

Code link:

https://colab.research.google.com/drive/1iUqsbWtDc9EyWVleSXast5QFBFt3g1f7#scrollTo=Pjj_9ygGTlqZ

DataSet Link: <https://www.kaggle.com/datasets/fedesoriano/traffic-prediction-dataset/data>

STUDENT NAME: UZUM STANLEY EKENE

STUDENT ID: UZU22571175

Traffic Prediction for Road Networks Using Deep Learning

In this comprehensive project, I aim to delve into the intricacies of traffic flow by examining a dataset encompassing four critical junctions. My objective is to construct a robust predictive model that can accurately forecast traffic volumes. By harnessing advanced analytical techniques, I anticipate uncovering underlying traffic patterns and trends. This insight is pivotal, as it could significantly contribute to alleviating traffic congestion issues. A thorough understanding of these dynamics will be instrumental in guiding the development of optimized infrastructure. Such enhancements are expected to streamline traffic management, thereby reducing bottlenecks and improving overall transportation efficiency. Ultimately, this endeavor seeks to pave the way for smoother commutes and a more sustainable urban mobility framework.

Table Of Contents

- [1. IMPORTING LIBRARIES](#)
- [2. LOADING DATA](#)
- [3. DATA EXPLORATION](#)
 - [3.1 FEATURE ENGINEERING](#)
 - [3.2 EXPLORATORY DATA ANALYSIS](#)
- [4. DATA TRANSFORMATION AND PREPROCESSING](#)
- [5. MODEL BUILDING](#)
- [6. FITTING THE MODEL](#)
- [7. INVERSING THE TRANSFORMATION OF DATA](#)
- [8. END](#)

▼ Importing Libraries

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

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

▼ Loading Data

```
from google.colab import drive
drive.mount('/content/drive')
```

Mounted at /content/drive

#Loading Data

```
dataset = pd.read_csv("/content/drive/MyDrive/DEEP LEARNING/traffic.csv")
dataset.head()
```

	DateTime	Junction	Vehicles	ID
0	2015-11-01 00:00:00	1	15	20151101001
1	2015-11-01 01:00:00	1	13	20151101011
2	2015-11-01 02:00:00	1	10	20151101021
3	2015-11-01 03:00:00	1	7	20151101031
4	2015-11-01 04:00:00	1	9	20151101041

```
data = dataset.copy()
data.shape
```

```
(48120, 4)
```

About the data

This dataset represents a meticulously compiled log of vehicular counts, recorded at hourly intervals across four pivotal junctions. It serves as a valuable resource for analyzing traffic flow patterns, providing insights into peak hours, and facilitating data-driven strategies to enhance road network efficiency and reduce congestion. The CSV file provides four features:

- DateTime
- Junctions
- Vehicles
- ID

The sensors on each of these junctions were collecting data at different times, hence the traffic data from different time periods. Some of the junctions have provided limited or sparse data.

▼ Data Exploration

- Check for missing values
- Pharsing dates
- Ploting timeseris.
- Feature engineering for EDA

```
# Check for missing values
missing_values = data.isnull().sum()
# Print the missing values count for each column
print("Missing values count for each column:")
print(missing_values)
# Check if there are any missing values in the entire dataset
if missing_values.any():
    print("\nThere are missing values in the dataset.")
else:
    print("\nNo missing values found in the dataset.")
```

Missing values count for each column:

```
DateTime    0
Junction    0
Vehicles    0
ID           0
dtype: int64
```

No missing values found in the dataset.

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

pandas.core.frame.DataFrame.info

```
def info(verbose: bool | None=None, buf: WriteBuffer[str] | None=None,
max_cols: int | None=None, memory_usage: bool | str | None=None,
show_counts: bool | None=None, null_counts: bool | None=None) -> None
```

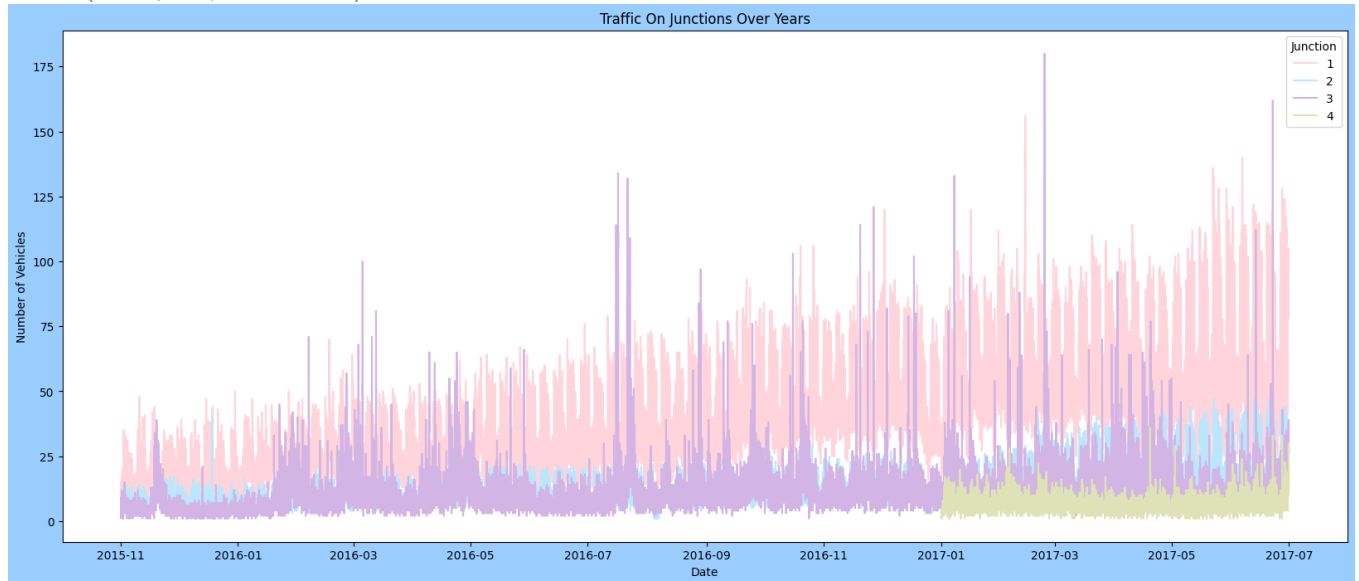
Print a concise summary of a DataFrame.

This method prints information about a DataFrame including the index dtype and columns, non-null values and memory usage.

Parameters

```
#df to be used for EDA
df=data.copy()
#Let's plot the Timeseries
colors = [ "#FFD4DB", "#BBE7FE", "#D3B5E5", "#dfe2b6"]
plt.figure(figsize=(20,8),facecolor="#99ccff")
Time_series=sns.lineplot(x=df['DateTime'],y="Vehicles",data=df, hue="Junction",
Time_series.set_title("Traffic On Junctions Over Years")
Time_series.set_ylabel("Number of Vehicles")
Time_series.set_xlabel("Date")
```

```
Text(0.5, 0, 'Date')
```



Noticeable information in the above plot:

- It can be seen here that the first junction is visibly having an upward trend.
- The data for the fourth junction is sparse starting only after 2017
- Seasonality is not evident from the above plot, So we must explore datetime composition to figure out more about it.

Feature Engineering

At this step, I am creating a few new features out of DateTime. Namely:

- Year
- Month
- Date in the given month
- Days of week
- Hour

#Exploring more features

```
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()
```

	DateTime	Junction	Vehicles	Year	Month	Date_no	Hour	Day
0	2015-11-01 00:00:00	1	15	2015	11	1	0	Sunday
1	2015-11-01 01:00:00	1	13	2015	11	1	1	Sunday
2	2015-11-01 02:00:00	1	10	2015	11	1	2	Sunday
3	2015-11-01 03:00:00	1	7	2015	11	1	3	Sunday
4	2015-11-01 04:00:00	1	9	2015	11	1	4	Sunday

✎ Exploratory Data Analysis

Plotting the newly created features

#Let's plot the Timeseries

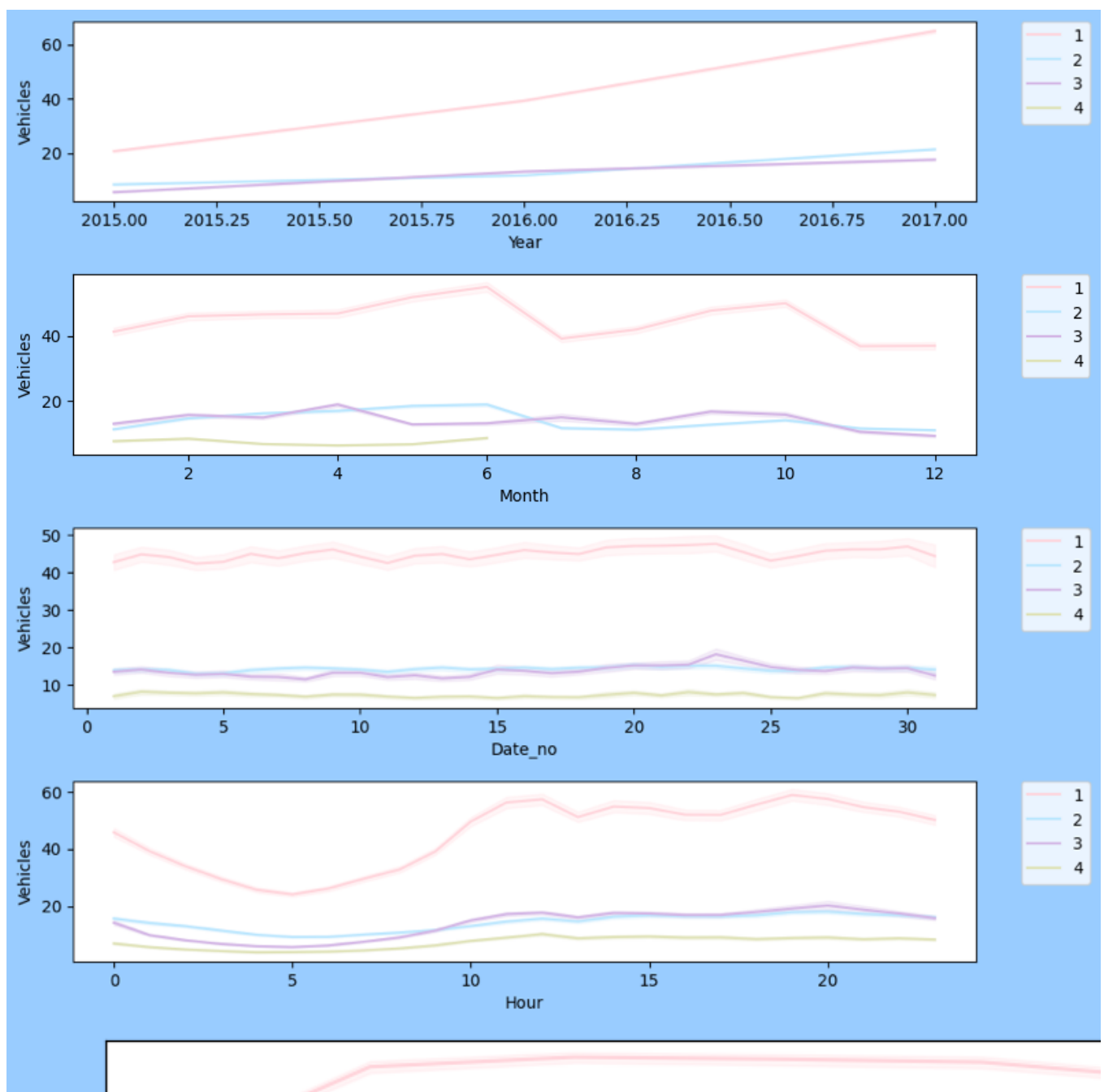
```
new_features = [ "Year","Month", "Date_no", "Hour", "Day"]
```

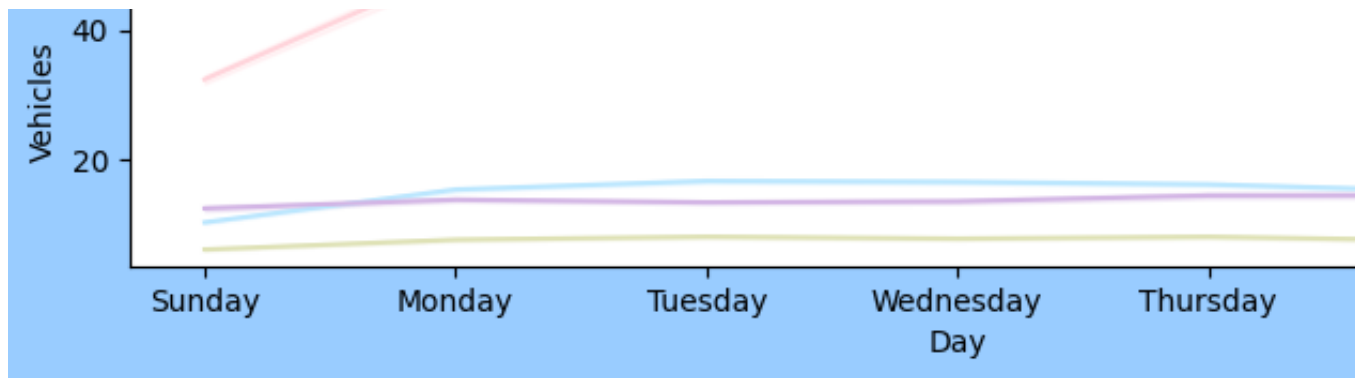
```
for i in new_features:
```

```
    plt.figure(figsize=(10, 2),facecolor="#99ccff")
```

```
    ax=sns.lineplot(x=df[i],y="Vehicles",data=df, hue="Junction", palette=
```

```
    plt.legend(bbox_to_anchor=(1.05, 1), loc=2, borderaxespad=0.)
```

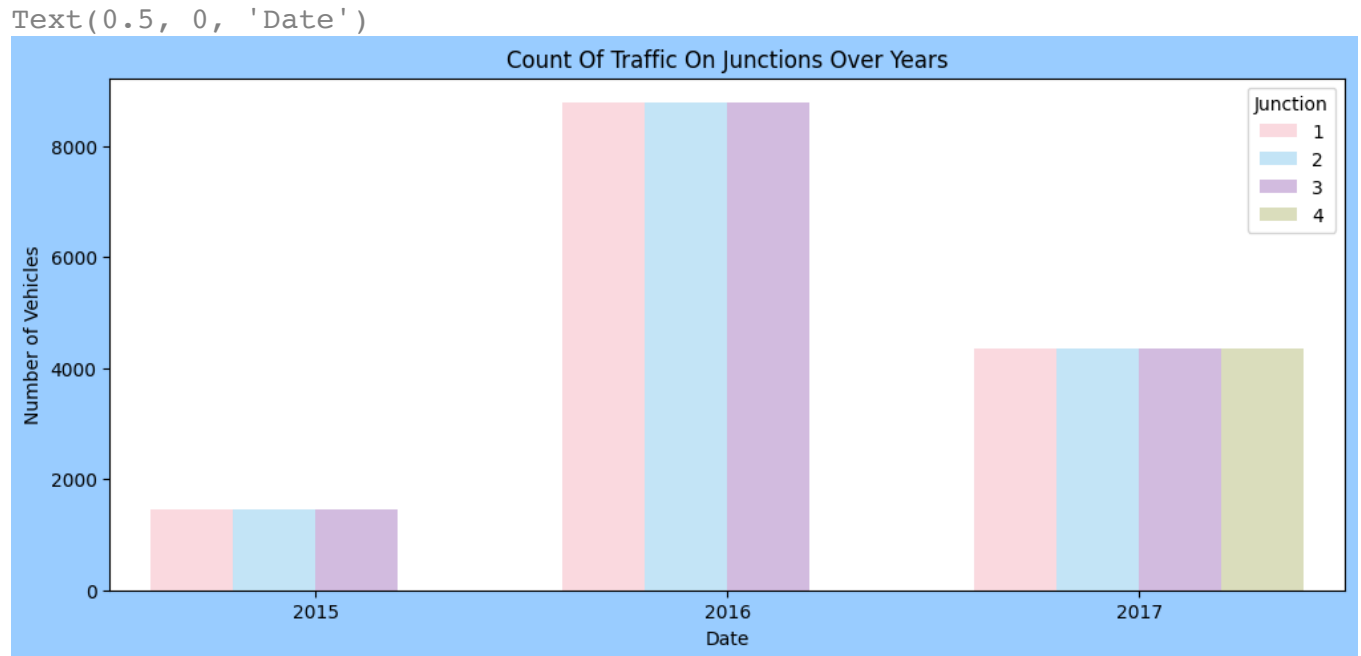




From the above plot following things can be concluded:

- 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.

```
plt.figure(figsize=(12,5),facecolor="#99ccff")
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")
```



The count plot shows that there is an increase in the number of vehicles between 2015 and 2016. However, it is inconclusive to say the same about 2017 as we have limited data for 2017 ie till the 7th month.

