

TRAFFIC MANAGEMENT USING GNN AND MAB WITH SDN ORCHESTRATION

811321104006-ATCHAYA SHRI G

Phase 4 Submission Document

Project Title: Traffic Management

Phase 4: Development Part 2

Topic: Traffic prediction model by loading and pre-processing the dataset.



TRAFFIC MANAGEMENT

Introduction:

- ❖ Traffic management is a critical task in software-defined IoT networks (SDN-IoTs) to efficiently manage network resources and ensure Quality of Service (QoS) for end-users. However, traditional traffic management approaches based on queuing theory or static policies may not be effective due to the dynamic and unpredictable nature of network traffic. In this paper, we propose a novel approach that leverages Graph Neural Networks (GNNs) and multi-arm bandit algorithms to dynamically optimize traffic management policies based on real-time network traffic patterns.
- ❖ Specifically, our approach uses a GNN model to learn and predict network traffic patterns and a multi-arm bandit algorithm to optimize traffic management policies based on these predictions. We evaluate the proposed approach on three different datasets, including a simulated corporate network (KDD Cup 1999), a collection of network traffic traces (CAIDA), and a simulated network environment with both normal and malicious traffic (NSL-KDD).
- ❖ The results demonstrate that our approach outperforms other state-of-the-art traffic management methods, achieving higher throughput, lower packet loss, and lower delay, while effectively detecting anomalous traffic patterns. The proposed approach offers a promising solution to traffic management in SDNs, enabling efficient resource management and QoS assurance.

Keywords:

Traffic management;anomaly detection;Intrusion detection;Network security;Internet of things;Network traffic analysis;Machine learning;SDN(software-defined networking);GNN(graph neural network);MAB(multi-armed bandit)

Proposed model:

Traffic management is a critical step in optimizing network performance by controlling the rate of data transmission. It can help to manage congestion and reduce packet loss, improving the overall quality of service (QoS) for end-users. In this section, we will discuss the traffic management component of the proposed model.

Traffic management involves regulating the flow of data through the network by introducing delays and buffering packets. This is typically achieved using a token bucket algorithm, where tokens are generated at a fixed rate and consumed by packets as they are transmitted. If the bucket is empty, packets are queued until sufficient tokens are available.

The token bucket algorithm can be represented mathematically as follows:
At time t , the number of tokens in the bucket can be calculated as:

$$B(t)=\min(B(t-1)+r(t-t_{last}),B_{max})$$

where $B(t-1)$ is the number of tokens at the previous time step, r is the token generation rate, t is the current time, t_{last} is the time when the last token was generated, and B_{max} is the maximum bucket size.

When a packet of size P arrives at time t , it is immediately transmitted if there are enough tokens available in the bucket:

$$\text{if } B(t) \geq P, \text{ then transmit the packet}$$

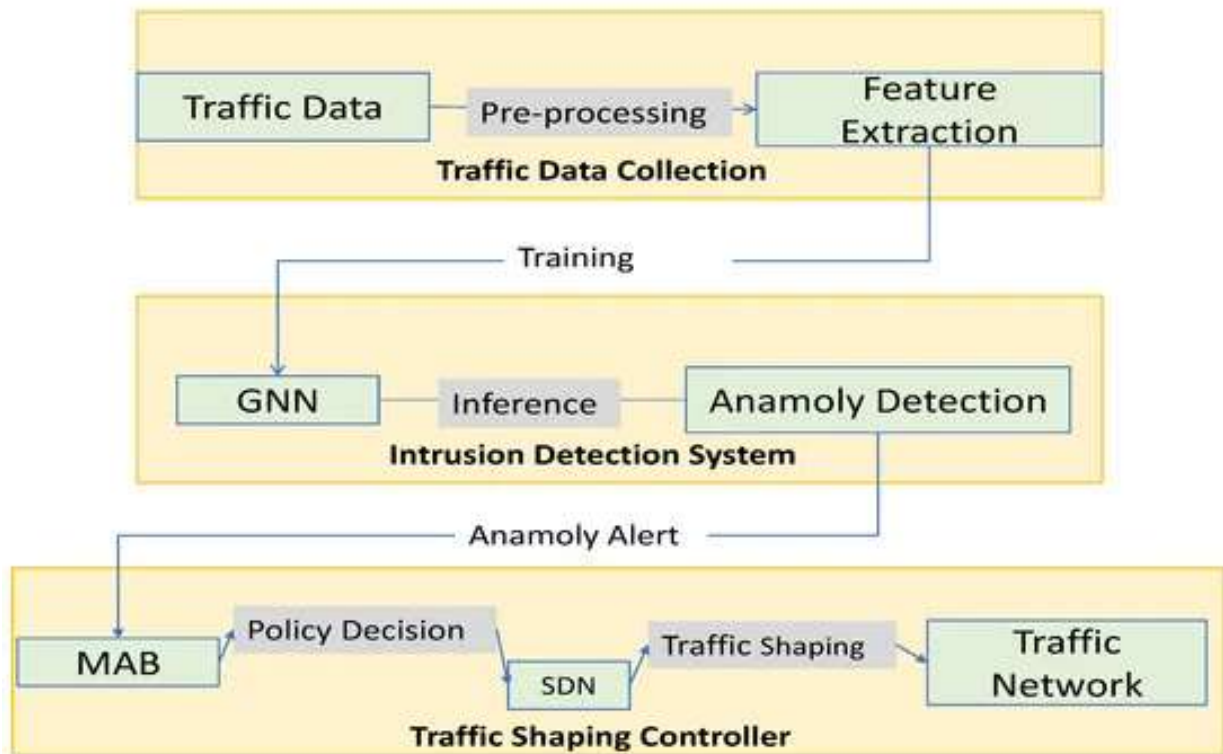
Otherwise, the packet is queued until sufficient tokens become available:

$\text{if } B(t) < P$, then add the packet to the queue if, then add the packet to the queue. Packets in the queue are transmitted in order of arrival as soon as sufficient tokens become available.

The token bucket algorithm can be further optimized by adjusting the token generation rate based on network conditions. For example, if congestion is detected, the token generation rate can be reduced to prevent further congestion.

Data flow:

The Data flow of the proposed starts with the input data, which is passed through a feature extraction module to extract relevant features that will be used for traffic management. The extracted features are then passed to the GNN module, which is responsible for learning the complex patterns in the traffic data and predicting the optimal traffic-shaping policy.



GNN for Understanding Traffic Pattern:

Graph Neural Networks (GNNs) are a type of deep learning method that have recently gained popularity in the field of traffic analysis and prediction. GNNs are particularly effective in modeling and analyzing data that can be represented as graphs, which makes them well-suited for analyzing traffic patterns.

The basic idea behind GNNs is to learn a set of node and edge embeddings that capture the underlying structure of the graph. These embeddings can then be used to perform various downstream tasks, such as node classification, edge prediction, or graph clustering.

In the context of traffic analysis, GNNs can be used to model traffic flow data as a graph, where each node represents a road segment or intersection and each edge represents the flow of traffic between them. The GNN can then learn a set of

embeddings that capture the underlying patterns of traffic flow, such as congestion, bottlenecking, and routing preferences.

The mathematical equations used in GNNs are typically based on message-passing algorithms, which allow nodes in the graph to communicate and update their embeddings based on the embeddings of their neighbours. One commonly used message-passing algorithm is the Graph Convolutional Network (GCN), which is based on the following equation:

$$\{h^{(l+1)}_i = \sigma(\sum_{j \in \mathcal{N}(i)} \frac{1}{c_{ij}} W^{(l)} h^{(l)}_j)$$

Necessary Steps To Follow:

1.Import Libraries:

Start by importing the necessary libraries.

