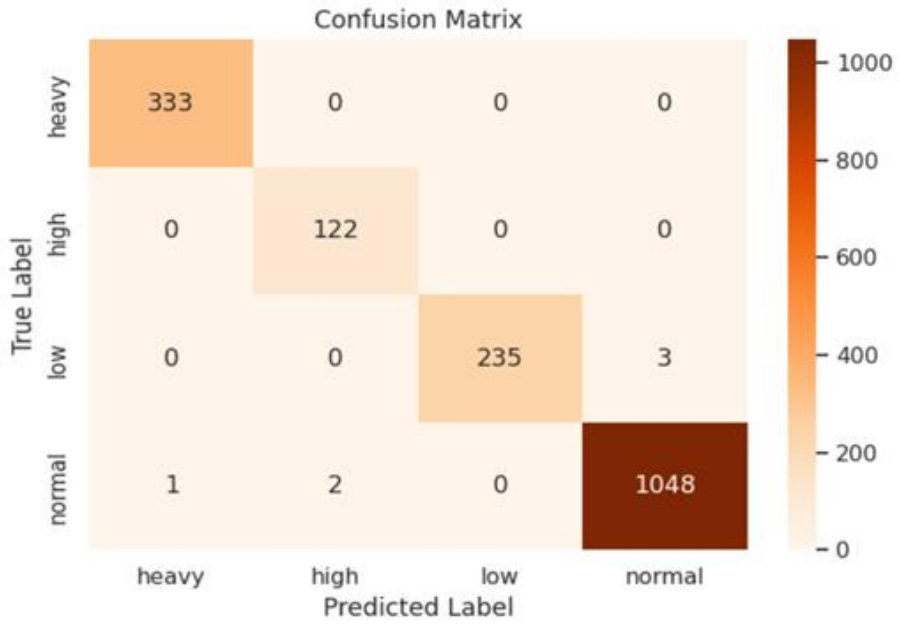
Classification Report:

Class	Precision	Recall	F1-score	Support
Heavy	1.00	1.00	1.00	333
High	0.98	1.00	0.99	122
Low	1.00	0.99	0.99	238
Normal	1.00	1.00	1.00	1051

Evaluation Metrics

Metric	Accuracy	Precision	Recall	F1-Score	AUC-ROC (multi-class)
value	0.9966	0.9966	0.9966	0.9966	0.9997

Confusion Matrix



- The model performed exceptionally well with only a few minor misclassifications:
  - 3 "low" samples predicted as "normal"
  - 1 "normal" predicted as "heavy"
  - 2 "normal" predicted as "high"
- This indicates the model is learning the class distribution effectively, and misclassifications are minimal.
- Excellent performance overall, especially for the Heavy and Normal classes.

Conclusion & Future Work

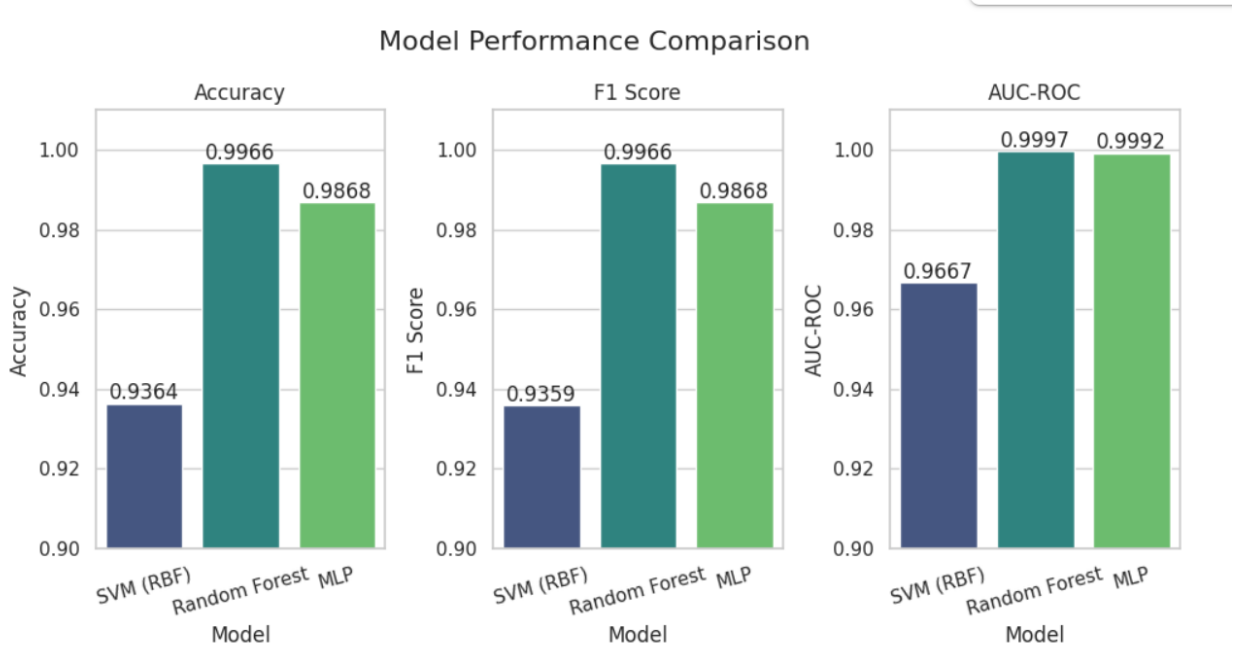
Conclusion:

- The Random Forest model showed outstanding performance on the classification task.
- The high precision, recall, and F1-score metrics across all classes demonstrate strong predictive capability.
- The optimal hyperparameters were selected using exhaustive Grid Search with cross-validation.

Future Work Suggestions:

- Try other models like XGBoost or LightGBM for benchmarking.
- Use RandomizedSearchCV to reduce tuning time.
- Apply model explainability tools like SHAP to understand decision-making.
- Validate on a separate unseen dataset for generalization testing.

Comparison



The chart compares the performance of three machine learning models—SVM (RBF), Random Forest, and MLP—on a classification task using three evaluation metrics: Accuracy, F1 Score, and AUC-ROC. Among the models, Random Forest consistently outperformed the others, achieving the highest scores across all metrics, with an accuracy and F1 score of 0.9966 and an AUC-ROC of 0.9997, indicating excellent classification and generalization ability. The MLP model also performed very well, with an accuracy and F1 score of 0.9868 and an AUC-ROC of 0.9992, slightly below Random Forest. In contrast, the SVM (RBF) model showed comparatively lower performance, with an accuracy of 0.9364, F1 score of 0.9359, and AUC-ROC of 0.9667. Overall, Random Forest demonstrated the most robust and reliable results for the task.

References

- **Traffic Dataset**
  Traffic Prediction Dataset. Kaggle.
  Available at: https://www.kaggle.com/datasets/hasibullahaman/traffic-prediction-dataset

Libraries Used

- **NumPy** – For numerical operations and array handling.
- **Pandas** – For data loading, cleaning, and exploration.
- **Matplotlib** – For plotting results, including loss curves and predictions vs actual values.
- **Scikit-learn (sklearn)**
  For evaluation metrics such as Accuracy, F1 Score, Precision, Recall, and AUC-ROC.
  For splitting the dataset into training, and testing sets.
  For data normalization using `MinMaxScaler`.
- **TensorFlow / Keras** – For building, training, and evaluating the models.