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

# Compute correlation matrix
corrmat = numeric_df.corr()

# Plot heatmap
plt.subplots(figsize=(10,10),facecolor="#99ccff")
sns.heatmap(corrmat,cmap= "Pastel2",annot=True,square=True)
```

<Axes: >

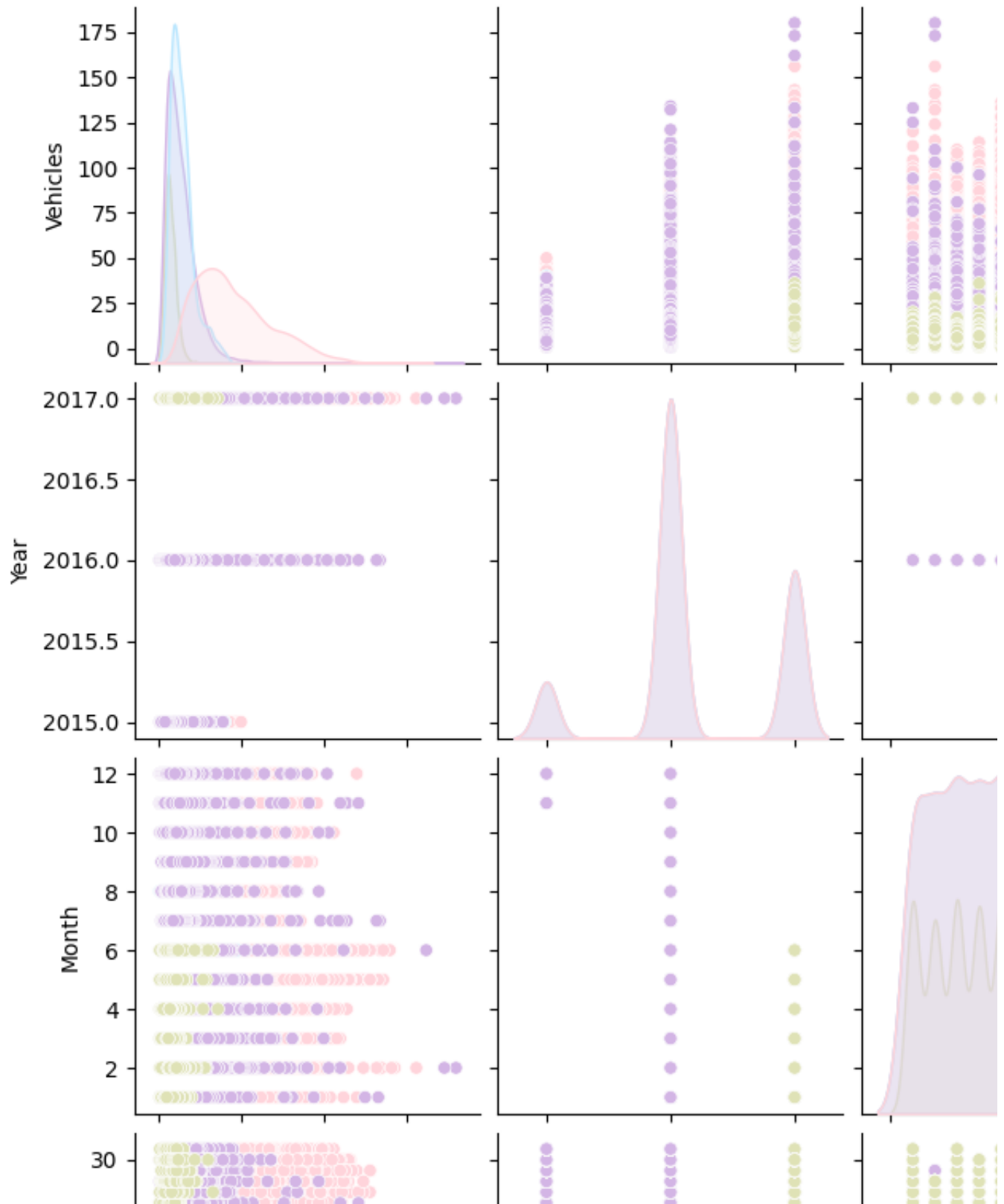


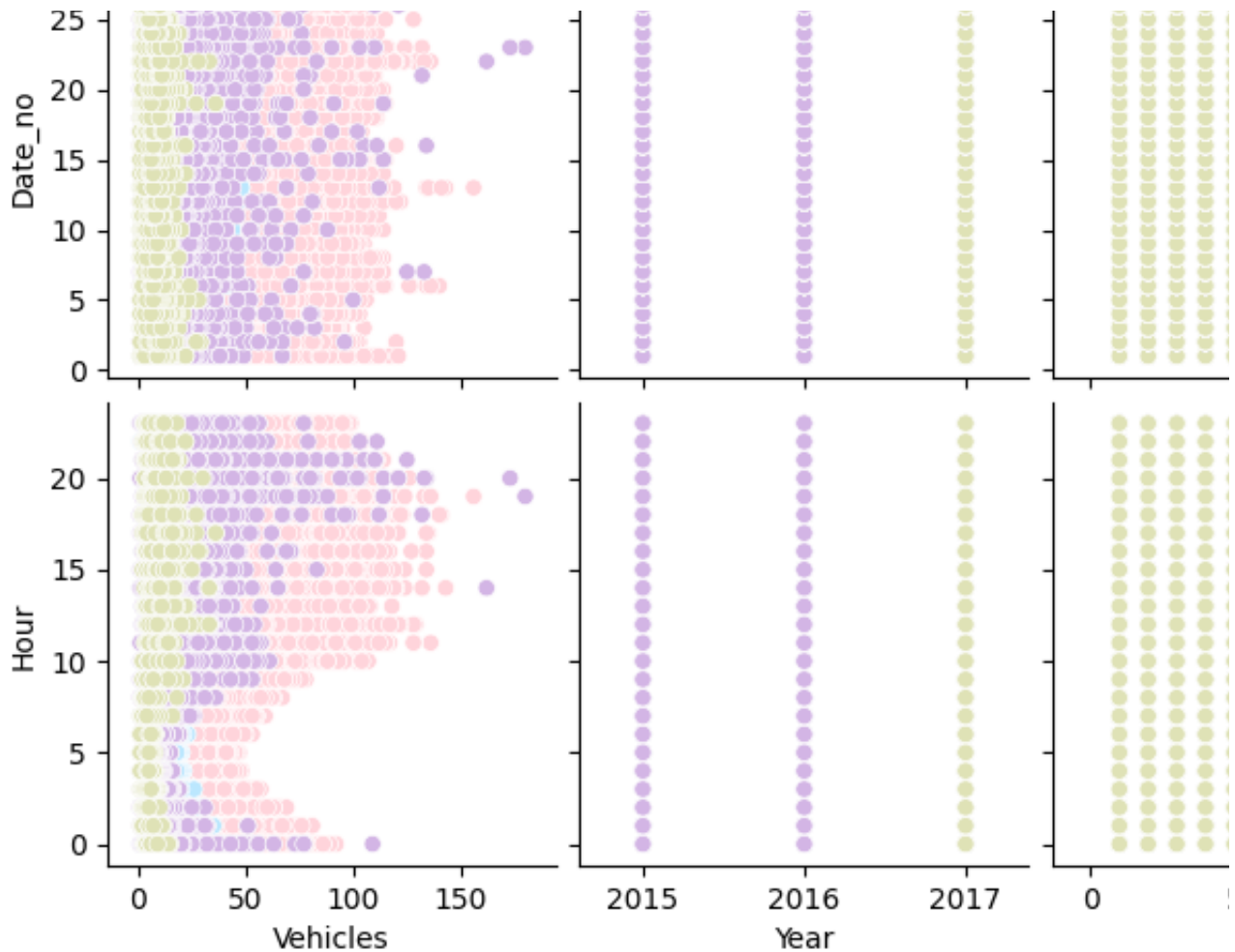
The highest correlation is certainly with the preexisting feature.

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 0x7935e93534f0>





Conclusions that I have come to draw after this EDA

- The span of data from all four junctions is not the same. Data provided for the fourth junction is limited to only 2017.
- The yearly trend for Junctions one, two and three have different slopes.
- Junction number one has a more strong weekly seasonality in comparison to the other junctions.

For the above-postulated reasons, I think that junctions must be transformed as per their individual needs.

✓ Data Transformation And Preprocessing

In this step I will be following the subsequent order:

- 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

```
#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

```
#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
```

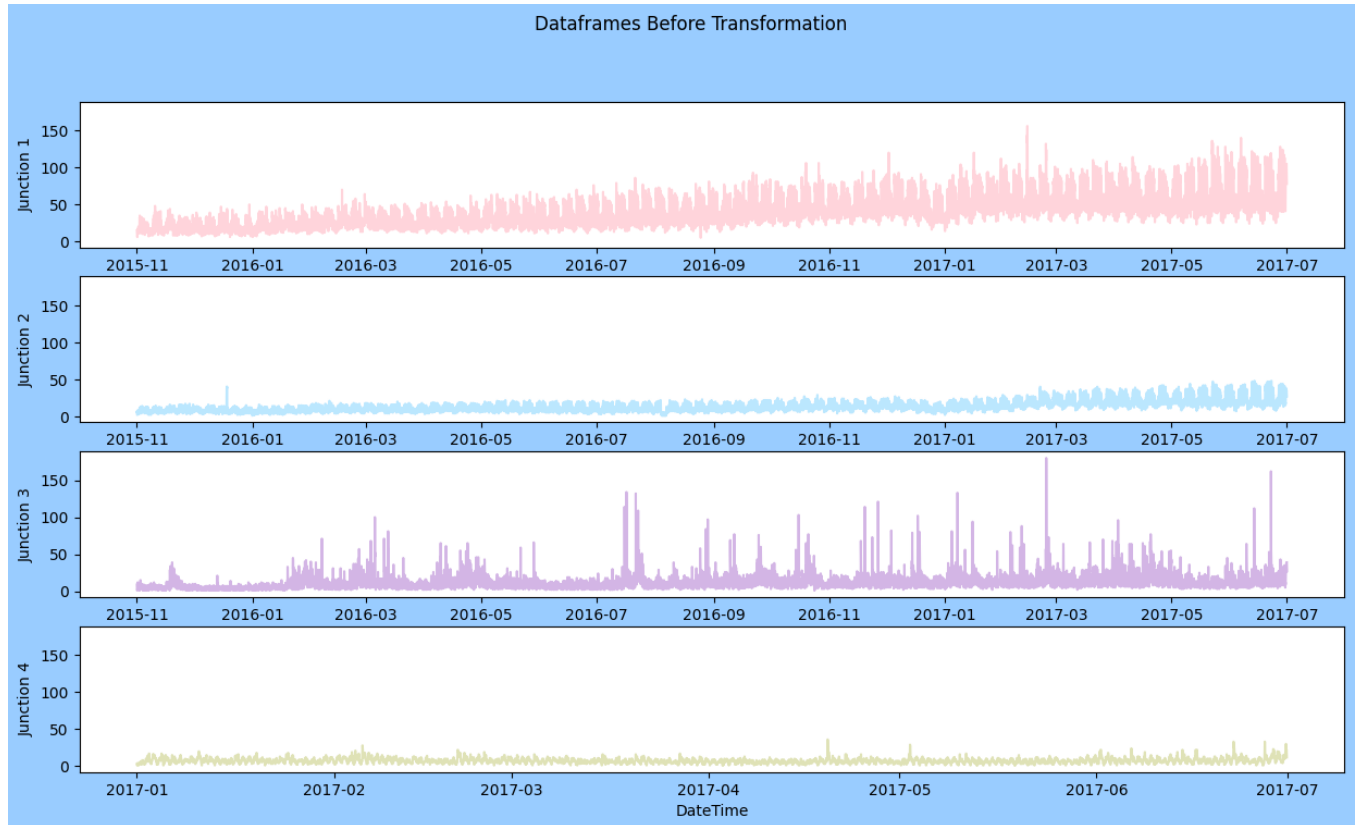
```
list_dfs = [df_1, df_2, df_3, df_4]
```

```
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="#99ccff", sharey=
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")

#Plotting the dataframe to check for stationarity
Sub_Plots4(df_1.Vehicles, df_2.Vehicles,df_3.Vehicles,df_4.Vehicles,"Dataframes
```

A time series is stationary if it does not have a trend or seasonality. However, in the EDA, we saw a weekly seasonality and an upwards trend over the years. In the above plot, it is again established that Junctions one and two have an upward trend. If we limit the span we will be able to further see the weekly seasonality. I will be spairing that step at this point and moving on with the respective transforms on datasets.

Steps for Transforming:

- Normalizing
- Differencing

```
# 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 di
df_N1 = df_N1[24*7:]
df_N1.columns = ["Norm"]
df_N1["Diff"] = Diff_1

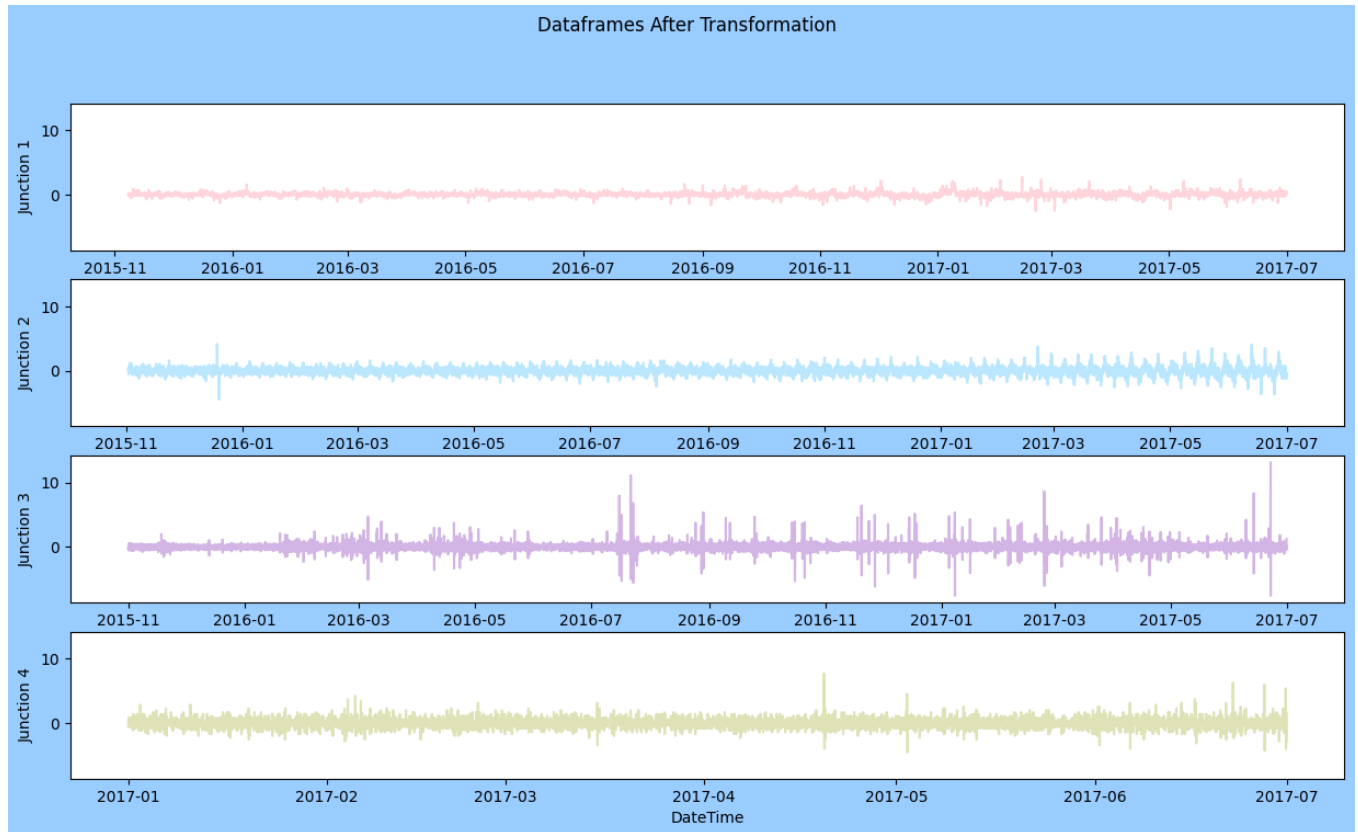
df_N2, av_J2, std_J2 = Normalize(df_2, "Vehicles")
Diff_2 = Difference(df_N2, col="Vehicles", interval=(24)) #taking a day's diffre
df_N2 = df_N2[24:]
df_N2.columns = ["Norm"]
df_N2["Diff"] = Diff_2

df_N3, av_J3, std_J3 = Normalize(df_3, "Vehicles")
Diff_3 = Difference(df_N3, col="Vehicles", interval=1) #taking an hour's diffre
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 diffre
df_N4 = df_N4[1:]
df_N4.columns = ["Norm"]
df_N4["Diff"] = Diff_4
```

Plots of Transformed Dataframe

```
Sub_Plots4(df_N1.Diff, df_N2.Diff,df_N3.Diff,df_N4.Diff,"Dataframes After Trans
```



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: -21.79589102694011
p-value: 0.0
Critical Values:
    1%: -3.431
    5%: -2.862
   10%: -2.567
Time Series is Stationary
```

```
ADF Statistic: -28.001759908832504
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
```

Now that the data is stationary, preprocessing the data for the neural net by:

- Splitting the test train sets
- Assigning X as features and y as target
- Reshaping data for neural net

```
#Differencing created some NA values as we took a weeks data into consideration
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)

```

▼ Model Building

In this section, I am creating a function for the neural net to call on and fit the data frames for all four junctions.