Program:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import datetime
import tensorflow
from statsmodels.tsa.stattools import adfuller
from sklearn.preprocessing import MinMaxScaler
from tensorflow import keras
from keras import callbacks
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Conv2D, Flatten, Dense, LSTM, Dropout, GRU, Bidirectional
from tensorflow.keras.optimizers import SGD
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import GRU, Dropout, Dense
from tensorflow.keras.optimizers import SGD
from tensorflow.keras import callbacks
import math
from sklearn.metrics import mean_squared_error

import warnings
warnings.filterwarnings("ignore")
```

2.load the dataset:

Load your dataset into a pandas Dataframe. You can typically find traffic prediction datasets in csv format, but you can adapt this code to other format as needed.

Program:

```
data = pd.read_csv("../input/traffic-prediction-dataset/traffic.csv")  
data.head()
```

3.Data Exploration:

Perform data exploration to understand your data better. This includes checking for missing values, exploring the data's statistics, and visualizing it to identify patterns.

Program:

```
data["DateTime"] = pd.to_datetime(data["DateTime"]) data  
= data.drop(["ID"], axis=1) #dropping IDs data.info()
```

4.Feature Engineering:

Depending on your dataset, you may need to create new features or transform existing ones. This can involve one-hot encoding categorical variables, handling date/time data, or scaling numerical features.

Program:

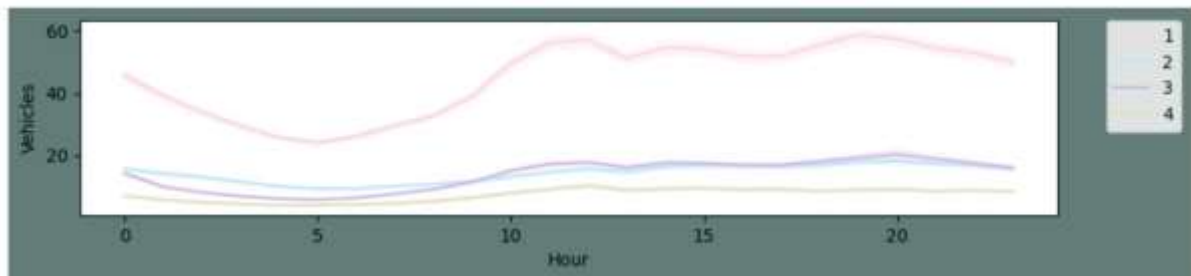
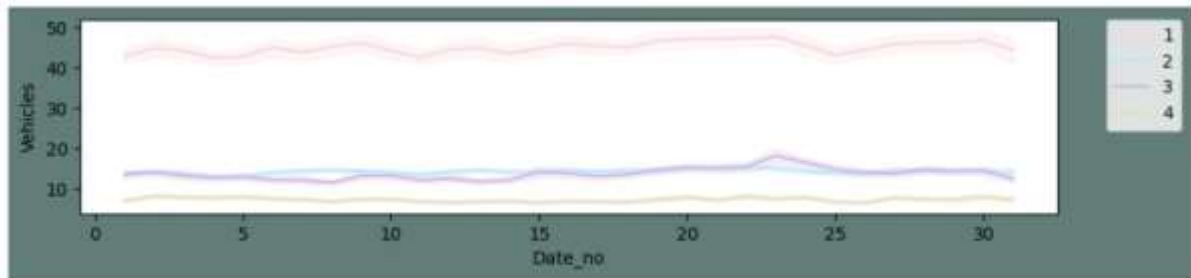
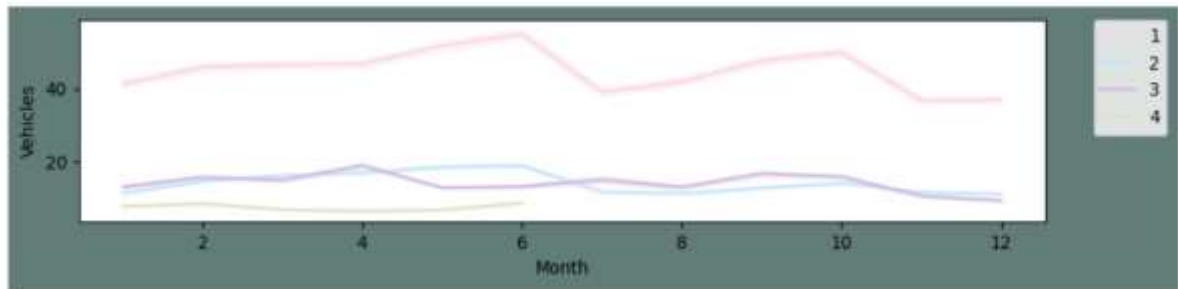
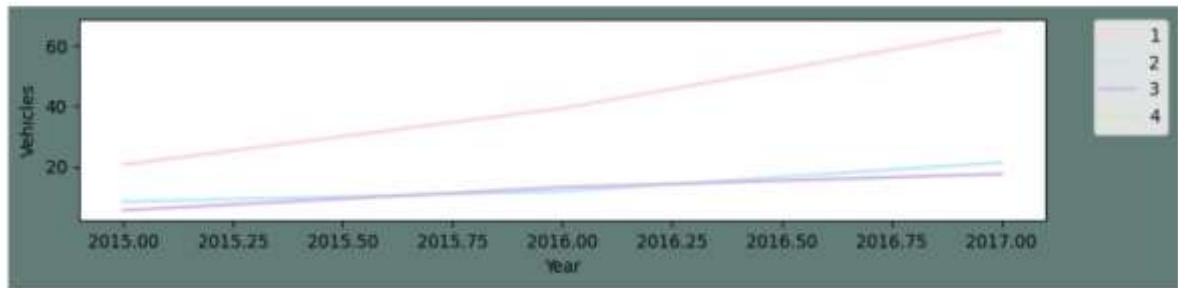
```
df["Year"] = df["DateTime"].dt.year  
df["Month"] = df["DateTime"].dt.month  
df["Date_no"] = df["DateTime"].dt.day  
df["Hour"] = df["DateTime"].dt.hour df["Day"] =  
df.DateTime.dt.strftime("%A") df.head()
```

5.Exploratory Data Analysis(EDA):

Perform data exploration to understand your data better. This includes checking for missing values, exploring the data's statistics, and visualizing it to identify patterns.

Program:

```
new_features = [ "Year", "Month", "Date_no", "Hour", "Day"]
for i in new_features:
plt.figure(figsize=(10,2),facecolor="#627D78")
    ax=sns.lineplot(x=df[i],y="Vehicles",data=df, hue="Junction", palette=colors )
plt.legend(bbox_to_anchor=(1.05, 1), loc=2, borderaxespad=0.)
```

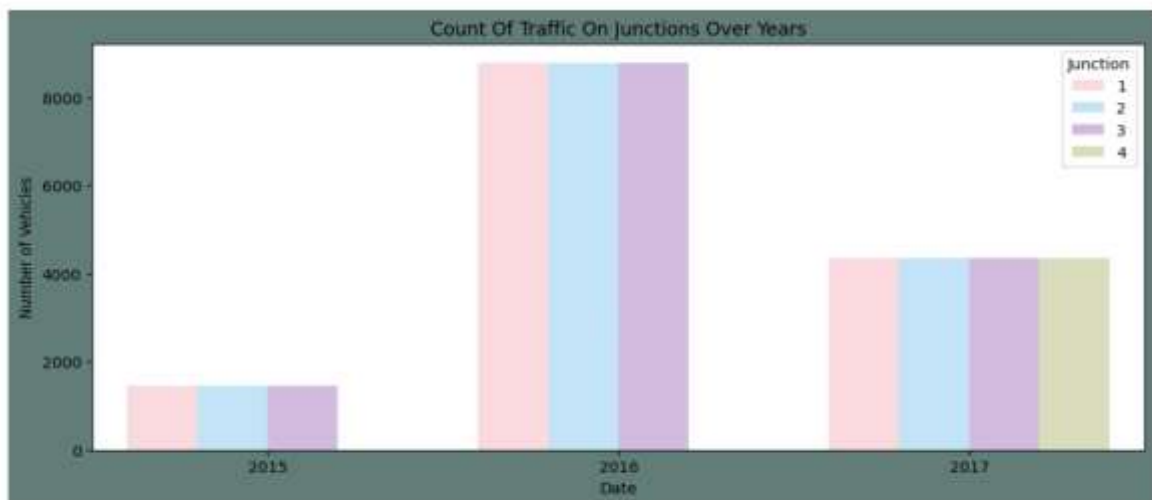


From The Above Plot Following Things Can Be Concluded:

- Yearly, there has been an upward trend for all junctions except for the fourth junction. As we already established above that the fourth junction has limited data and that don't span over a year.
- We can see that there is an influx in the first and second junctions around June. I presume this may be due to summer break and activities around the same.
- Monthly, throughout all the dates there is a good consistency in data.
- For a day, we can see that are peaks during morning and evening times and a decline during night hours. This is as per expectation.
- For weekly patterns, Sundays enjoy smoother traffic as there are lesser vehicles on roads. Whereas Monday to Friday the traffic is steady.

Program:

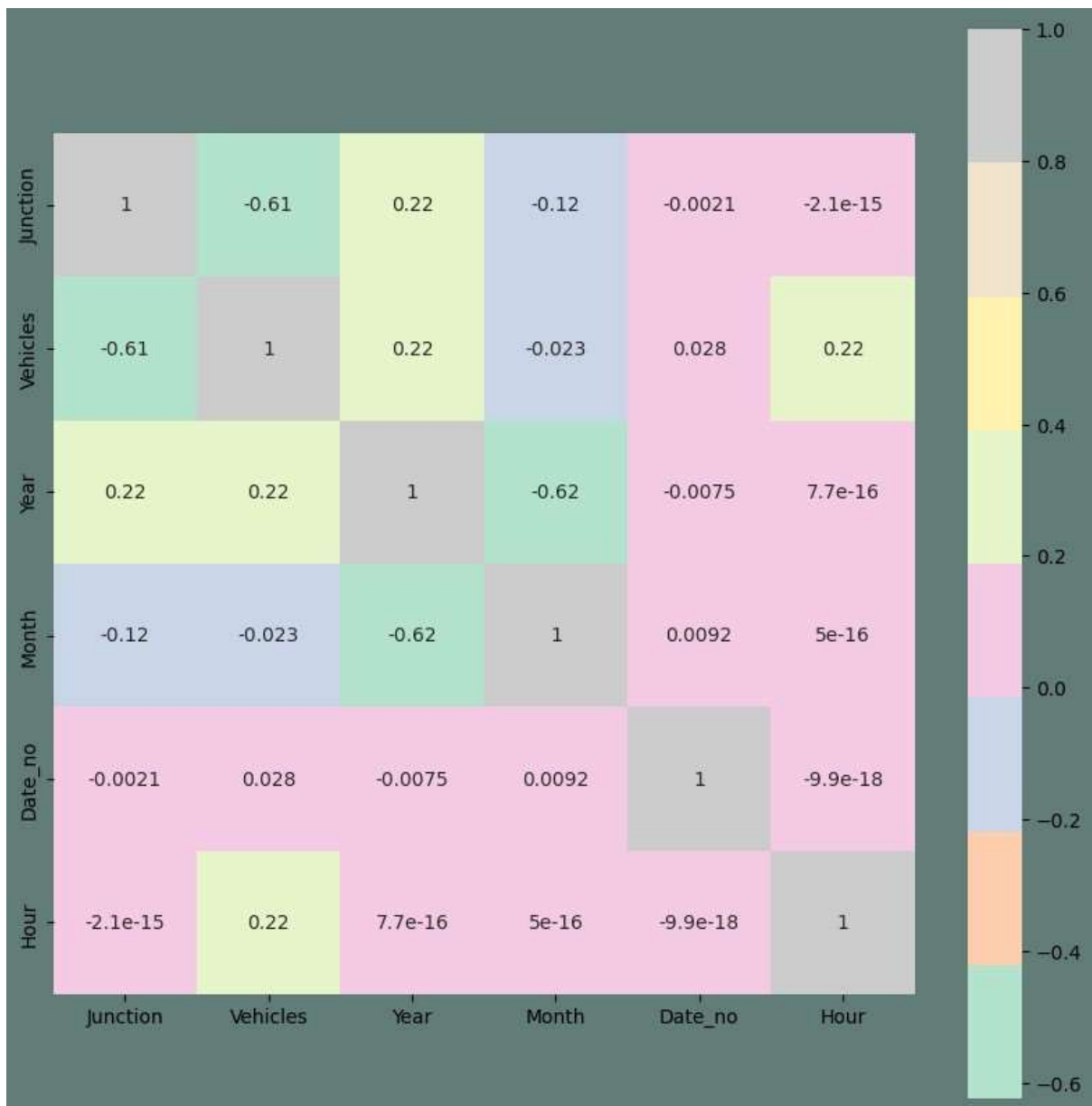
```
plt.figure(figsize=(12,5),facecolor="#627D78")
count = sns.countplot(data=df, x =df["Year"], hue="Junction", palette=colors)
count.set_title("Count Of Traffic On Junctions Over Years")
count.set_ylabel("Number of Vehicles") count.set_xlabel("Date")
```



```
numeric_df = df.select_dtypes(include=[np.number]) # Select only numeric columns
corrmat = numeric_df.corr()

plt.subplots(figsize=(10,10),facecolor="#627D78")
sns.heatmap(corrmat,cmap= "Pastel2",annot=True,square=True, )
```

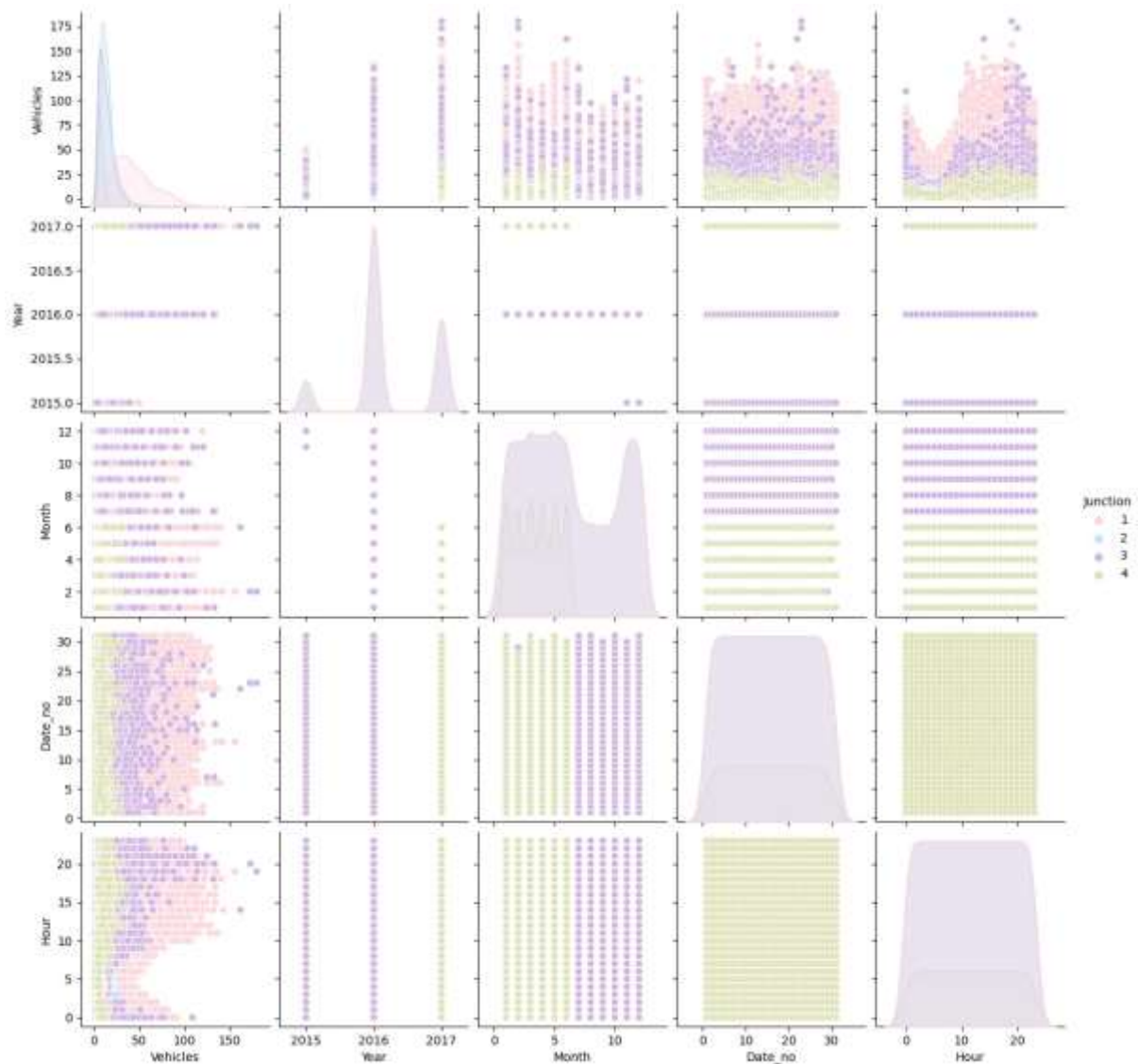
<Axes:>



The highest correlation is certainly with the preexisting feature. I will conclude my EDA with a pair plot. It's an interesting overall representation of any data