```

# Custom Layer
class CustomLayer(layers.Layer):
    def __init__(self, units=32):
        super(CustomLayer, self).__init__()
        self.units = units

    def build(self, input_shape):
        self.w = self.add_weight(
            shape=(input_shape[-1], self.units),
            initializer="random_normal",
            trainable=True,
        )
        self.b = self.add_weight(
            shape=(self.units,),
            initializer="zeros",
            trainable=True,
        )

    def call(self, inputs):
        return tf.matmul(inputs, self.w) + self.b

lr_schedule = ExponentialDecay(
    initial_learning_rate=0.01,
    decay_steps=10000,
    decay_rate=0.9)

# Custom_Model for the prediction
def Custom_model(X_Train, y_Train, X_Test, y_Test):
    early_stopping = callbacks.EarlyStopping(min_delta=0.001, patience=10, restore_best_weights=True)
    model = Sequential()

```



```

model.add(layers.Flatten(input_shape=(X_Train.shape[1], X_Train.shape[2])))
model.add(Dense(150, activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(50, activation='relu'))
model.add(Dropout(0.2))
model.add(CustomLayer(50)) # Custom layer
model.add(Dropout(0.2))
model.add(Dense(units=1))
model.compile(optimizer=SGD(learning_rate=lr_schedule, momentum=0.9), loss=
model.fit(X_Train, y_Train, validation_data=(X_Test, y_Test), epochs=50, ba
pred_Custom = model.predict(X_Test)
return pred_Custom

```

#GRU Model for the prediction

```

def GRU_model(X_Train, y_Train, X_Test, y_Test):
    early_stopping = callbacks.EarlyStopping(min_delta=0.001,patience=10, restc
    #callback delta 0.01 may interrupt the learning.

```

#The GRU model

```

model = Sequential()
model.add(GRU(units=150, return_sequences=True, input_shape=(X_Train.shape[1], X_Train.shape[2])))
model.add(Dropout(0.2))
model.add(GRU(units=150, return_sequences=True, input_shape=(X_Train.shape[1], X_Train.shape[2])))
model.add(Dropout(0.2))
model.add(GRU(units=50, return_sequences=True, input_shape=(X_Train.shape[1], X_Train.shape[2])))
model.add(Dropout(0.2))
model.add(GRU(units=50, return_sequences=True, input_shape=(X_Train.shape[1], X_Train.shape[2])))
model.add(Dropout(0.2))
#model.add(GRU(units=50, return_sequences=True, input_shape=(X_Train.shape[1], X_Train.shape[2])))
#model.add(Dropout(0.2))
model.add(GRU(units=50, input_shape=(X_Train.shape[1],1), activation='tanh'))
model.add(Dropout(0.2))
model.add(Dense(units=1))

```

#Compiling the model

```

model.compile(optimizer=SGD(learning_rate=lr_schedule, momentum=0.9), loss='
model.fit(X_Train,y_Train, validation_data=(X_Test, y_Test), epochs=50, bat
pred_GRU= model.predict(X_Test)
return pred_GRU

```

LSTM Model for the prediction

```

def LSTM_model(X_Train, y_Train, X_Test, y_Test):
    early_stopping = callbacks.EarlyStopping(min_delta=0.001,patience=10, restc

```

#The LSTM model

```

model = Sequential()
model.add(LSTM(units=150, return_sequences=True, input_shape=(X_Train.shape[1], X_Train.shape[2])))
model.add(Dropout(0.2))

```

```

model.add(LSTM(units=150, return_sequences=True, input_shape=(X_Train.shape[1],X_Train.shape[2])))
model.add(Dropout(0.2))
model.add(LSTM(units=50, return_sequences=True, input_shape=(X_Train.shape[1],X_Train.shape[2])))
model.add(Dropout(0.2))
model.add(LSTM(units=50, return_sequences=True, input_shape=(X_Train.shape[1],X_Train.shape[2])))
model.add(Dropout(0.2))
model.add(LSTM(units=50, input_shape=(X_Train.shape[1],1), activation='tanh'))
model.add(Dropout(0.2))
model.add(Dense(units=1))

#Compiling the model
model.compile(optimizer=SGD(learning_rate=lr_schedule, momentum=0.9),loss='binary_crossentropy')
model.fit(X_Train,y_Train, validation_data=(X_Test, y_Test), epochs=50, batch_size=32)
pred_LSTM = model.predict(X_Test)
return pred_LSTM

```

CNN Model for the prediction

```

def CNN_model(X_Train, y_Train, X_Test, y_Test):
    early_stopping = callbacks.EarlyStopping(min_delta=0.001,patience=10, restore_best_weights=True)

    #The CNN model
    model = Sequential()
    model.add(Conv1D(filters=64, kernel_size=3, activation='relu', input_shape=(X_Train.shape[1],X_Train.shape[2])))
    model.add(Conv1D(filters=64, kernel_size=3, activation='relu'))
    model.add(Dropout(0.5))
    model.add(MaxPooling1D(pool_size=2))
    model.add(Flatten())
    model.add(Dense(50, activation='relu'))
    model.add(Dense(units=1))

    #Compiling the model
    model.compile(optimizer=SGD(learning_rate=lr_schedule, momentum=0.9),loss='binary_crossentropy')
    model.fit(X_Train,y_Train, validation_data=(X_Test, y_Test), epochs=50, batch_size=32)
    pred_CNN = model.predict(X_Test)
    return pred_CNN

```

MLP Model for the prediction

```

def MLP_model(X_Train, y_Train, X_Test, y_Test):
    X_Train = np.squeeze(X_Train) # This line removes the unnecessary dimension
    X_Test = np.squeeze(X_Test) # Do the same for the test data
    early_stopping = callbacks.EarlyStopping(min_delta=0.001,patience=10, restore_best_weights=True)

    model = Sequential()
    model.add(Dense(150, activation='relu', input_dim=X_Train.shape[1]))
    model.add(Dropout(0.2))
    model.add(Dense(150, activation='relu'))
    model.add(Dropout(0.2))
    model.add(Dense(50, activation='relu'))
    model.add(Dropout(0.2))
    model.add(Dense(1))

```

```

model.add(Dense(50, activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(50, activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(units=1))
model.compile(optimizer=SGD(learning_rate=lr_schedule, momentum=0.9), loss='
model.fit(X_Train, y_Train, validation_data=(X_testJ1, y_testJ1), epochs=50
pred_MLP = model.predict(X_Test)
return pred_MLP

```

```

#To calculate the root mean squared error in predictions
def RMSE_Value(test,predicted):
    rmse = math.sqrt(mean_squared_error(test, predicted))
    print("The root mean squared error is {}".format(rmse))
    return rmse

```

```

#To plot the comparative plot of targets and predictions
def PredictionsPlot(test,predicted,m):
    plt.figure(figsize=(12,5),facecolor="#99ccff")
    plt.plot(test, color=colors[m],label="True Value",alpha=0.5 )
    plt.plot(predicted, color="#627D78",label="Predicted Values")
    plt.title("Traffic Prediction Vs True values")
    plt.xlabel("DateTime")
    plt.ylabel("Number of Vehicles")
    plt.legend()
    plt.show()

```

✓ Fitting The Model

Now, I will be fitting the transformed training sets of four junctions to the model created and compare them to the transformed test sets.

Fitting the first junction and plotting the predictions and testset

```

#Predictions For First Junction
print("\033[1;31;2m#####-----Custom Model-----#####\033[0m")
PredJ1_Custom = Custom_model(X_trainJ1,y_trainJ1,X_testJ1, y_testJ1)
print("-" * 70)

print("\033[1;31;2m#####-----GRU Model-----#####\033[0m")
PredJ1_GRU = GRU_model(X_trainJ1,y_trainJ1,X_testJ1, y_testJ1)
print("-" * 70)

print("\033[1;34;2m#####-----LSTM Model-----#####\033[0m")

```

```
PredJ1_LSTM = LSTM_model(X_trainJ1,y_trainJ1,X_testJ1, y_testJ1)
print("-" * 70)
```

```
print("\033[1;32;2m#####-----CNN Model-----#####\033[0m")
PredJ1_CNN = CNN_model(X_trainJ1,y_trainJ1,X_testJ1, y_testJ1)
print("-" * 70)
```

```
print("\033[1;33;2m#####-----MLP Model-----#####\033[0m")
PredJ1_MLP = MLP_model(X_trainJ1,y_trainJ1,X_testJ1, y_testJ1)
print("-" * 70)
```

```
#####-----Custom Model-----#####
```

```
Epoch 1/50
108/108 [=====] - 3s 5ms/step - loss: 0.0747 - val
Epoch 2/50
108/108 [=====] - 0s 4ms/step - loss: 0.0578 - val
Epoch 3/50
108/108 [=====] - 0s 4ms/step - loss: 0.0543 - val
Epoch 4/50
108/108 [=====] - 0s 4ms/step - loss: 0.0522 - val
Epoch 5/50
108/108 [=====] - 0s 4ms/step - loss: 0.0508 - val
Epoch 6/50
108/108 [=====] - 0s 4ms/step - loss: 0.0512 - val
Epoch 7/50
108/108 [=====] - 0s 4ms/step - loss: 0.0493 - val
Epoch 8/50
108/108 [=====] - 0s 4ms/step - loss: 0.0488 - val
Epoch 9/50
108/108 [=====] - 0s 3ms/step - loss: 0.0484 - val
Epoch 10/50
108/108 [=====] - 0s 4ms/step - loss: 0.0483 - val
Epoch 11/50
108/108 [=====] - 0s 4ms/step - loss: 0.0480 - val
Epoch 12/50
108/108 [=====] - 0s 3ms/step - loss: 0.0473 - val
Epoch 13/50
108/108 [=====] - 0s 4ms/step - loss: 0.0474 - val
Epoch 14/50
108/108 [=====] - 0s 3ms/step - loss: 0.0475 - val
Epoch 15/50
108/108 [=====] - 1s 5ms/step - loss: 0.0468 - val
Epoch 16/50
108/108 [=====] - 1s 5ms/step - loss: 0.0464 - val
Epoch 17/50
108/108 [=====] - 1s 5ms/step - loss: 0.0470 - val
Epoch 18/50
108/108 [=====] - 1s 6ms/step - loss: 0.0463 - val
Epoch 19/50
108/108 [=====] - 1s 6ms/step - loss: 0.0459 - val
Epoch 20/50
```

```

108/108 [=====] - 0s 4ms/step - loss: 0.0460 - val
Epoch 21/50
108/108 [=====] - 0s 4ms/step - loss: 0.0454 - val
Epoch 22/50
108/108 [=====] - 0s 4ms/step - loss: 0.0458 - val
Epoch 23/50
108/108 [=====] - 0s 3ms/step - loss: 0.0461 - val
Epoch 24/50
108/108 [=====] - 0s 4ms/step - loss: 0.0456 - val
Epoch 25/50
108/108 [=====] - 0s 3ms/step - loss: 0.0455 - val
Epoch 26/50
108/108 [=====] - 0s 4ms/step - loss: 0.0457 - val
Epoch 27/50
108/108 [=====] - 0s 4ms/step - loss: 0.0449 - val
Epoch 28/50
108/108 [=====] - 0s 4ms/step - loss: 0.0458 - val
45/45 [=====] - 0s 3ms/step
-----

```

Results for J1 - Custom Model

```

print("\033[1;31;2m#####-----Custom Model-----#####\033[0m")
RMSE_J1_Custom = RMSE_Value(y_testJ1, PredJ1_Custom)
PredictionsPlot(y_testJ1, PredJ1_Custom, 0)

```

Results for J1 - GRU Model

```

print("\033[1;31;2m#####-----GRU Model-----#####\033[0m")
RMSE_J1_GRU = RMSE_Value(y_testJ1, PredJ1_GRU)
PredictionsPlot(y_testJ1, PredJ1_GRU, 0)

```

Results for J1 - LSTM Model

```

print("\n\033[1;34;2m#####-----LSTM Model-----#####\033[0m")
RMSE_J1_LSTM = RMSE_Value(y_testJ1, PredJ1_LSTM)
PredictionsPlot(y_testJ1, PredJ1_LSTM, 0)

```

Results for J1 - CNN Model

```

print("\n\033[1;32;2m#####-----CNN Model-----#####\033[0m")
RMSE_J1_CNN = RMSE_Value(y_testJ1, PredJ1_CNN)
PredictionsPlot(y_testJ1, PredJ1_CNN, 0)

```

Results for J1 - MLP Model

```

print("\n\033[1;33;2m#####-----MLP Model-----#####\033[0m")
RMSE_J1_MLP = RMSE_Value(y_testJ1, PredJ1_MLP)
PredictionsPlot(y_testJ1, PredJ1_MLP, 0)

```

Create a list of model names and their corresponding RMSE values

```

model_names = ["Custom", "GRU", "LSTM", "CNN", "MLP"]
rmse_values = [RMSE_J1_Custom, RMSE_J1_GRU, RMSE_J1_LSTM, RMSE_J1_CNN, RMSE_J1_

model_rmse = list(zip(model_names, rmse_values))

```

```

Results_df = pd.DataFrame(model_rmse, columns=["MODEL", "RMSE"])
styled_df = Results_df.style.background_gradient(cmap="cool")

# Find the best model with the minimum RMSE value
best_model_index = rmse_values.index(min(rmse_values))
best_model_name_1 = model_names[best_model_index]
best_model_rmse_1 = rmse_values[best_model_index]

# Print the best model name and its RMSE value
print("\n\033[1;31;4mBest Model: {} - RMSE: {}\n".format(best_model_name_1, best_model_rmse_1))