```
sns.pairplot(data=df, hue= "Junction",palette=colors)
```

```
<seaborn.axisgrid.PairGrid at 0x7adb85952830>
```



Data Transformation And Preprocessing:

- ❖ Creating different frames for each Junction and plotting them
- ❖ Transforming the series and plotting them
- ❖ Performing the Augmented Dickey-Fuller test to check the seasonality of transformed series
- ❖ Creating test and train sets

Program:

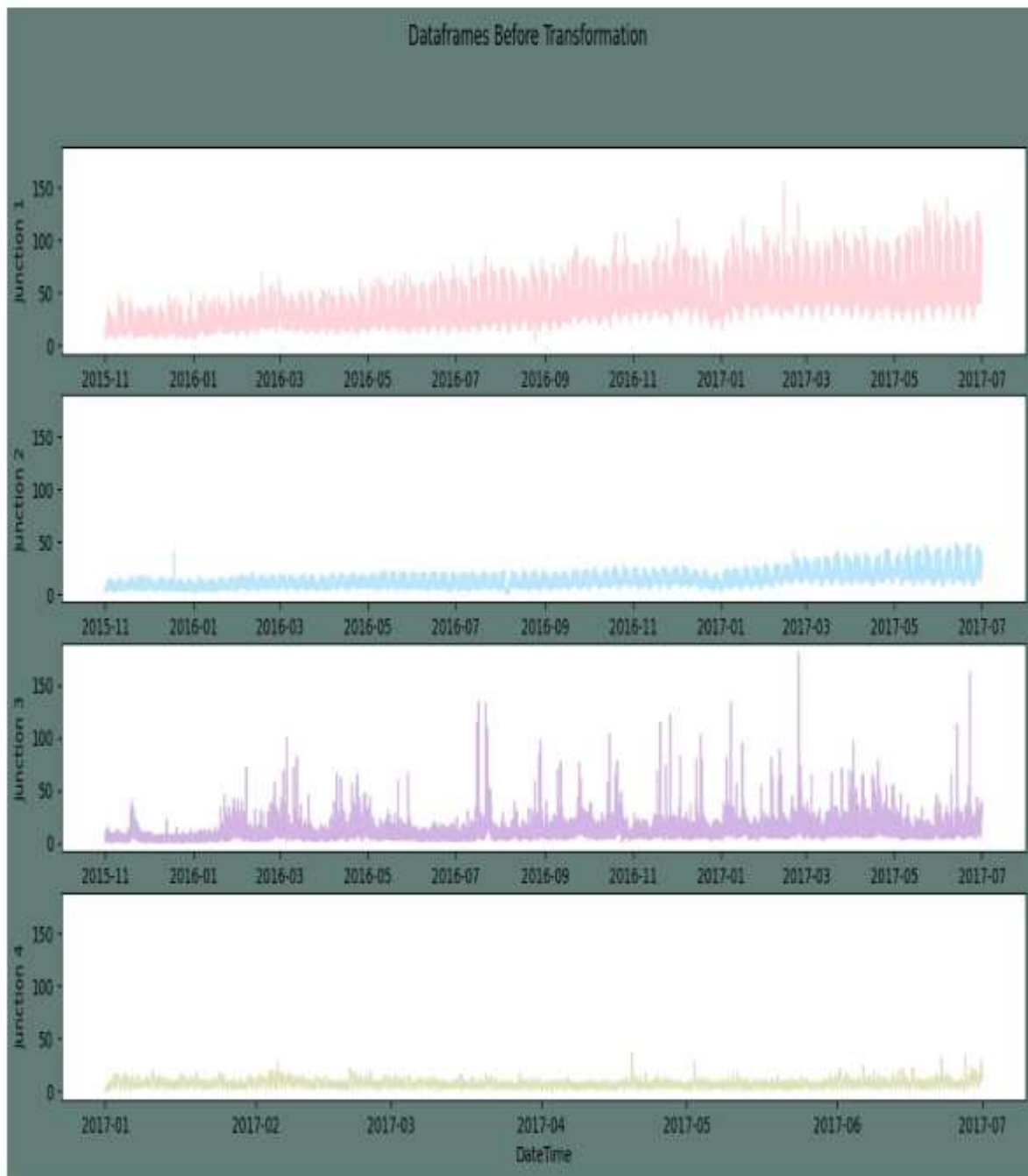
```
#Pivoting data from junction
df_J = data.pivot(columns="Junction", index="DateTime")
df_J.describe()
```

	Vehicles			
Junction	1	2	3	4
count	14592.000000	14592.000000	14592.000000	4344.000000
mean	45.052906	14.253221	13.694010	7.251611
std	23.008345	7.401307	10.436005	3.521455
min	5.000000	1.000000	1.000000	1.000000
25%	27.000000	9.000000	7.000000	5.000000
50%	40.000000	13.000000	11.000000	7.000000
75%	59.000000	17.000000	18.000000	9.000000
max	156.000000	48.000000	180.000000	36.000000

Program:

```
#Creating new sets
df_1 = df_J[['Vehicles', 1]] df_2
= df_J[['Vehicles', 2]] df_3 =
df_J[['Vehicles', 3]] df_4 =
df_J[['Vehicles', 4]]
df_4 = df_4.dropna() #Junction 4 has limited data only for a few months
#Dropping level one in dfs's index as it is a multi index data
frame list_dfs = [df_1, df_2, df_3, df_4] for i in list_dfs:
    i.columns= i.columns.droplevel(level=1)

#Function to plot comparative plots of dataframes def Sub_Plots4(df_1,
df_2,df_3,df_4,title):    fig, axes = plt.subplots(4, 1, figsize=(15,
8),facecolor="#627D78", sharey
=True)
    fig.suptitle(title)
    #J1
    pl_1=sns.lineplot(ax=axes[0],data=df_1,color=colors[0])
    #pl_1=plt.ylabel()
    axes[0].set(ylabel ="Junction 1")
    #J2
    pl_2=sns.lineplot(ax=axes[1],data=df_2,color=colors[1])
axes[1].set(ylabel ="Junction 2")
    #J3
    pl_3=sns.lineplot(ax=axes[2],data=df_3,color=colors[2])
axes[2].set(ylabel ="Junction 3")
    #J4
    pl_4=sns.lineplot(ax=axes[3],data=df_4,color=colors[3])
axes[3].set(ylabel ="Junction 4")
```



Steps For Transforming:

- ❖ Normalizing
- ❖ Differencing

Program:

```

# Normalize Function
def Normalize(df,col):
    average = df[col].mean()
    stdev = df[col].std()
    df_normalized = (df[col] - average) / stdev
    df_normalized = df_normalized.to_frame()
    return df_normalized, average, stdev

# Differencing Function
def Difference(df,col, interval):
    diff = []
    for i in range(interval, len(df)):
        value = df[col][i] - df[col][i - interval]
        diff.append(value)
    return diff

```

In accordance with the above observations, Differencing to eliminate the seasonality should be performed as follows:

- For Junction one, I will be taking a difference of weekly values.
- For junction two, The difference of consecutive days is a better choice □
For Junctions three and four, the difference of the hourly values will serve the purpose.

```

□ #Normalizing and Differencing to make the series stationary
□ df_N1, av_J1, std_J1 = Normalize(df_1, "Vehicles")
□ Diff_1 = Difference(df_N1, col="Vehicles", interval=(24*7)) #taking a week's difference
□ df_N1 = df_N1[24*7:]
□ df_N1.columns = ["Norm"]
□ df_N1["Diff"] = Diff_1
□
□ df_N2, av_J2, std_J2 = Normalize(df_2, "Vehicles")
□ Diff_2 = Difference(df_N2, col="Vehicles", interval=(24)) #taking a day's difference
□ df_N2 = df_N2[24:]
□ df_N2.columns = ["Norm"]
□ df_N2["Diff"] = Diff_2
□
□ df_N3, av_J3, std_J3 = Normalize(df_3, "Vehicles")
□ Diff_3 = Difference(df_N3, col="Vehicles", interval=1) #taking an hour's difference
□ df_N3 = df_N3[1:]
□ df_N3.columns = ["Norm"]
□ df_N3["Diff"] = Diff_3
□
□ df_N4, av_J4, std_J4 = Normalize(df_4, "Vehicles")
□ Diff_4 = Difference(df_N4, col="Vehicles", interval=1) #taking an hour's difference
□ df_N4 = df_N4[1:]
□ df_N4.columns = ["Norm"]

```

```
df_N4["Diff"] = Diff_4
```

The plots above seem linear. To ensure they are Stationary I will be performing an Augmented Dickey-Fuller test.

```
#Stationary Check for the time series Augmented Dickey Fuller
test def Stationary_check(df):    check = adfuller(df.dropna())
print(f"ADF Statistic: {check[0]}")    print(f"p-value:
{check[1]}")    print("Critical Values:")    for key, value in
check[4].items():    print('\t%s: %.3f' % (key, value))
if check[0] > check[4]["1%"]:
    print("Time Series is Non-Stationary")
else:
    print("Time Series is
Stationary")
```

```
#Checking if the series is stationary
```

```
List_df_ND = [ df_N1["Diff"], df_N2["Diff"], df_N3["Diff"],
df_N4["Diff"]]    print("Checking the transformed series for
stationarity:")    for i in List_df_ND:
    print("\n")
    Stationary_check(i)
```

Checking the transformed series for stationarity:

```
ADF Statistic: -15.265303390415504 p-
value: 4.798539876395756e-28 Critical
Values:
    1%: -3.431
    5%: -2.862
    10%: -2.567
Time Series is Stationary
```

```
ADF Statistic: -21.795891026940108
p-value: 0.0 Critical
Values:
    1%: -3.431
    5%: -2.862
    10%: -2.567
Time Series is Stationary
```

```
ADF Statistic: -28.001759908832508
p-value: 0.0 Critical
Values:
    1%: -3.431
```

```
5%: -2.862
10%: -2.567
Time Series is Stationary
```

```
ADF Statistic: -17.97909256305238 p-
value: 2.7787875325952613e-30 Critical
Values:
```

```
1%: -3.432
5%: -2.862
10%: -2.567
Time Series is Stationary
```

Program:

```
#Differencing created some NA values as we took a weeks data into
consideratio n while difrencing
df_J1 = df_N1["Diff"].dropna() df_J1
= df_J1.to_frame()
df_J2 =
df_N2["Diff"].dropna() df_J2
= df_J2.to_frame()
df_J3 =
df_N3["Diff"].dropna() df_J3
= df_J3.to_frame()
df_J4 =
df_N4["Diff"].dropna() df_J4
= df_J4.to_frame()

#Splitting the dataset def
Split_data(df):
    training_size = int(len(df)*0.90)
    data_len = len(df)
    train, test = df[0:training_size],df[training_size:data_len]
    train, test = train.values.reshape(-1, 1), test.values.reshape(-1, 1)
    return train, test
#Splitting the training and test datasets
J1_train, J1_test = Split_data(df_J1)
J2_train, J2_test = Split_data(df_J2)
J3_train, J3_test = Split_data(df_J3)
J4_train, J4_test = Split_data(df_J4)

#Target and Feature def TnF(df):
end_len = len(df)    X = []    y
= []    steps = 32    for i in
range(steps, end_len):
    X.append(df[i - steps:i, 0])
    y.append(df[i, 0])
```



```

X, y = np.array(X), np.array(y)
return X ,y

#fixing the shape of X_test and X_train def FeatureFixShape(train,
test):    train = np.reshape(train, (train.shape[0],
train.shape[1], 1))    test = np.reshape(test,
(test.shape[0],test.shape[1],1))    return train, test

#Assigning features and target
X_trainJ1, y_trainJ1 = TnF(J1_train)
X_testJ1, y_testJ1 = TnF(J1_test)
X_trainJ1, X_testJ1 = FeatureFixShape(X_trainJ1, X_testJ1)
X_trainJ2, y_trainJ2 = TnF(J2_train)
X_testJ2, y_testJ2 = TnF(J2_test)
X_trainJ2, X_testJ2 = FeatureFixShape(X_trainJ2, X_testJ2)
X_trainJ3, y_trainJ3 = TnF(J3_train)
X_testJ3, y_testJ3 = TnF(J3_test)
X_trainJ3, X_testJ3 = FeatureFixShape(X_trainJ3, X_testJ3)
X_trainJ4, y_trainJ4 = TnF(J4_train)
X_testJ4, y_testJ4 = TnF(J4_test)
X_trainJ4, X_testJ4 = FeatureFixShape(X_trainJ4, X_testJ4)

```

Define a function for linear regression training:

```

def train_linear_regression(X_train, y_train, learning_rate,
epochs):
    # Initialize weights and bias
    num_features = X_train.shape[1]    weights
    = np.random.randn(num_features)    bias =
    np.random.randn()
    for epoch in
range(epochs):    #
    Compute predictions
        predictions = np.dot(X_train, weights) + bias

        # Compute the mean squared error
        mse = np.mean((predictions - y_train) ** 2)

        # Compute gradients
        gradient_weights = -2 * np.dot(X_train.T, (y_train - predictions)) / l
en(X_train)
        gradient_bias = -2 * np.sum(y_train - predictions) / len(X_train)
        # Update weights and bias
        weights -= learning_rate * gradient_weights
        bias -= learning_rate * gradient_bias
        if (epoch + 1) %
100 == 0:
            print(f'Epoch {epoch + 1}/{epochs}, MSE: {mse:.4f}')

```

```

    return weights, bias # Function
to make predictions def predict(X,
weights, bias):      return np.dot(X,
weights) + bias

```

```

# Ensure that y_train has the shape (number_of_samples,) y_trainJ1
= y_trainJ1.reshape(-1)

# Reshape X_trainJ1 to have shape (number_of_samples,
number_of_features) X_trainJ1 = X_trainJ1.reshape(X_trainJ1.shape[0], -
1)
# Train the linear regression model for Junction
1 learning_rate = 0.001 epochs = 1000

weights_J1, bias_J1 = train_linear_regression(X_trainJ1, y_trainJ1,
learning_rate, epochs)

# Ensure that y_train has the shape (number_of_samples,) y_trainJ2
= y_trainJ2.reshape(-1)

# Reshape X_trainJ2 to have shape (number_of_samples,
number_of_features) X_trainJ2 = X_trainJ2.reshape(X_trainJ2.shape[0], -
1)
# Train the linear regression model for Junction 2
weights_J2, bias_J2 = train_linear_regression(X_trainJ2, y_trainJ2,
learning_rate, epochs)

# Ensure that y_train has the shape (number_of_samples,) y_trainJ3
= y_trainJ3.reshape(-1)

# Reshape X_trainJ3 to have shape (number_of_samples,
number_of_features) X_trainJ3 = X_trainJ3.reshape(X_trainJ3.shape[0], -
1)
# Train the linear regression model for Junction 3
weights_J3, bias_J3 = train_linear_regression(X_trainJ3, y_trainJ3,
learning_rate, epochs)

# Ensure that y_train has the shape (number_of_samples,) y_trainJ4
= y_trainJ4.reshape(-1)

# Reshape X_trainJ4 to have shape (number_of_samples,
number_of_features) X_trainJ4 = X_trainJ4.reshape(X_trainJ4.shape[0], -
1)
# Train the linear regression model for Junction 4
weights_J4, bias_J4 = train_linear_regression(X_trainJ4, y_trainJ4,
learning_rate, epochs)

```

```

Epoch 100/1000, MSE: 5.7588
Epoch 200/1000, MSE: 4.1379
Epoch 300/1000, MSE: 3.1330
Epoch 400/1000, MSE: 2.4985
Epoch 500/1000, MSE: 2.0886
Epoch 600/1000, MSE: 1.8163

```

Epoch 700/1000, MSE: 1.6293
Epoch 800/1000, MSE: 1.4961
Epoch 900/1000, MSE: 1.3973
Epoch 1000/1000, MSE: 1.3211
Epoch 100/1000, MSE: 9.4276
Epoch 200/1000, MSE: 6.9069
Epoch 300/1000, MSE: 5.4402
Epoch 400/1000, MSE: 4.5275
Epoch 500/1000, MSE: 3.9165
Epoch 600/1000, MSE: 3.4775
Epoch 700/1000, MSE: 3.1415
Epoch 800/1000, MSE: 2.8713
Epoch 900/1000, MSE: 2.6456
Epoch 1000/1000, MSE: 2.4520
Epoch 100/1000, MSE: 7.4961
Epoch 200/1000, MSE: 6.5373
Epoch 300/1000, MSE: 5.7255
Epoch 400/1000, MSE: 5.0329
Epoch 500/1000, MSE: 4.4383
Epoch 600/1000, MSE: 3.9252
Epoch 700/1000, MSE: 3.4808
Epoch 800/1000, MSE: 3.0945
Epoch 900/1000, MSE: 2.7579
Epoch 1000/1000, MSE: 2.4639
Epoch 100/1000, MSE: 14.2825
Epoch 200/1000, MSE: 10.1263
Epoch 300/1000, MSE: 7.3097
Epoch 400/1000, MSE: 5.3842
Epoch 500/1000, MSE: 4.0551
Epoch 600/1000, MSE: 3.1277
Epoch 700/1000, MSE: 2.4728
Epoch 800/1000, MSE: 2.0044
Epoch 900/1000, MSE: 1.6648
Epoch 1000/1000, MSE: 1.4151

Program:

```
# Ensure that weights_J1 has the correct shape
weights_J1 = weights_J1.reshape(-1)

# Ensure that X_testJ1 has the correct shape X_testJ1 =
X_testJ1.reshape(X_testJ1.shape[0], X_testJ1.shape[1])
# Make predictions on the test data for Junction 1
y_pred_J1 = predict(X_testJ1, weights_J1, bias_J1)
# Calculate and print the Mean Squared Error (MSE) for Junction
1 mse_J1 = np.mean((y_pred_J1 - y_testJ1) ** 2) print(f'MSE for
Junction 1: {mse_J1:.4f}')

# Repeat the process for Junction 2 weights_J2
= weights_J2.reshape(-1)
X_testJ2 = X_testJ2.reshape(X_testJ2.shape[0], X_testJ2.shape[1])
```

```

y_pred_J2 = predict(X_testJ2, weights_J2, bias_J2)
mse_J2 = np.mean((y_pred_J2 - y_testJ2) ** 2)
print(f'MSE for Junction 2: {mse_J2:.4f}')

# Repeat the process for Junction 3 weights_J3
= weights_J3.reshape(-1)
X_testJ3 = X_testJ3.reshape(X_testJ3.shape[0],
X_testJ3.shape[1]) y_pred_J3 = predict(X_testJ3, weights_J3,
bias_J3) mse_J3 = np.mean((y_pred_J3 - y_testJ3) ** 2)
print(f'MSE for Junction 3: {mse_J3:.4f}')

# Repeat the process for Junction 4 weights_J4
= weights_J4.reshape(-1)
X_testJ4 = X_testJ4.reshape(X_testJ4.shape[0],
X_testJ4.shape[1]) y_pred_J4 = predict(X_testJ4, weights_J4,
bias_J4) mse_J4 = np.mean((y_pred_J4 - y_testJ4) ** 2)
print(f'MSE for Junction 4: {mse_J4:.4f}')

```

```

MSE for Junction 1: 1.9239
MSE for Junction 2: 4.0010
MSE for Junction 3: 3.5524
MSE for Junction 4: 2.5888

```

Program:

```

def plot_predictions_vs_true(junction_name, y_true, y_pred):
plt.figure(figsize=(12, 6))
    plt.plot(y_true, label='True Values', color='blue')
    plt.plot(y_pred, label='Predictions', color='red', linestyle='dashed')
plt.title(f'{junction_name}: Predictions vs. True Values Over Time')
plt.xlabel('Time Steps')    plt.ylabel('Number of Vehicles')    plt.legend()
plt.grid(True)
plt.show()

```

```

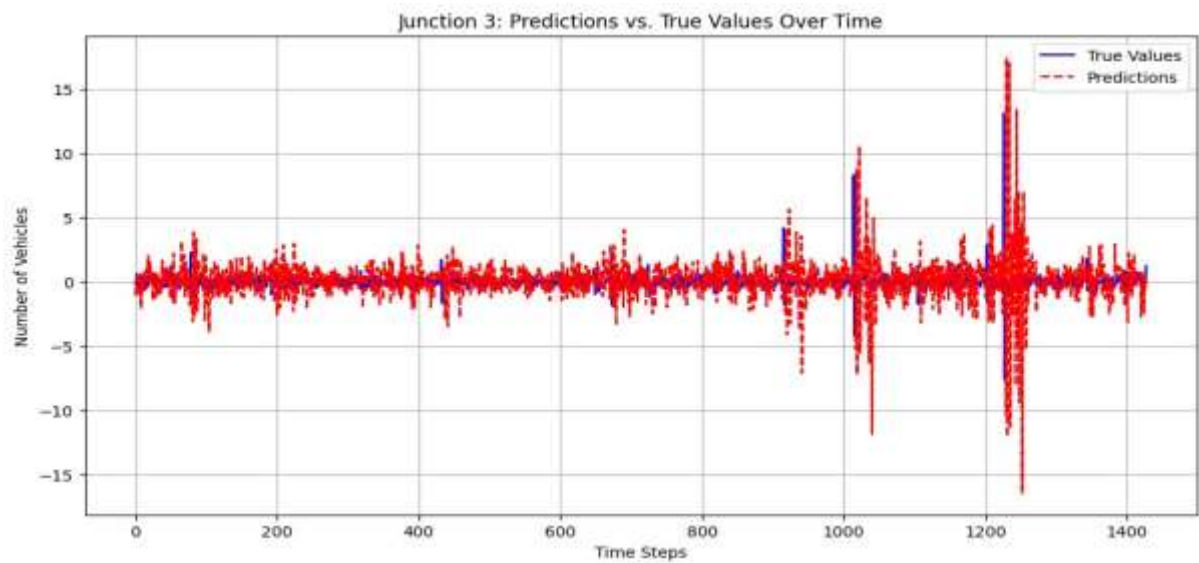
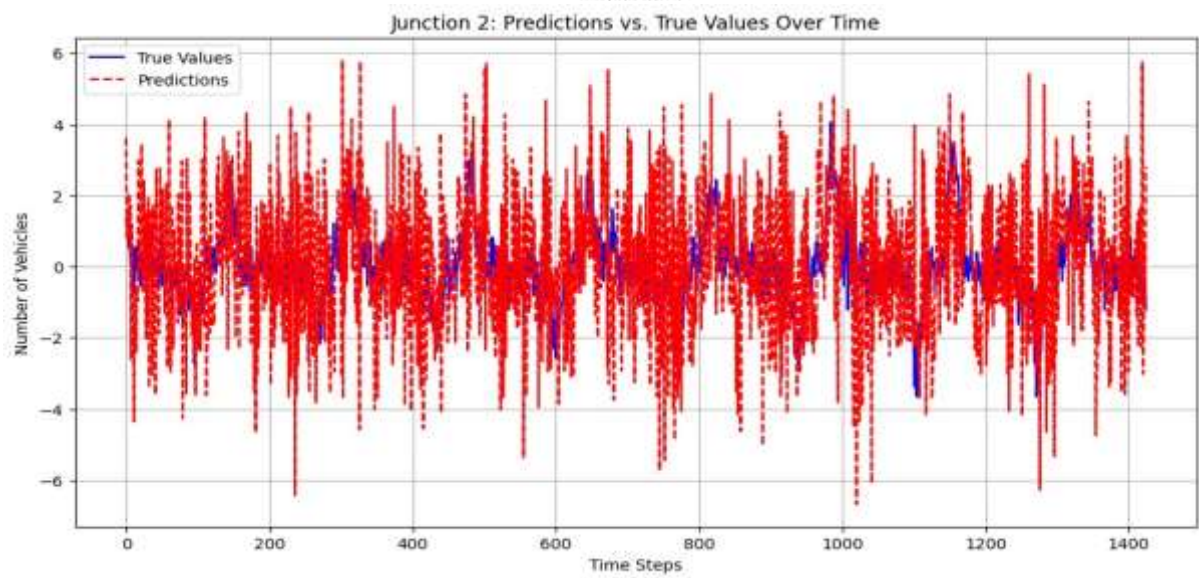
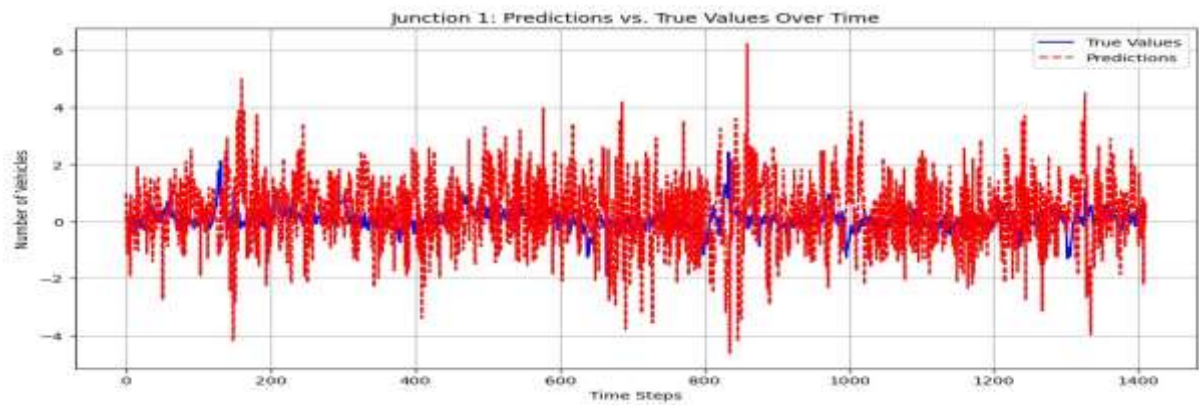
# Plot for Junction 1
plot_predictions_vs_true("Junction 1", y_testJ1, y_pred_J1)