# Plot the bar graph for model names and RMSE values with transparency and light colors
colors = ['lightcoral', 'lightblue', 'lightgreen', 'lightsalmon', 'lightyellow']
alpha = 0.9
fig, ax = plt.subplots(figsize=(10, 4)) # Set the figure size (width, height)
ax.bar(model_names, rmse_values, color=colors, alpha=alpha)
ax.set_xlabel('Model')
ax.set_ylabel('RMSE Value')
ax.set_title('RMSE Value for Different Models - J1')
plt.show()

display(styled_df)

```

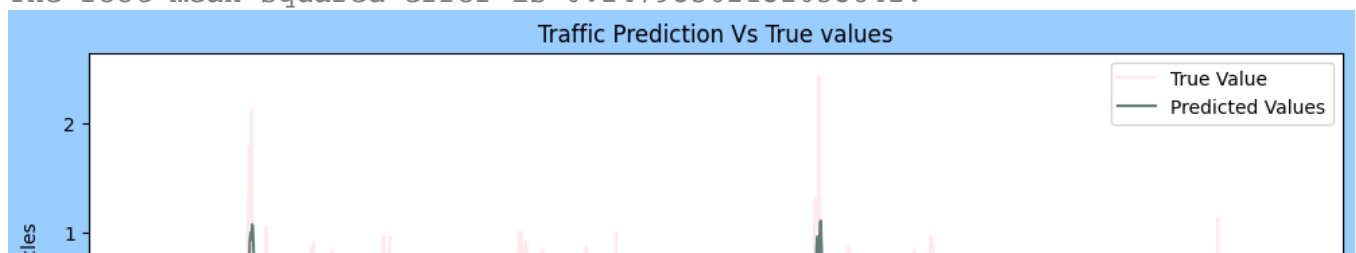
#####-----Custom Model-----#####

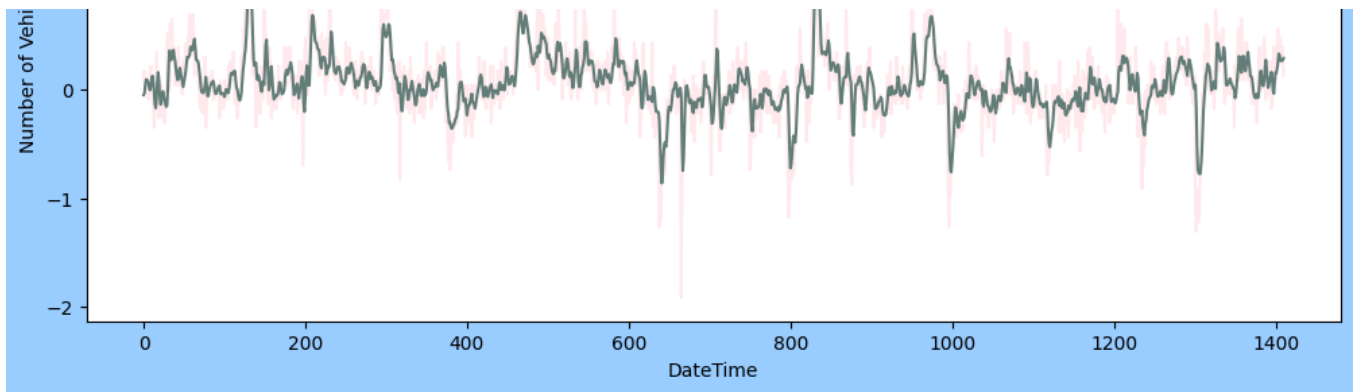
The root mean squared error is 0.24033386260972242.



#####-----GRU Model-----#####

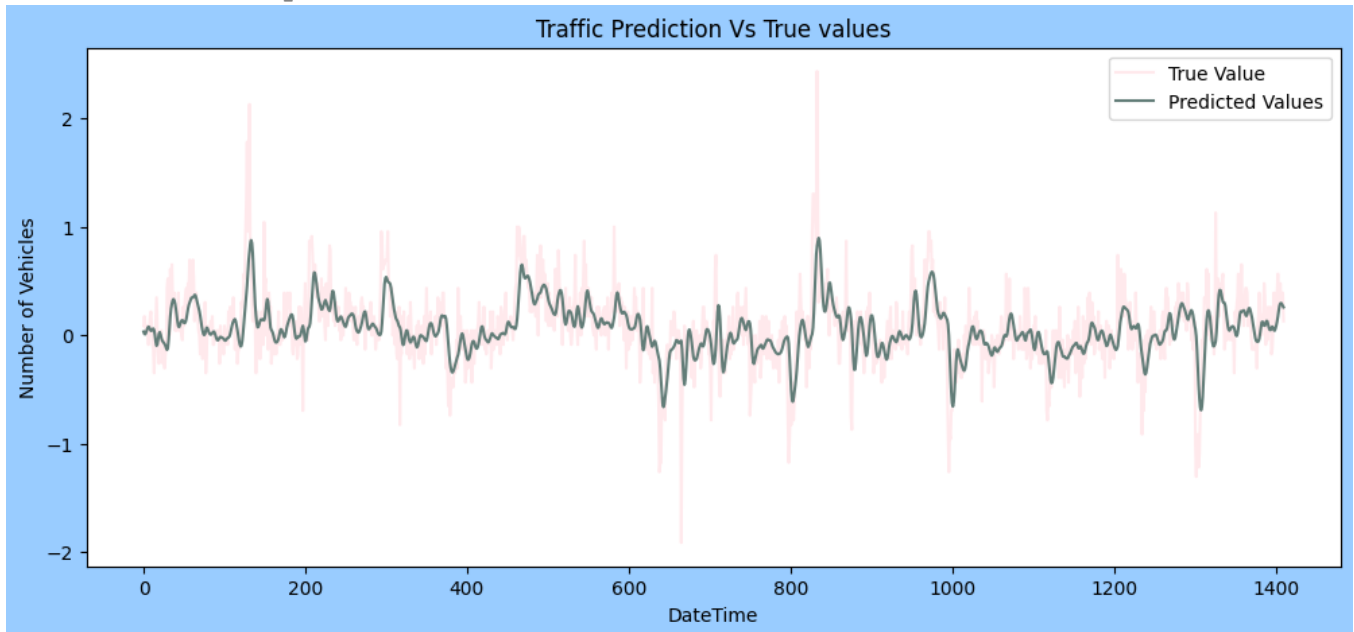
The root mean squared error is 0.24793562132053842.





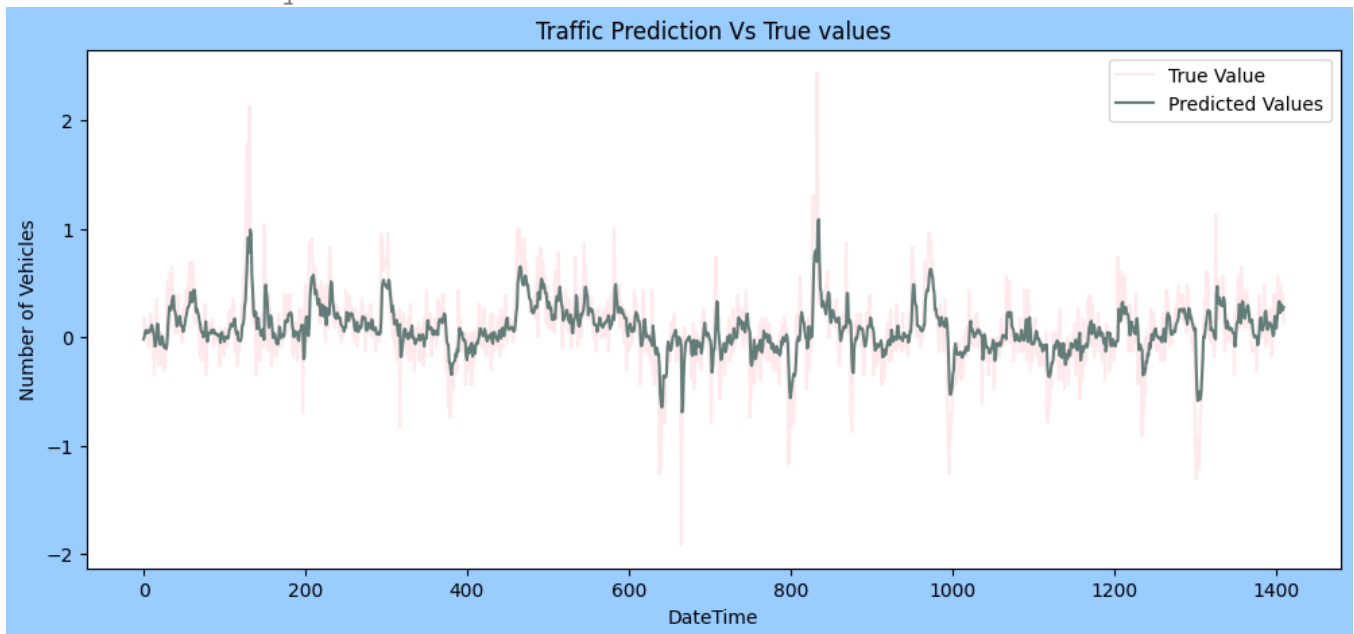
#####-----LSTM Model-----#####

The root mean squared error is 0.27238365102223266.



#####-----CNN Model-----#####

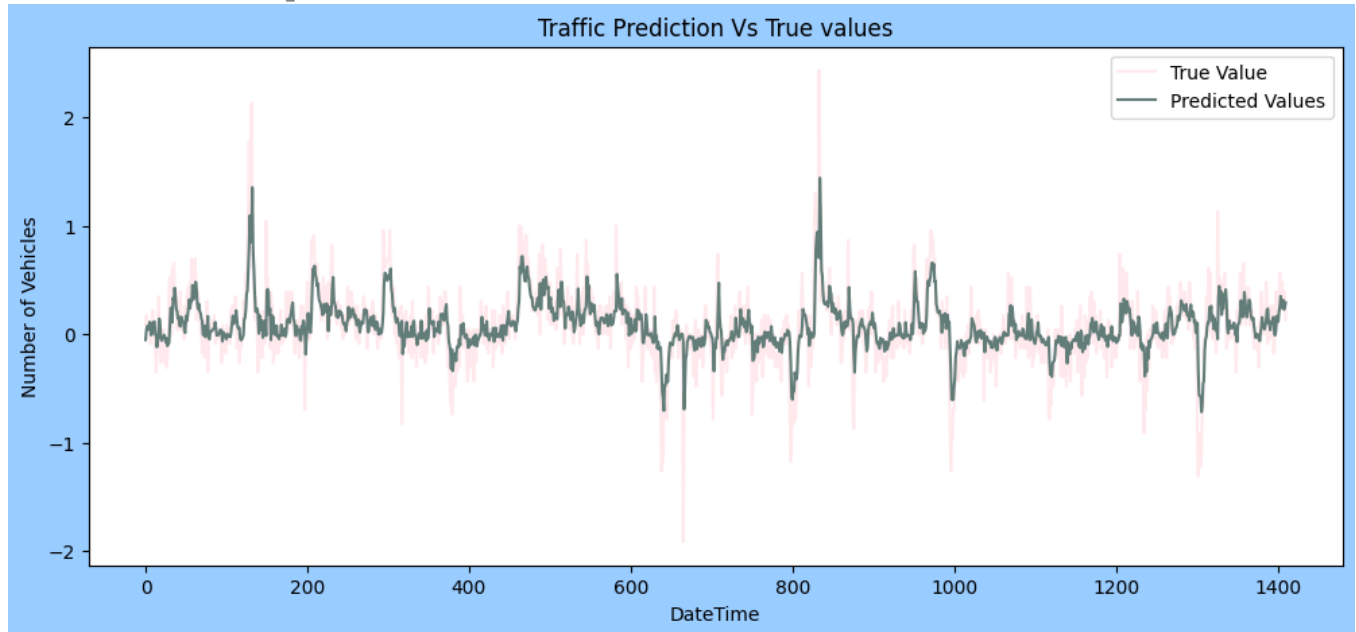
The root mean squared error is 0.24593204961517137.



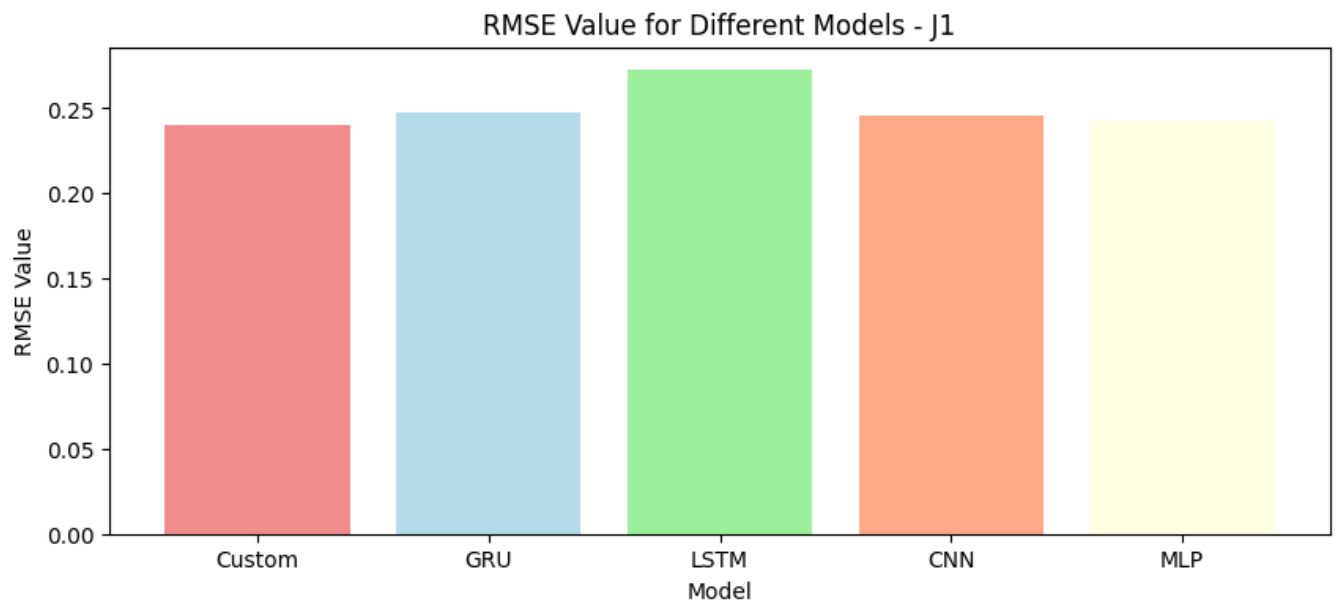
#####-----MLP Model-----#####

The root mean squared error is 0.24301931085575879

THE 1000 MEAN SQUARED ERROR IS 0.24301931003375075.



Best Model: Custom - RMSE: 0.24033386260972242



	MODEL	RMSE
0	Custom	0.240334
1	GRU	0.247936
2	LSTM	0.272384
3	CNN	0.245932
4	MLP	0.243019


```
# Prediction For Second Function
```

```
# Predictions for Second Junction
print("\033[1;31;2m#####-----Custom Model-----#####\033[0m")
PredJ2_Custom = Custom_model(X_trainJ2, y_trainJ2, X_testJ2, y_testJ2)
print("-" * 70)
print("\033[1;31;2m#####-----GRU Model-----#####\033[0m")
PredJ2_GRU = GRU_model(X_trainJ2, y_trainJ2, X_testJ2, y_testJ2)
print("-" * 70)
print("\033[1;34;2m#####-----LSTM Model-----#####\033[0m")
PredJ2_LSTM = LSTM_model(X_trainJ2, y_trainJ2, X_testJ2, y_testJ2)
print("-" * 70)
print("\033[1;32;2m#####-----CNN Model-----#####\033[0m")
PredJ2_CNN = CNN_model(X_trainJ2, y_trainJ2, X_testJ2, y_testJ2)
print("-" * 70)
print("\033[1;33;2m#####-----MLP Model-----#####\033[0m")
PredJ2_MLP = MLP_model(X_trainJ2, y_trainJ2, X_testJ2, y_testJ2)
print("-" * 70)
```

#####-----Custom Model-----#####

```
Epoch 1/50
109/109 [=====] - 1s 5ms/step - loss: 0.2104 - val
Epoch 2/50
109/109 [=====] - 0s 4ms/step - loss: 0.1675 - val
Epoch 3/50
109/109 [=====] - 0s 4ms/step - loss: 0.1592 - val
Epoch 4/50
109/109 [=====] - 0s 4ms/step - loss: 0.1553 - val
Epoch 5/50
109/109 [=====] - 0s 4ms/step - loss: 0.1517 - val
Epoch 6/50
109/109 [=====] - 0s 4ms/step - loss: 0.1469 - val
Epoch 7/50
109/109 [=====] - 1s 6ms/step - loss: 0.1456 - val
Epoch 8/50
109/109 [=====] - 1s 5ms/step - loss: 0.1448 - val
Epoch 9/50
109/109 [=====] - 1s 6ms/step - loss: 0.1447 - val
Epoch 10/50
109/109 [=====] - 1s 6ms/step - loss: 0.1429 - val
Epoch 11/50
109/109 [=====] - 1s 5ms/step - loss: 0.1421 - val
Epoch 12/50
109/109 [=====] - 0s 4ms/step - loss: 0.1421 - val
Epoch 13/50
109/109 [=====] - 0s 4ms/step - loss: 0.1410 - val
Epoch 14/50
109/109 [=====] - 0s 4ms/step - loss: 0.1404 - val
Epoch 15/50
109/109 [=====] - 0s 4ms/step - loss: 0.1399 - val
Epoch 16/50
109/109 [=====] - 0s 4ms/step - loss: 0.1398 - val
Epoch 17/50
109/109 [=====] - 0s 4ms/step - loss: 0.1385 - val
```

```
Epoch 18/50
109/109 [=====] - 0s 4ms/step - loss: 0.1403 - val
Epoch 19/50
109/109 [=====] - 1s 5ms/step - loss: 0.1383 - val
45/45 [=====] - 0s 3ms/step
```

```
#####-----GRU Model-----#####
```

```
Epoch 1/50
109/109 [=====] - 10s 27ms/step - loss: 0.2319 - v
Epoch 2/50
109/109 [=====] - 1s 13ms/step - loss: 0.2016 - va
Epoch 3/50
109/109 [=====] - 1s 13ms/step - loss: 0.1964 - va
Epoch 4/50
109/109 [=====] - 1s 13ms/step - loss: 0.1950 - va
Epoch 5/50
109/109 [=====] - 1s 14ms/step - loss: 0.1942 - va
Epoch 6/50
109/109 [=====] - 2s 15ms/step - loss: 0.1907 - va
Epoch 7/50
109/109 [=====] - 2s 18ms/step - loss: 0.1894 - va
Epoch 8/50
109/109 [=====] - 1s 13ms/step - loss: 0.1892 - va
Epoch 9/50
```

```
# Results for J2 - Custom Model
```

```
print("\033[1;31;2m#####-----Custom Model-----#####\033[0m")
RMSE_J2_Custom = RMSE_Value(y_testJ2, PredJ2_Custom)
PredictionsPlot(y_testJ2, PredJ2_Custom, 0)
```

```
# Results for J2 - GRU Model
```

```
print("\033[1;31;2m#####-----GRU Model-----#####\033[0m")
RMSE_J2_GRU = RMSE_Value(y_testJ2, PredJ2_GRU)
PredictionsPlot(y_testJ2, PredJ2_GRU, 0)
```

```
# Results for J2 - LSTM Model
```

```
print("\n\033[1;34;2m#####-----LSTM Model-----#####\033[0m")
RMSE_J2_LSTM = RMSE_Value(y_testJ2, PredJ2_LSTM)
PredictionsPlot(y_testJ2, PredJ2_LSTM, 0)
```

```
# Results for J2 - CNN Model
```

```
print("\n\033[1;32;2m#####-----CNN Model-----#####\033[0m")
RMSE_J2_CNN = RMSE_Value(y_testJ2, PredJ2_CNN)
PredictionsPlot(y_testJ2, PredJ2_CNN, 0)
```

```
# Results for J2 - MLP Model
```

```
print("\n\033[1;33;2m#####-----MLP Model-----#####\033[0m")
RMSE_J2_MLP = RMSE_Value(y_testJ2, PredJ2_MLP)
PredictionsPlot(y_testJ2, PredJ2_MLP, 0)
```

```
# Create a list of model names and their corresponding RMSE values
model_names = ["Custom", "GRU", "LSTM", "CNN", "MLP"]
rmse_values = [RMSE_J2_Custom, RMSE_J2_GRU, RMSE_J2_LSTM, RMSE_J2_CNN, RMSE_J2_MLP]

model_rmse = list(zip(model_names, rmse_values))
Results_df = pd.DataFrame(model_rmse, columns=["MODEL", "RMSE"])
styled_df = Results_df.style.background_gradient(cmap="cool")

# Find the best model with the minimum RMSE value
best_model_index = rmse_values.index(min(rmse_values))
best_model_name_2 = model_names[best_model_index]
best_model_rmse_2 = rmse_values[best_model_index]

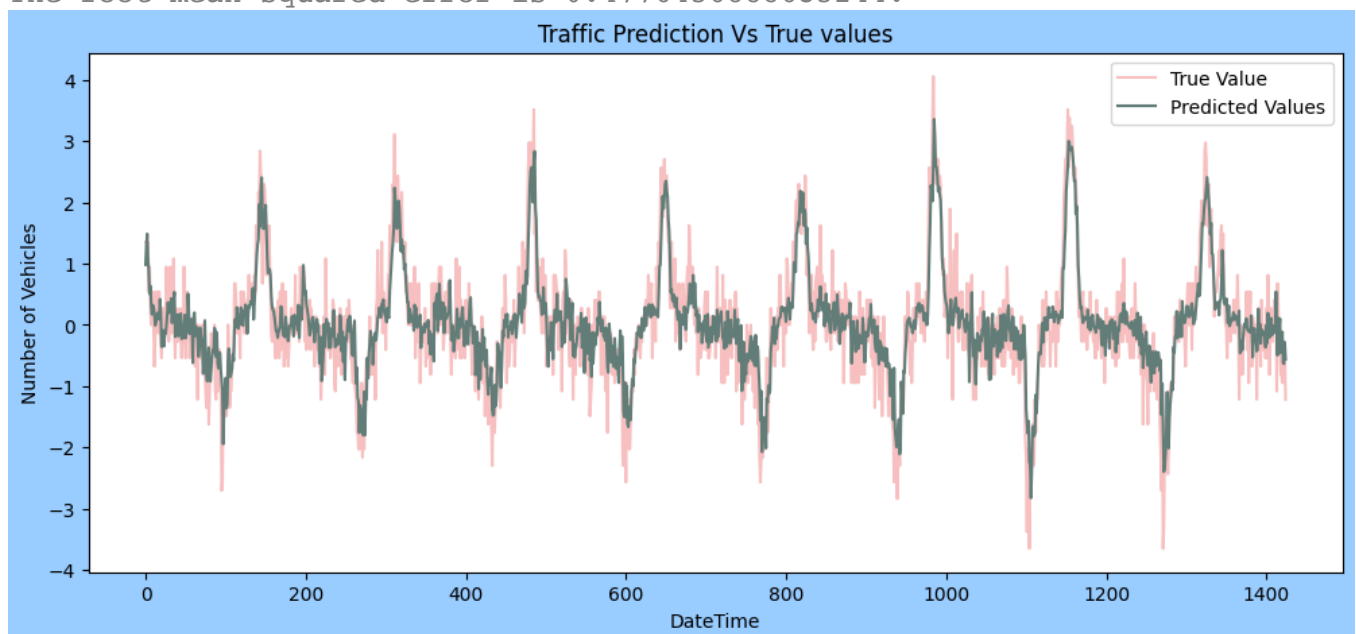
# Print the best model name and its RMSE value
print("\n\033[1;31;4mBest Model: {} - RMSE: {}\n".format(best_model_name_2, best_model_rmse_2))

# Plot the bar graph for model names and RMSE values with transparency and light colors
colors = ['lightcoral', 'lightblue', 'lightgreen', 'lightsalmon', 'lightyellow']
alpha = 0.7
fig, ax = plt.subplots(figsize=(12, 8)) # Set the figure size (width, height)
ax.bar(model_names, rmse_values, color=colors, alpha=alpha)
ax.set_xlabel('Model')
ax.set_ylabel('RMSE Value')
ax.set_title('RMSE Value for Different Models - J2')
plt.show()

display(styled_df)
```

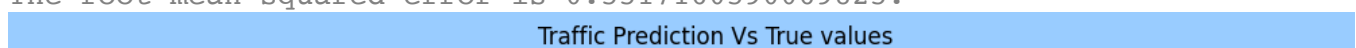
#####-----Custom Model-----#####

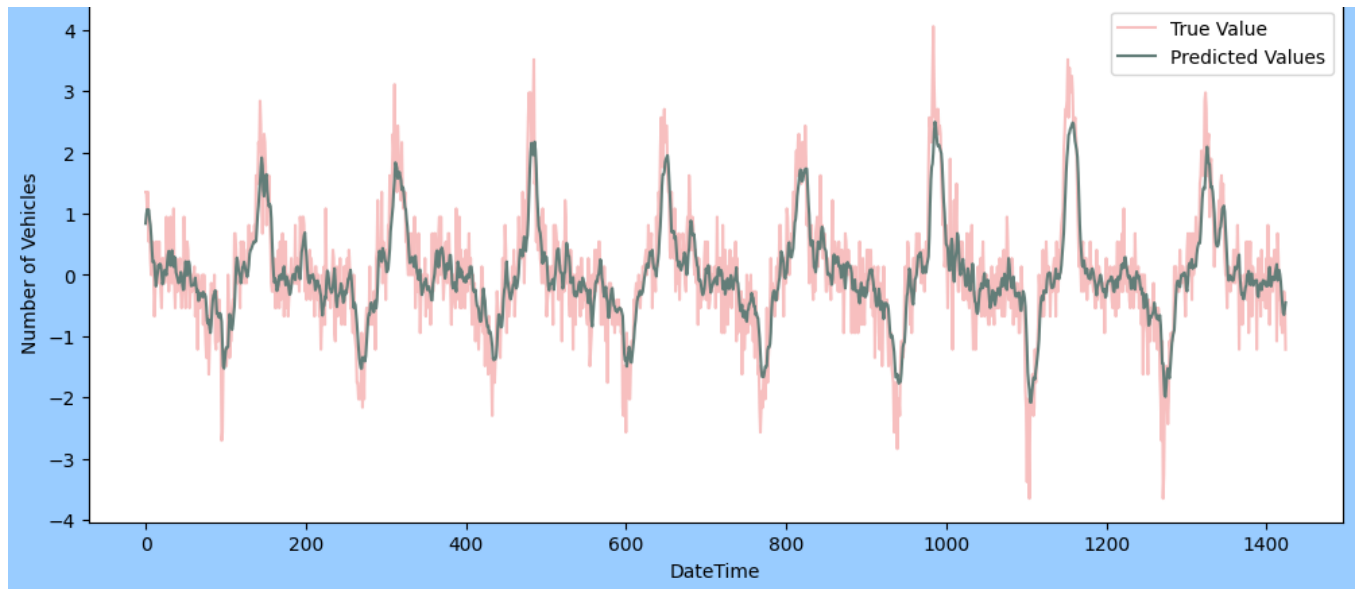
The root mean squared error is 0.4776456888635244.



#####-----GRU Model-----#####

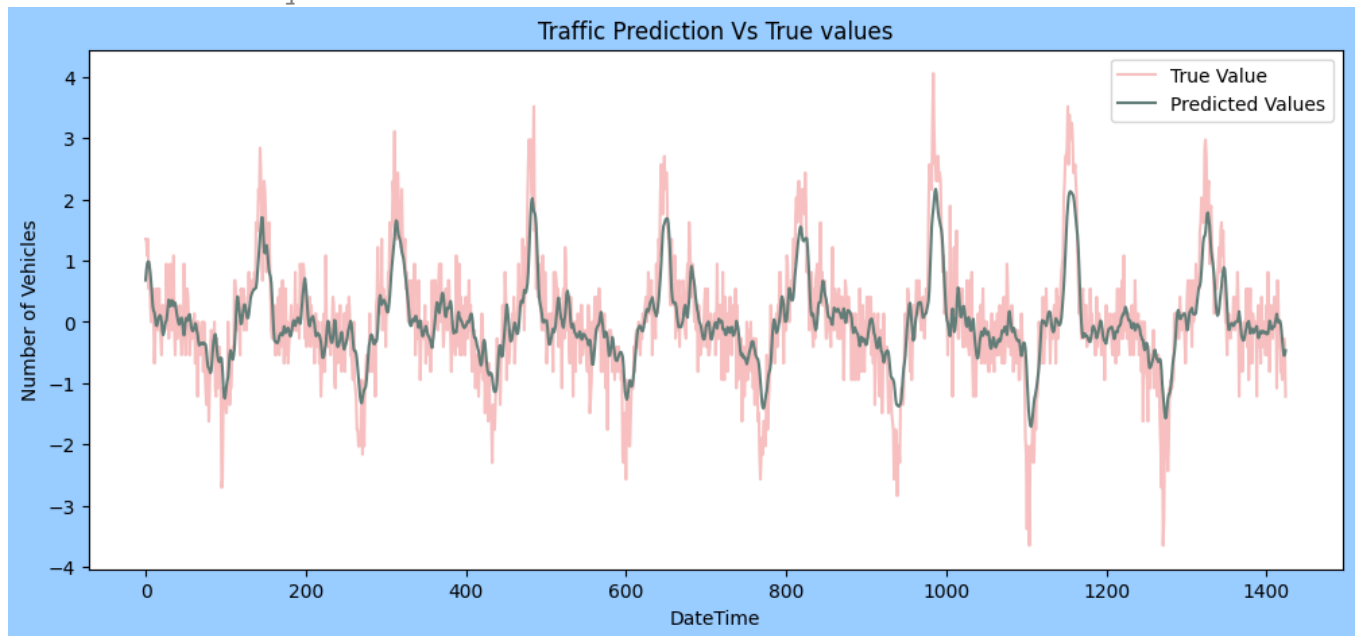
The root mean squared error is 0.5517160590069825.





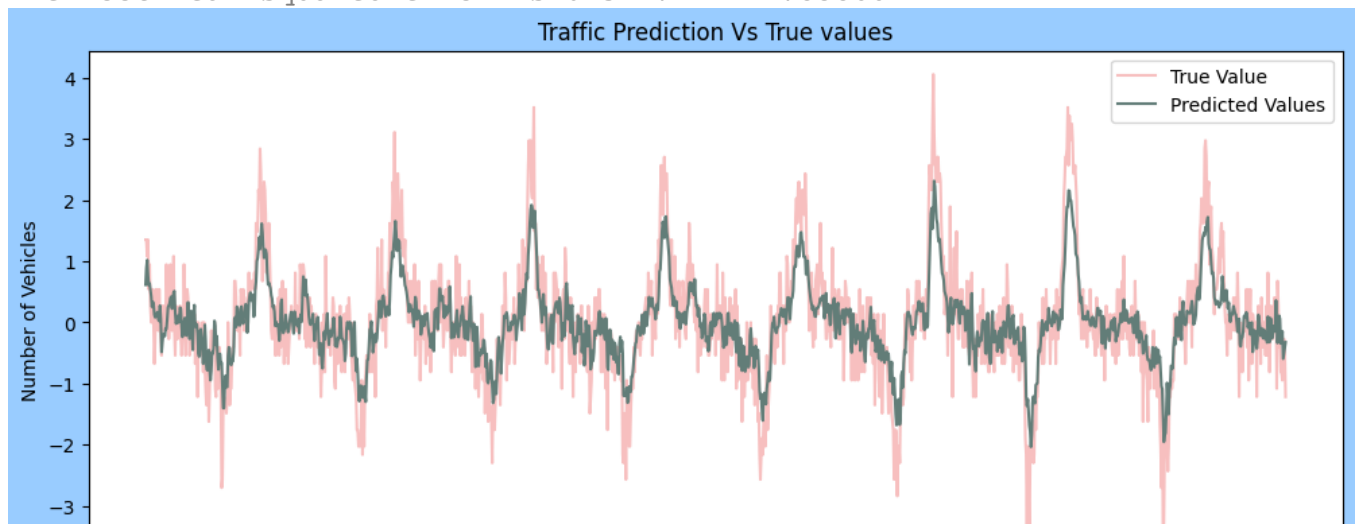
#####-----LSTM Model-----#####

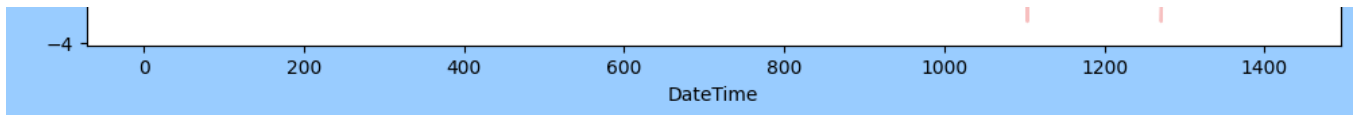
The root mean squared error is 0.5792634231672263.



#####-----CNN Model-----#####

The root mean squared error is 0.5427212212783866.



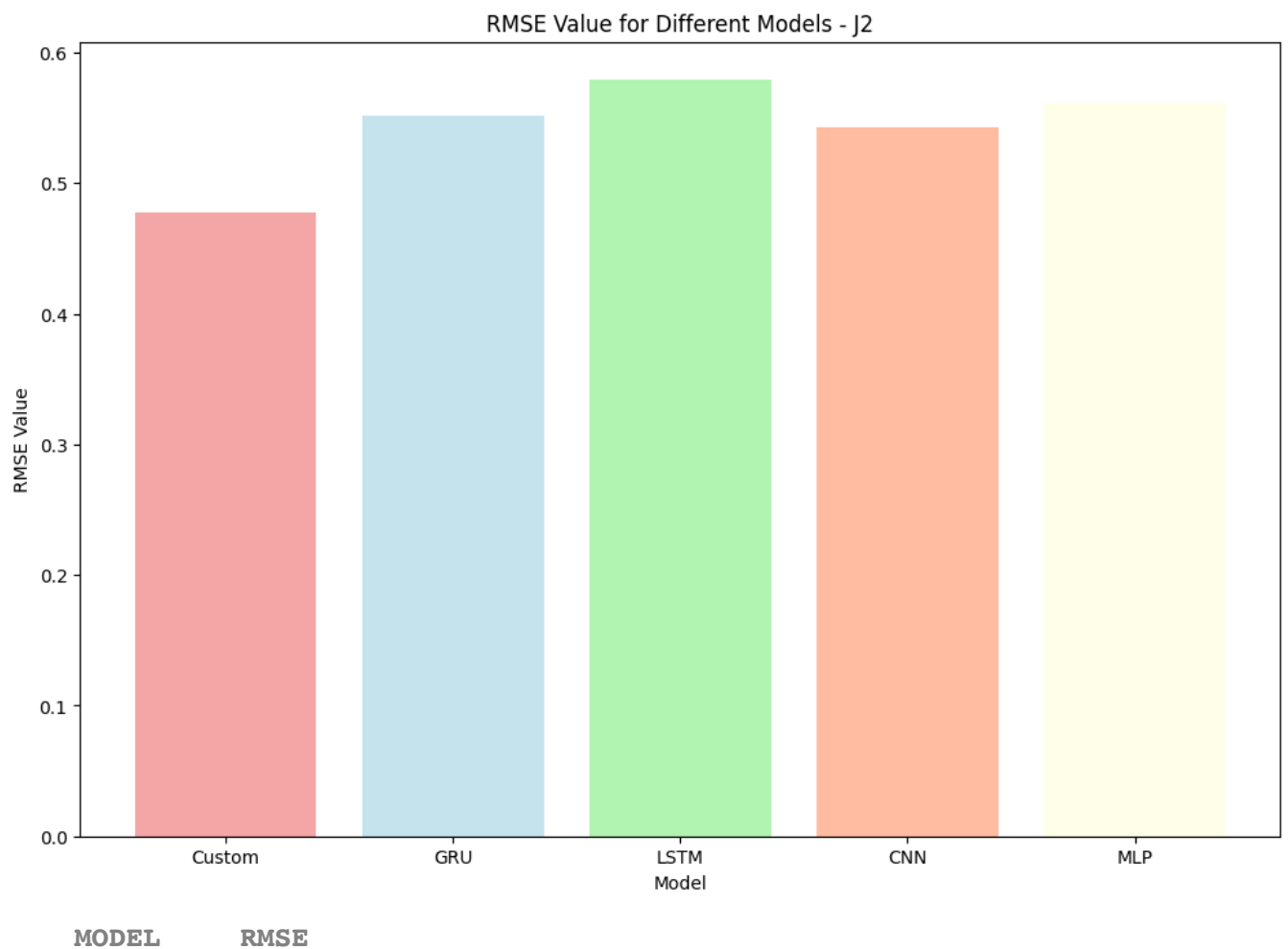


#####-----MLP Model-----#####

The root mean squared error is 0.5612801490671254.



Best Model: Custom - RMSE: 0.4776456888635244



0	Custom	0.477646
1	GRU	0.551716
2	LSTM	0.579263
3	CNN	0.542721
4	MLP	0.561280