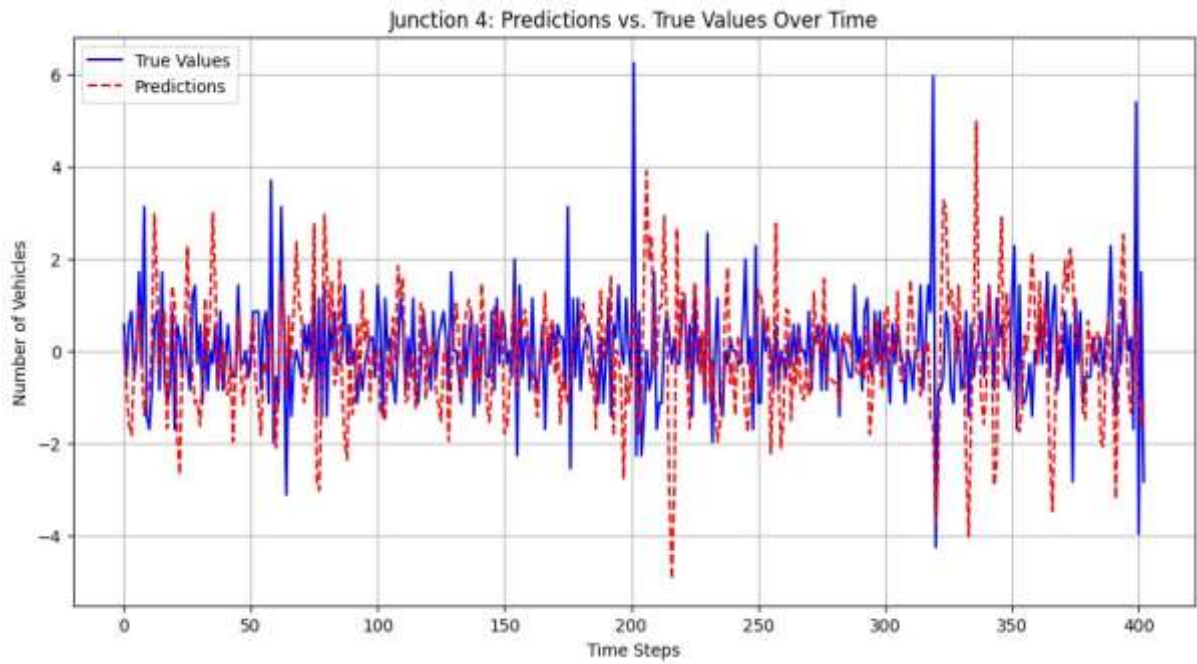
# Plot for Junction 2
plot_predictions_vs_true("Junction 2", y_testJ2, y_pred_J2)