```

# Predictions For Third Junction
# Predictions For Third Junction
print("\033[1;31;2m#####-----Custom Model-----#####\033[0m")
PredJ3_Custom = Custom_model(X_trainJ3, y_trainJ3, X_testJ3, y_testJ3)
print("-" * 70)
print("\033[1;31;2m#####-----GRU Model-----#####\033[0m")
PredJ3_GRU = GRU_model(X_trainJ3, y_trainJ3, X_testJ3, y_testJ3)
print("-" * 70)
print("\033[1;34;2m#####-----LSTM Model-----#####\033[0m")
PredJ3_LSTM = LSTM_model(X_trainJ3, y_trainJ3, X_testJ3, y_testJ3)
print("-" * 70)
print("\033[1;32;2m#####-----CNN Model-----#####\033[0m")
PredJ3_CNN = CNN_model(X_trainJ3, y_trainJ3, X_testJ3, y_testJ3)
print("-" * 70)
print("\033[1;33;2m#####-----MLP Model-----#####\033[0m")
PredJ3_MLP = MLP_model(X_trainJ3, y_trainJ3, X_testJ3, y_testJ3)
print("-" * 70)

```

#####-----Custom Model-----#####

Epoch 1/50

110/110 [=====] - 2s 7ms/step - loss: 0.3025 - val

Epoch 2/50

110/110 [=====] - 1s 5ms/step - loss: 0.2820 - val

Epoch 3/50

110/110 [=====] - 1s 5ms/step - loss: 0.2679 - val

Epoch 4/50

110/110 [=====] - 1s 5ms/step - loss: 0.2606 - val

Epoch 5/50


```
110/110 [=====] - 0s 4ms/step - loss: 0.2588 - val
Epoch 6/50
110/110 [=====] - 0s 4ms/step - loss: 0.2539 - val
Epoch 7/50
110/110 [=====] - 0s 4ms/step - loss: 0.2489 - val
Epoch 8/50
110/110 [=====] - 0s 4ms/step - loss: 0.2455 - val
Epoch 9/50
110/110 [=====] - 0s 4ms/step - loss: 0.2436 - val
Epoch 10/50
110/110 [=====] - 0s 4ms/step - loss: 0.2407 - val
Epoch 11/50
110/110 [=====] - 0s 3ms/step - loss: 0.2403 - val
Epoch 12/50
110/110 [=====] - 0s 4ms/step - loss: 0.2362 - val
Epoch 13/50
110/110 [=====] - 0s 4ms/step - loss: 0.2341 - val
Epoch 14/50
110/110 [=====] - 0s 4ms/step - loss: 0.2351 - val
Epoch 15/50
110/110 [=====] - 0s 4ms/step - loss: 0.2350 - val
Epoch 16/50
110/110 [=====] - 0s 4ms/step - loss: 0.2323 - val
Epoch 17/50
110/110 [=====] - 0s 4ms/step - loss: 0.2267 - val
Epoch 18/50
110/110 [=====] - 0s 4ms/step - loss: 0.2288 - val
Epoch 19/50
110/110 [=====] - 0s 4ms/step - loss: 0.2293 - val
Epoch 20/50
110/110 [=====] - 0s 4ms/step - loss: 0.2331 - val
Epoch 21/50
110/110 [=====] - 0s 4ms/step - loss: 0.2227 - val
Epoch 22/50
110/110 [=====] - 0s 4ms/step - loss: 0.2249 - val
Epoch 23/50
110/110 [=====] - 0s 3ms/step - loss: 0.2210 - val
Epoch 24/50
110/110 [=====] - 0s 4ms/step - loss: 0.2178 - val
Epoch 25/50
110/110 [=====] - 0s 3ms/step - loss: 0.2181 - val
Epoch 26/50
110/110 [=====] - 0s 3ms/step - loss: 0.2163 - val
Epoch 27/50
110/110 [=====] - 0s 4ms/step - loss: 0.2199 - val
Epoch 28/50
110/110 [=====] - 0s 3ms/step - loss: 0.2162 - val
Epoch 29/50
110/110 [=====] - 0s 4ms/step - loss: 0.2153 - val
```

Results for J3 - Custom Model

```
print("\033[1;31;2m#####-----Custom Model-----#####\033[0m")
RMSE_J3_Custom = RMSE_Value(y_testJ3, PredJ3_Custom)
```

```

PredictionsPlot(y_testJ3, PredJ3_Custom, 0)

# Results for J3 - GRU Model
print("\n\033[1;31;2m#####-----GRU Model-----#####\033[0m")
RMSE_J3_GRU = RMSE_Value(y_testJ3, PredJ3_GRU)
PredictionsPlot(y_testJ3, PredJ3_GRU, 0)

# Results for J3 - LSTM Model
print("\n\033[1;34;2m#####-----LSTM Model-----#####\033[0m")
RMSE_J3_LSTM = RMSE_Value(y_testJ3, PredJ3_LSTM)
PredictionsPlot(y_testJ3, PredJ3_LSTM, 0)

# Results for J3 - CNN Model
print("\n\033[1;32;2m#####-----CNN Model-----#####\033[0m")
RMSE_J3_CNN = RMSE_Value(y_testJ3, PredJ3_CNN)
PredictionsPlot(y_testJ3, PredJ3_CNN, 0)

# Results for J3 - MLP Model
print("\n\033[1;33;2m#####-----MLP Model-----#####\033[0m")
RMSE_J3_MLP = RMSE_Value(y_testJ3, PredJ3_MLP)
PredictionsPlot(y_testJ3, PredJ3_MLP, 0)

# Create a list of model names and their corresponding RMSE values
model_names = ["Custom", "GRU", "LSTM", "CNN", "MLP"]
rmse_values = [RMSE_J3_Custom, RMSE_J3_GRU, RMSE_J3_LSTM, RMSE_J3_CNN, RMSE_J3_MLP]

model_rmse = list(zip(model_names, rmse_values))
Results_df = pd.DataFrame(model_rmse, columns=["MODEL", "RMSE"])
styled_df = Results_df.style.background_gradient(cmap="cool")

# Find the best model with the minimum RMSE value
best_model_index = rmse_values.index(min(rmse_values))
best_model_name_3 = model_names[best_model_index]
best_model_rmse_3 = rmse_values[best_model_index]

# Print the best model name and its RMSE value
print("\n\033[1;31;4mBest Model: {} - RMSE: {}\n".format(best_model_name_3, best_model_rmse_3))

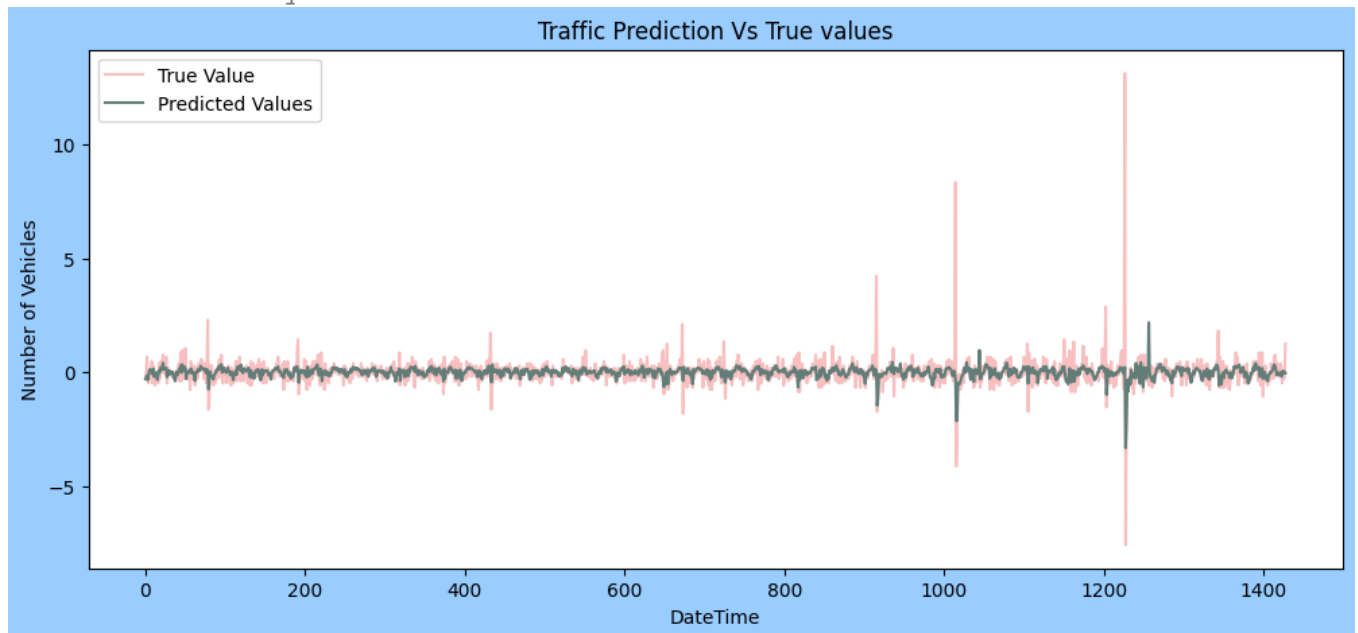
# Plot the bar graph for model names and RMSE values with transparency and light colors
colors = ['lightcoral', 'lightblue', 'lightgreen', 'lightsalmon', 'lightyellow']
alpha = 0.7
fig, ax = plt.subplots(figsize=(12, 8)) # Set the figure size (width, height)
ax.bar(model_names, rmse_values, color=colors, alpha=alpha)
ax.set_xlabel('Model')
ax.set_ylabel('RMSE Value')
ax.set_title('RMSE Value for Different Models - J3')
plt.show()

```

```
display(styled_df)
```

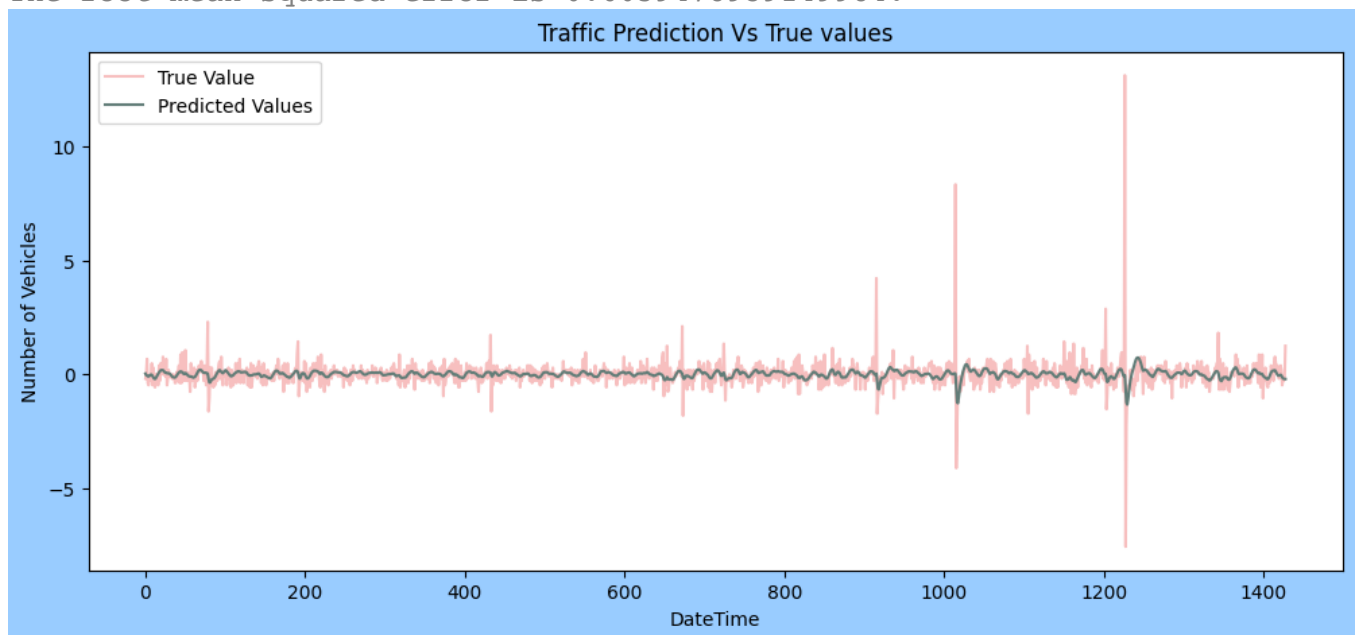
```
#####-----Custom Model-----#####
```

The root mean squared error is 0.5655572705684435.



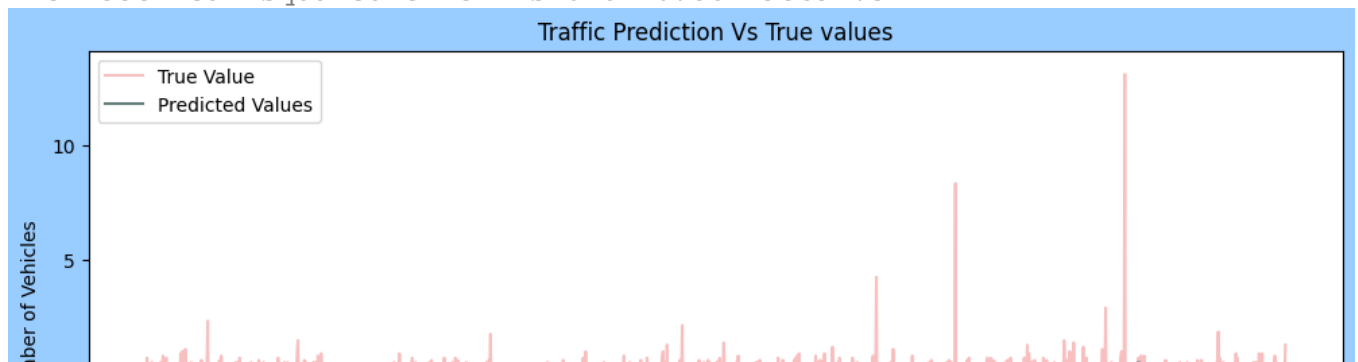
```
#####-----GRU Model-----#####
```

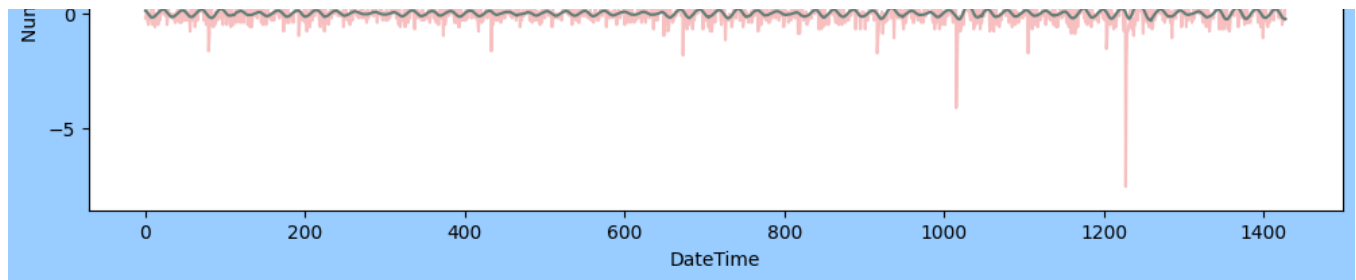
The root mean squared error is 0.6059478939149984.