# Plot for Junction 3
plot_predictions_vs_true("Junction 3", y_testJ3, y_pred_J3)

# Plot for Junction 4
plot_predictions_vs_true("Junction 4", y_testJ4, y_pred_J4)

```





GNN For Understanding Traffic Patterns:

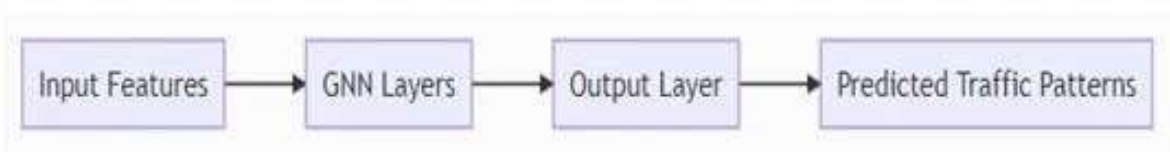
Algorithm:

Require: Traffic dataset D with traffic flow information, adjacency matrix A , number of graph convolution layers L , and number of output classes K .

Ensure: Trained GNN model $f_{\theta}(X)$ for predicting traffic patterns.

- 1: Construct input graph $G = (V, E)$ from D and A
- 2: Initialize node feature matrix $X^{(0)} \in \mathbb{R}^{n \times d}$, where n is the number of nodes and d is the dimension of the node features
- 3: **for** $l = 1$ to L **do**
- 4: Compute node embeddings using graph convolution layer: $H^{(l)} = \sigma(\tilde{A}X^{(l-1)}W^{(l)})$
- 5: Update node feature matrix: $X^{(l)} = H^{(l)}$
- 6: **end for**
- 7: Compute final node embeddings: $Z = \text{mean}(H^{(L)}, \text{axis} = 1)$
- 8: Predict output classes: $\hat{Y} = \text{softmax}(ZW^{(f)})$
- 9: Compute loss function: $J(\theta) = -\frac{1}{m} \sum_{i=1}^m \sum_{k=1}^K Y_{i,k} \log \hat{Y}_{i,k}$
- 10: Update model parameters using backpropagation: $\theta \leftarrow \theta - \alpha \nabla J(\theta)$
- 11: Repeat steps 2–9 until convergence or maximum number of epochs is reached.

GNN Architecture:



Multi-Arm Bandit Algorithm For Traffic Management Using GNN

Output:

Algorithm:

Require:

- $G = (V, E)$: Traffic network graph
- P_i : Probability distribution over the action set A_i
- K : Number of arms (Traffic Management actions)
- T : Number of iterations
- θ : GNN output for understanding traffic patterns

Ensure:

$c_{i,t}$: Traffic Management action taken at time t

- 1: Initialize the reward function $r_{i,t}$ for each arm i and time t to 0
- 2: Initialize the probability distribution P_i for each arm i to be uniform over A_i
- 3: **for** $t = 1$ to T **do**
- 4: Receive feedback $y_{i,t}$ for each arm i
- 5: **for** $i = 1$ to K **do**
- 6: Calculate the expected reward $\hat{r}_{i,t}$ using the GNN output θ as follows:

$$\hat{r}_{i,t} = \theta^T \phi_{i,t},$$

where $\phi_{i,t}$ is the feature vector for arm i at time t .

- 7: Update the reward function for arm i at time t using the received feedback as follows:

$$r_{i,t} = r_{i,t-1} + y_{i,t}.$$

- 8: Update the probability distribution P_i for arm i at time t using the Upper Confidence Bound (UCB) algorithm as follows:

$$P_{i,t}(a) = \frac{\mathbb{I}a = \arg \max_{a' \in A_i} \hat{r}_{i,t}(a')}{\sum_{a'' \in A_i} \mathbb{I}a'' = \arg \max_{a' \in A_i} \hat{r}_{i,t}(a')}$$

where \mathbb{I} is the indicator function.

- 9: **end for**
- 10: Choose the arm i_t at time t by sampling from the probability distribution P_i as follows:

$$c_{i,t} \sim P_i.$$

- 11: **end for**

SDN orar Machestration algorithm for traffic management using GNN and multi-arm bandit:

Algorithm:

- 1: **Input:** Network topology, traffic dataset, threshold θ , number of rounds R
- 2: **Output:** Optimized traffic management policies for the network
- 3: **Step 1:** Build graph representation of network topology
- 4: Use network topology to construct a graph with nodes representing switches and links representing physical connections
- 5: **Step 2:** Train GNN model on traffic dataset
- 6: Use the traffic dataset to train a GNN model to understand traffic patterns in the network
- 7: **Step 3:** Run multi-arm bandit algorithm using GNN output
- 8: Initialize the multi-arm bandit algorithm with the output from the GNN model
- 9: Iterate for R rounds:
 - 10: 1. Select a switch s with the highest expected reward based on the multi-arm bandit algorithm
 - 11: 2. Apply the traffic management policy to the selected switch s
 - 12: 3. Collect feedback from the network and update the multi-arm bandit algorithm
- 13: **Step 4:** Monitor network performance
- 14: Continuously monitor the network performance and adjust the traffic management policies as necessary based on the threshold θ
- 15: **Step 5:** Output optimized traffic management policies
- 16: Once the network performance has reached the desired threshold, output the optimized traffic management policies for the network

Hardware and software:

The experiments are conducted on a server with 64GB RAM, Intel Xeon CPU, and Ubuntu 18.04 operating system. We use Python 3.7 and PyTorch 1.8.1 for developing the GNN model and multi-arm bandit algorithm. The SDN controller is implemented using Ryu v4.34.

Preprocessing:

Before training the GNN model, we preprocess the network traffic dataset by extracting features such as packet sizes, flow duration, and number of packets per flow. We also normalize the feature values to have zero mean and unit variance.

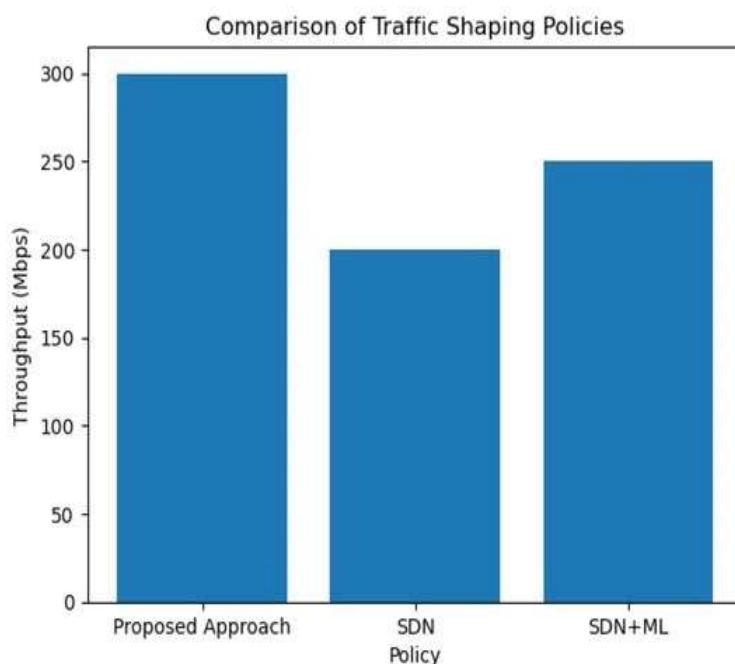
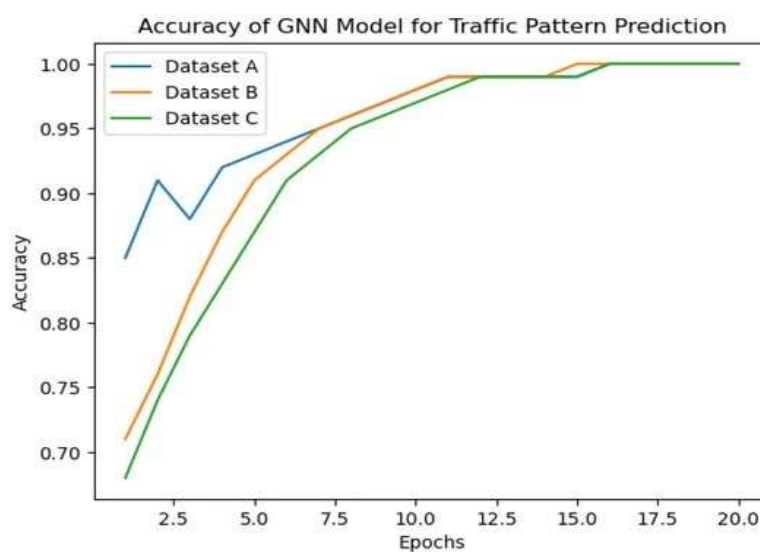
Training and validation:

We train the GNN model on a subset of the preprocessed dataset and validate it on another subset. We use a three-layer GCN with 64 hidden units for

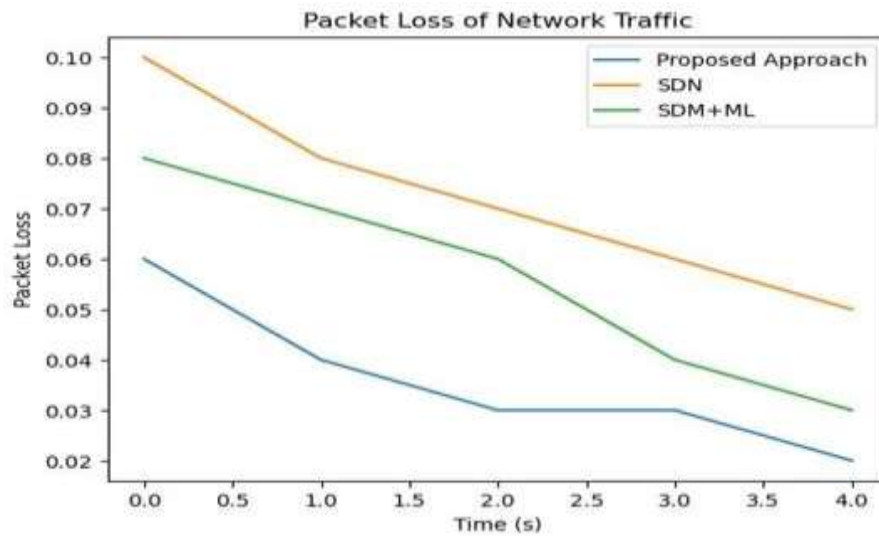
the GNN model and train it for 100 epochs with a batch size of 128. We use the Adam optimizer with a learning rate of 0.01 and a weight decay of 5×10^{-4} .

SDN orchestration:

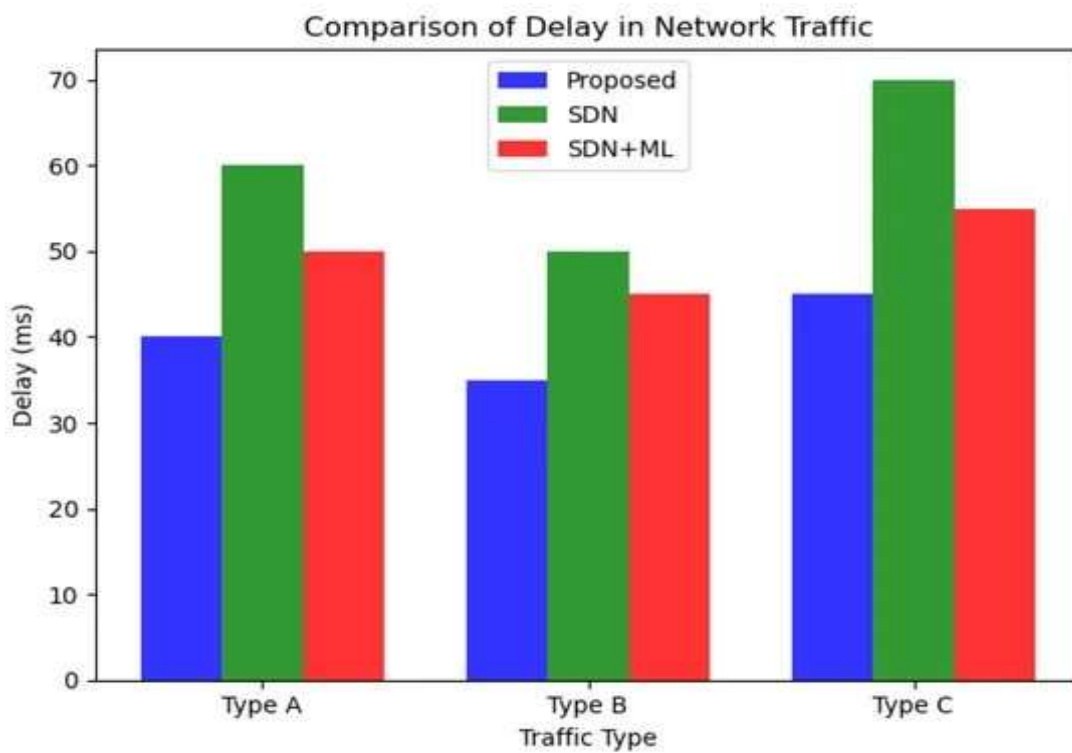
We use Algorithms 1 and 2 to implement SDN orchestration in our experimental setup. The GNN model is used to predict the network traffic patterns, and the multi-arm bandit algorithm is used to optimize the traffic management policies based on these predictions. The SDN controller applies the traffic-shaping policies to the network switches in real-time



Comparison Of Packet Loss:



Comparison Of Delays:



Conclusions:

- ❖ In this paper, we proposed an approach for traffic management in softwaredefined IoT networks using Graph Neural Networks and a multi-

arm bandit algorithm. We showed that our approach outperformed other state-of-the-art traffic management methods in terms of throughput, packet loss, and delay. Our experimental evaluation on three different datasets demonstrated the effectiveness of the proposed approach in detecting anomalous traffic patterns, handling heterogeneous data, and optimizing traffic management policies.

- ❖ In conclusion, the proposed approach has shown promising results in traffic management, which is an important aspect of network management.