```
#####-----LSTM Model-----#####
```

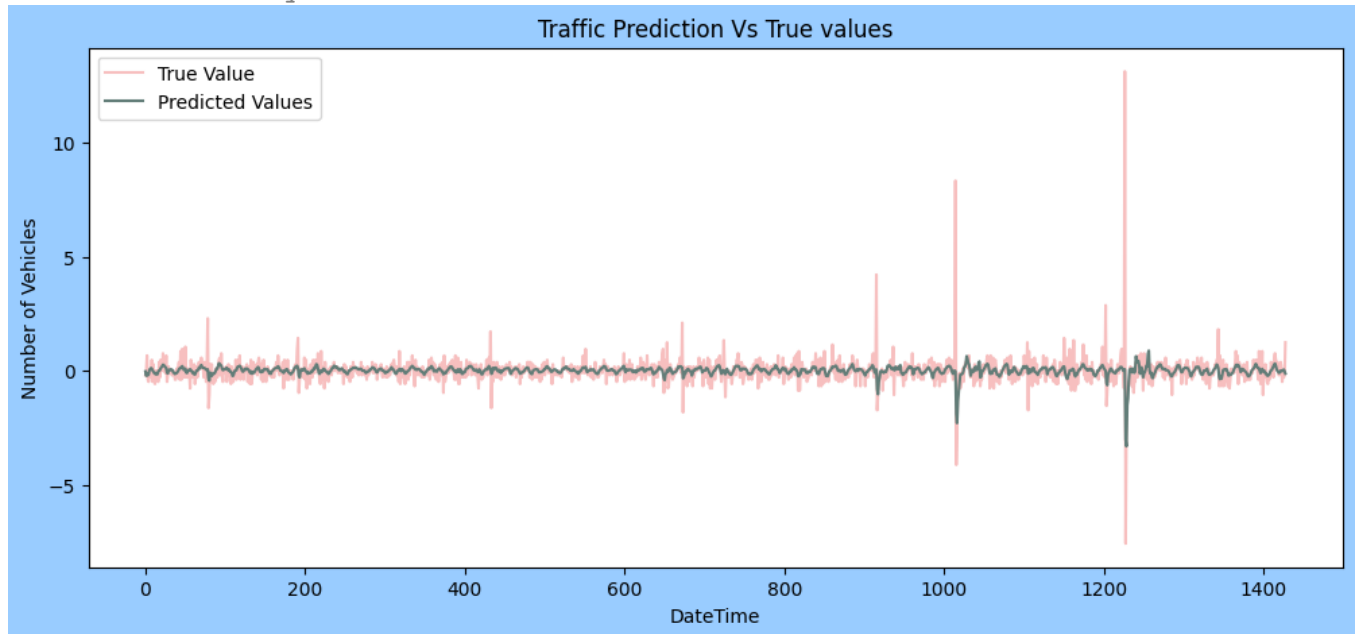
The root mean squared error is 0.6210786449389175.





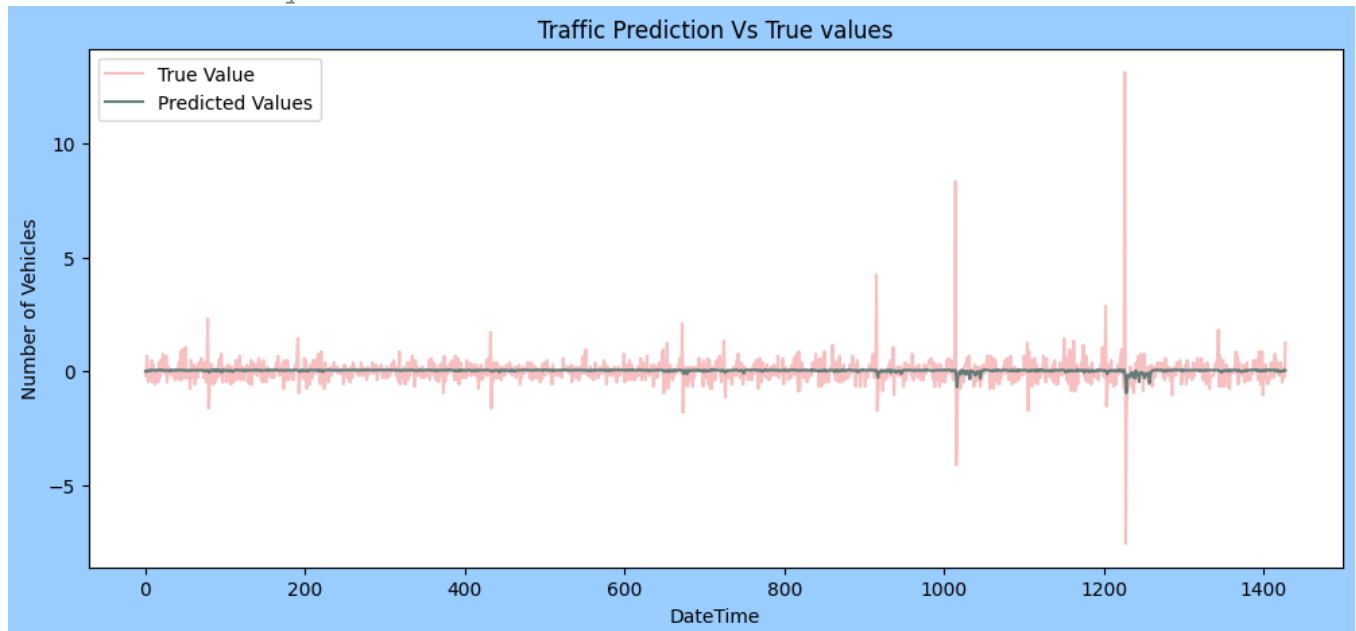
#####-----CNN Model-----#####

The root mean squared error is 0.5781111817022215.



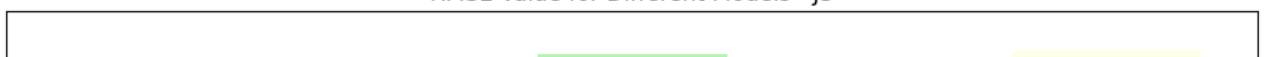
#####-----MLP Model-----#####

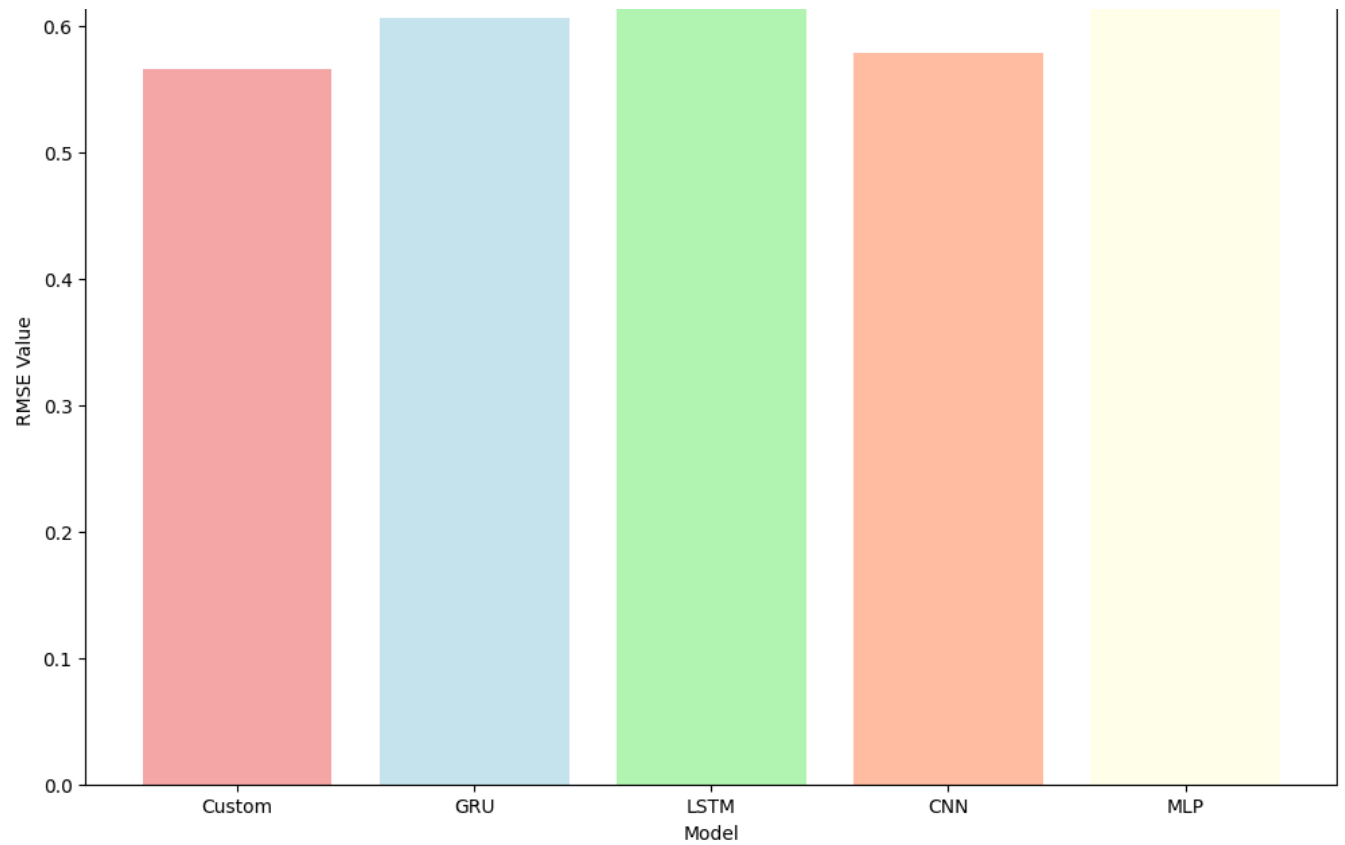
The root mean squared error is 0.624077810361193.



Best Model: Custom - RMSE: 0.565557270568435

RMSE Value for Different Models - J3





	MODEL	RMSE
0	Custom	0.565557
1	GRU	0.605948
2	LSTM	0.621079
3	CNN	0.578111
4	MLP	0.624078

```
# Predictions For Fourth Junction
print("\033[1;31;2m#####-----Custom Model-----#####\033[0m")
PredJ4_Custom = Custom_model(X_trainJ4, y_trainJ4, X_testJ4, y_testJ4)
print("-" * 70)
print("\033[1;31;2m#####-----GRU Model-----#####\033[0m")
```

```

PredJ4_GRU = GRU_model(X_trainJ4, y_trainJ4, X_testJ4, y_testJ4)
print("-" * 70)
print("\033[1;34;2m#####-----LSTM Model-----#####\033[0m")
PredJ4_LSTM = LSTM_model(X_trainJ4, y_trainJ4, X_testJ4, y_testJ4)
print("-" * 70)
print("\033[1;32;2m#####-----CNN Model-----#####\033[0m")
PredJ4_CNN = CNN_model(X_trainJ4, y_trainJ4, X_testJ4, y_testJ4)
print("-" * 70)
print("\033[1;33;2m#####-----MLP Model-----#####\033[0m")
PredJ4_MLP = MLP_model(X_trainJ4, y_trainJ4, X_testJ4, y_testJ4)
print("-" * 70)

```

#####-----Custom Model-----#####

```

Epoch 1/50
33/33 [=====] - 2s 13ms/step - loss: 0.6910 - val_
Epoch 2/50
33/33 [=====] - 0s 6ms/step - loss: 0.6371 - val_l
Epoch 3/50
33/33 [=====] - 0s 6ms/step - loss: 0.5892 - val_l
Epoch 4/50
33/33 [=====] - 0s 6ms/step - loss: 0.5696 - val_l
Epoch 5/50
33/33 [=====] - 0s 4ms/step - loss: 0.5581 - val_l
Epoch 6/50
33/33 [=====] - 0s 5ms/step - loss: 0.5304 - val_l
Epoch 7/50
33/33 [=====] - 0s 4ms/step - loss: 0.5245 - val_l
Epoch 8/50
33/33 [=====] - 0s 4ms/step - loss: 0.5119 - val_l
Epoch 9/50
33/33 [=====] - 0s 4ms/step - loss: 0.5111 - val_l
Epoch 10/50
33/33 [=====] - 0s 4ms/step - loss: 0.5066 - val_l
Epoch 11/50
33/33 [=====] - 0s 4ms/step - loss: 0.4929 - val_l
Epoch 12/50
33/33 [=====] - 0s 5ms/step - loss: 0.4905 - val_l
Epoch 13/50
33/33 [=====] - 0s 5ms/step - loss: 0.4845 - val_l
Epoch 14/50
33/33 [=====] - 0s 4ms/step - loss: 0.4783 - val_l
Epoch 15/50
33/33 [=====] - 0s 4ms/step - loss: 0.4722 - val_l
Epoch 16/50
33/33 [=====] - 0s 5ms/step - loss: 0.4627 - val_l
Epoch 17/50
33/33 [=====] - 0s 5ms/step - loss: 0.4604 - val_l
Epoch 18/50
33/33 [=====] - 0s 4ms/step - loss: 0.4637 - val_l
Epoch 19/50
33/33 [=====] - 0s 4ms/step - loss: 0.4586 - val_l
Epoch 20/50

```

```

33/33 [=====] - 0s 5ms/step - loss: 0.4518 - val_l
Epoch 21/50
33/33 [=====] - 0s 4ms/step - loss: 0.4434 - val_l
Epoch 22/50
33/33 [=====] - 0s 5ms/step - loss: 0.4428 - val_l
13/13 [=====] - 0s 2ms/step

```

#####-----GRU Model-----#####

```

Epoch 1/50
33/33 [=====] - 9s 67ms/step - loss: 0.6901 - val_
Epoch 2/50
33/33 [=====] - 0s 14ms/step - loss: 0.6938 - val_
Epoch 3/50
33/33 [=====] - 0s 14ms/step - loss: 0.6981 - val_
Epoch 4/50
33/33 [=====] - 0s 15ms/step - loss: 0.6877 - val_
Epoch 5/50
33/33 [=====] - 0s 14ms/step - loss: 0.6855 - val_
Epoch 6/50

```

Results for J4 - Custom Model

```

print("\033[1;31;2m#####-----Custom Model-----#####\033[0m")
RMSE_J4_Custom = RMSE_Value(y_testJ4, PredJ4_Custom)
PredictionsPlot(y_testJ4, PredJ4_Custom, 0)

```

Results for J4 - GRU Model

```

print("\033[1;31;2m#####-----GRU Model-----#####\033[0m")
RMSE_J4_GRU = RMSE_Value(y_testJ4, PredJ4_GRU)
PredictionsPlot(y_testJ4, PredJ4_GRU, 0)

```

Results for J4 - LSTM Model

```

print("\n\033[1;34;2m#####-----LSTM Model-----#####\033[0m")
RMSE_J4_LSTM = RMSE_Value(y_testJ4, PredJ4_LSTM)
PredictionsPlot(y_testJ4, PredJ4_LSTM, 0)

```

Results for J4 - CNN Model

```

print("\n\033[1;32;2m#####-----CNN Model-----#####\033[0m")
RMSE_J4_CNN = RMSE_Value(y_testJ4, PredJ4_CNN)
PredictionsPlot(y_testJ4, PredJ4_CNN, 0)

```

Results for J4 - MLP Model

```

print("\n\033[1;33;2m#####-----MLP Model-----#####\033[0m")
RMSE_J4_MLP = RMSE_Value(y_testJ4, PredJ4_MLP)
PredictionsPlot(y_testJ4, PredJ4_MLP, 0)

```

Create a list of model names and their corresponding RMSE values

```

model_names = ["Custom", "GRU", "LSTM", "CNN", "MLP"]
rmse_values = [RMSE_J4_Custom, RMSE_J4_GRU, RMSE_J4_LSTM, RMSE_J4_CNN, RMSE_J4_

model_rmse = list(zip(model_names, rmse_values))

```



```
Results_df = pd.DataFrame(model_rmse, columns=["MODEL", "RMSE"])
styled_df = Results_df.style.background_gradient(cmap="cool")

# Find the best model with the minimum RMSE value
best_model_index = rmse_values.index(min(rmse_values))
best_model_name_4 = model_names[best_model_index]
best_model_rmse_4 = rmse_values[best_model_index]

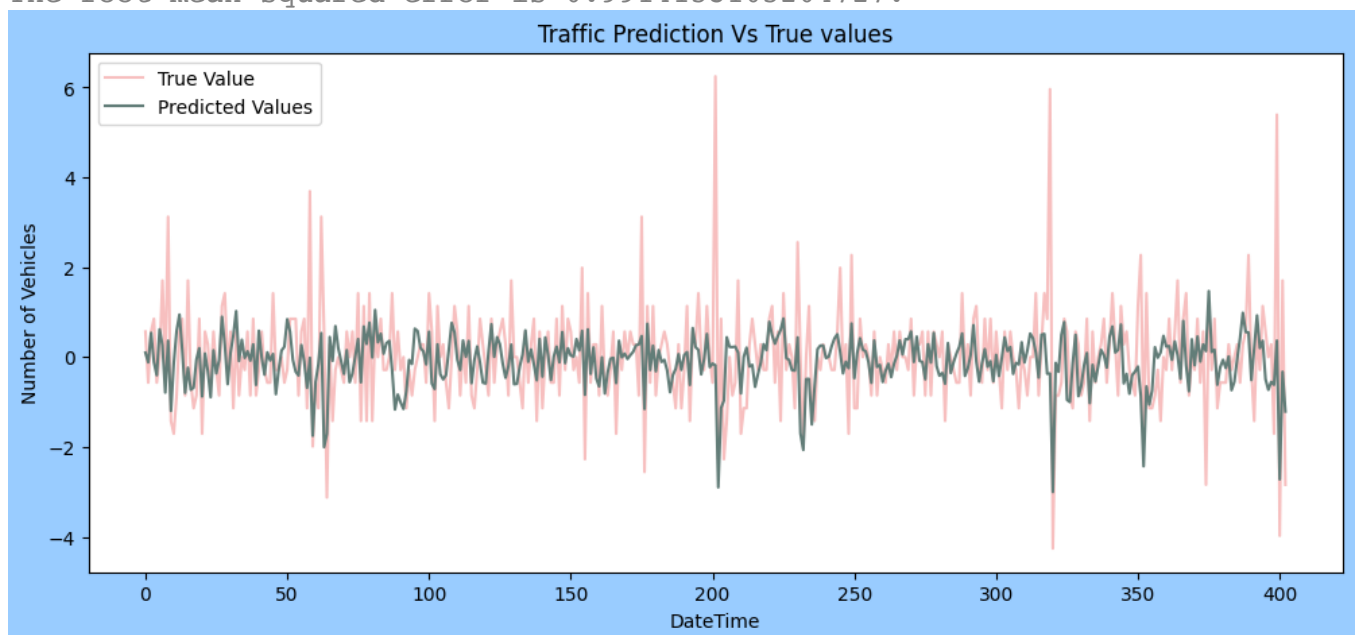
# Print the best model name and its RMSE value
print("\n\033[1;31;4mBest Model: {} - RMSE: {}\n".format(best_model_name_4, best_model_rmse_4))

# Plot the bar graph for model names and RMSE values with transparency and light colors
colors = ['lightcoral', 'lightblue', 'lightgreen', 'lightsalmon', 'lightyellow']
alpha = 0.7
fig, ax = plt.subplots(figsize=(12, 8)) # Set the figure size (width, height)
ax.bar(model_names, rmse_values, color=colors, alpha=alpha)
ax.set_xlabel('Model')
ax.set_ylabel('RMSE Value')
ax.set_title('RMSE Value for Different Models - J4')
plt.show()

display(styled_df)
```

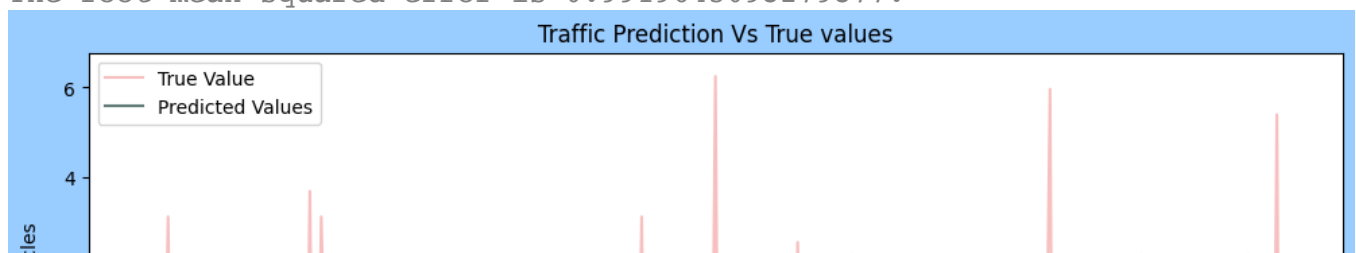
#####-----Custom Model-----#####

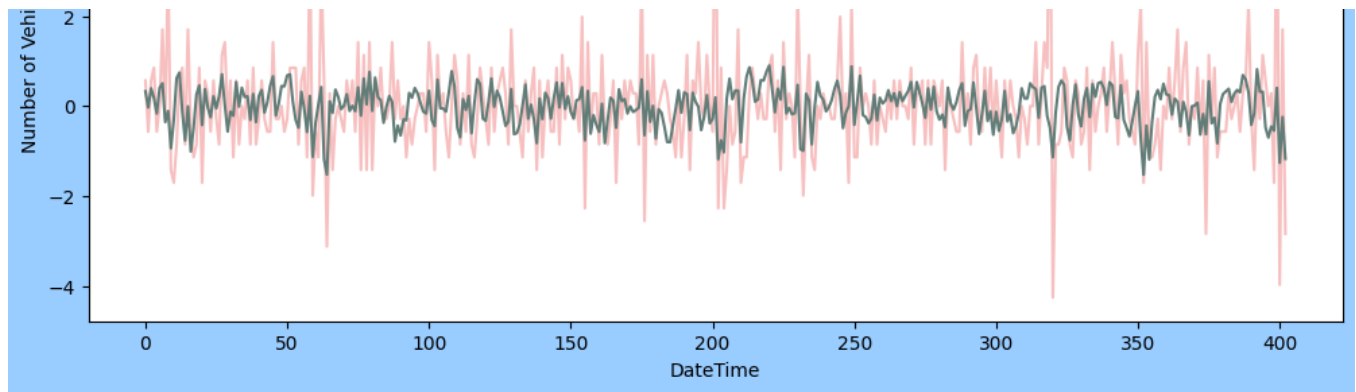
The root mean squared error is 0.9914153105204727.



#####-----GRU Model-----#####

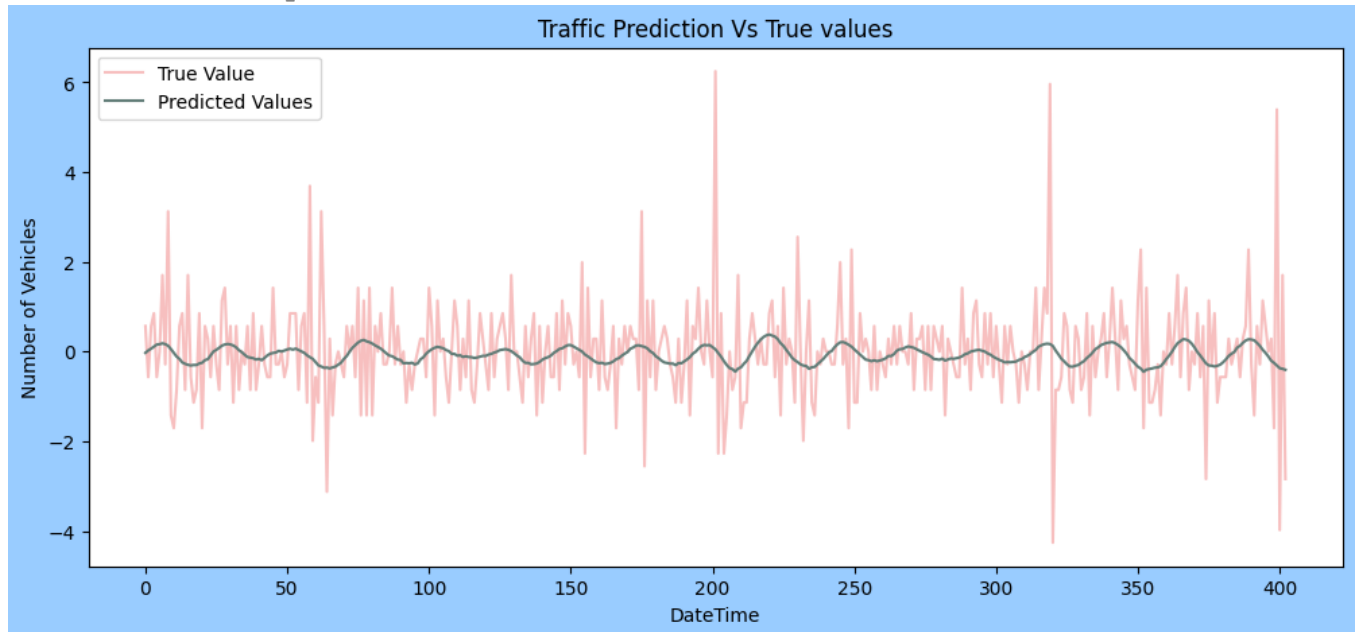
The root mean squared error is 0.9919645093279377.





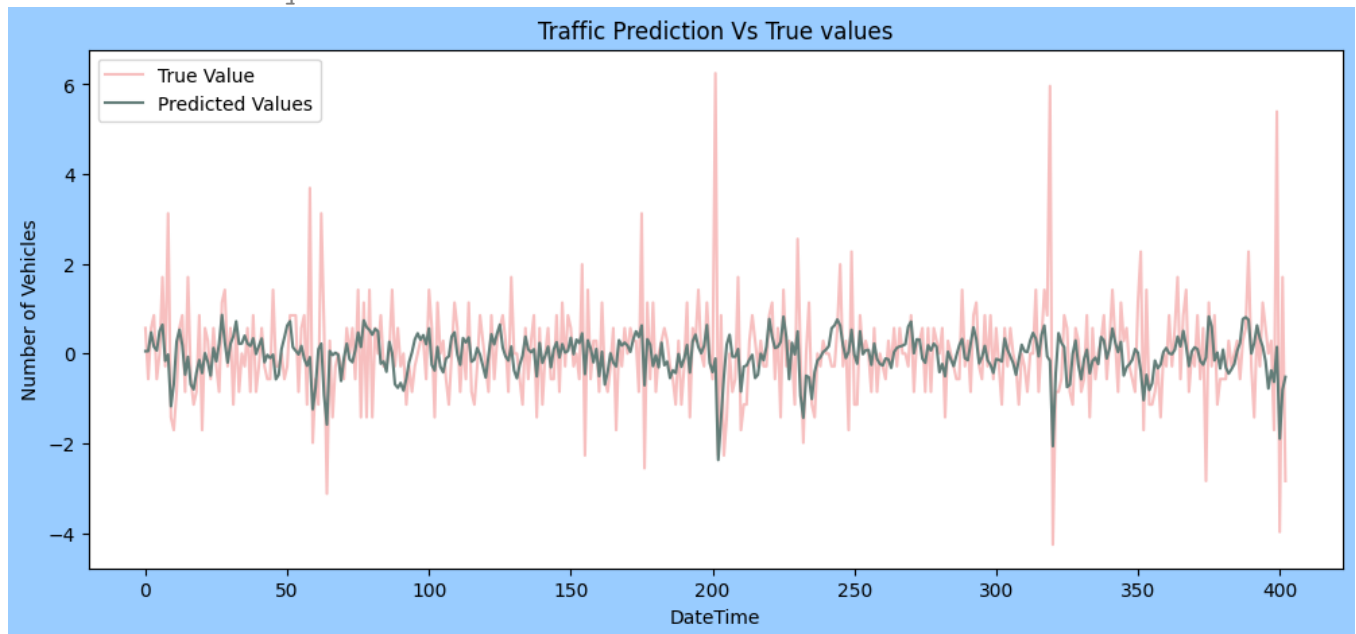
#####-----LSTM Model-----#####

The root mean squared error is 1.0985445909394698.



#####-----CNN Model-----#####

The root mean squared error is 1.0018679252449971.



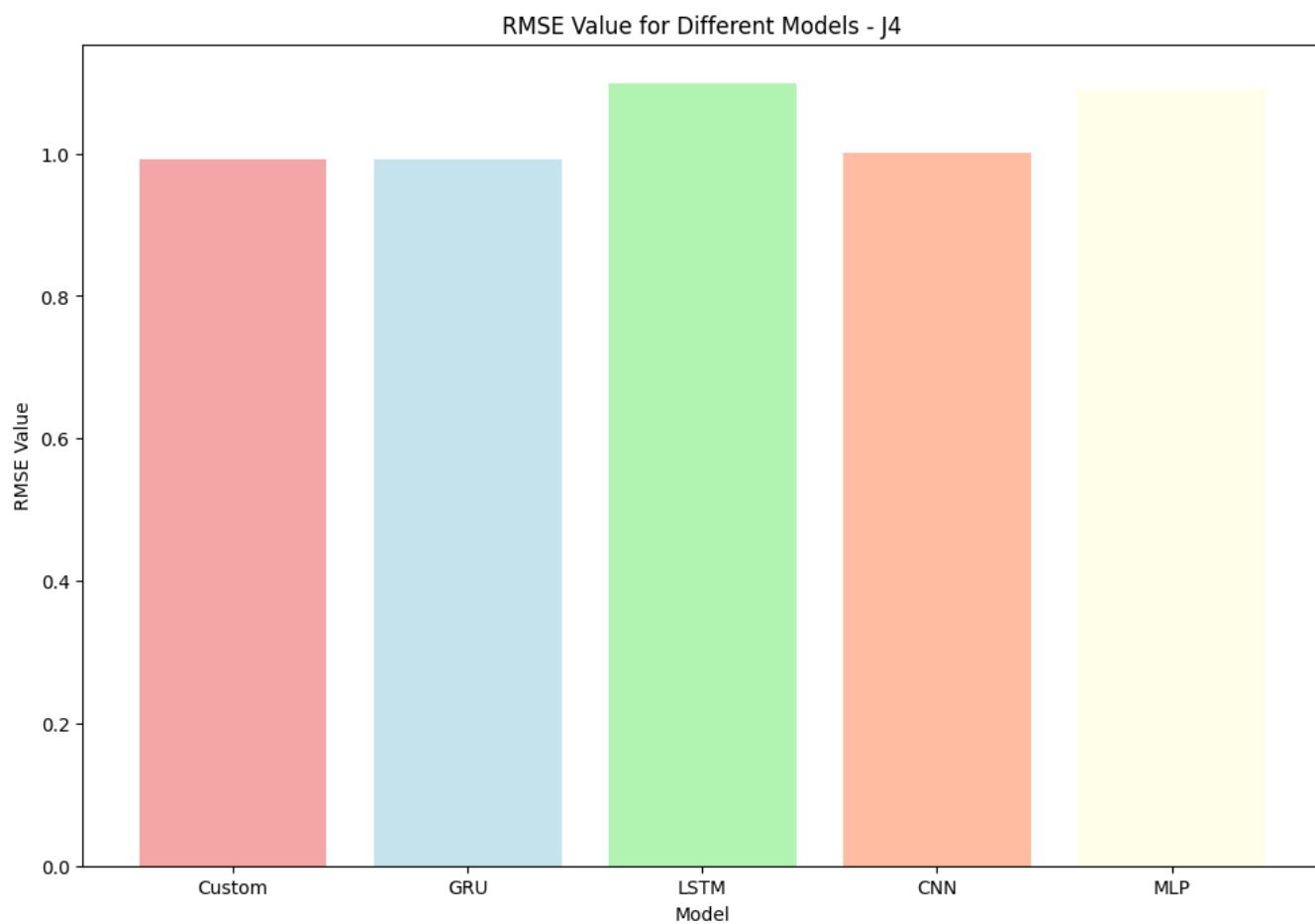
#####-----MLP Model-----#####

The root mean squared error is 1.0015507823080031

THE 1000 MEAN SQUARED ERROR IS 1.0915337023000531.



Best Model: Custom - RMSE: 0.9914153105204727



	MODEL	RMSE
0	Custom	0.991415
1	GRU	0.991965
2	LSTM	1.098545

3	CNN	1.001868
4	MLP	1.091560

The results of the model

```
#Initialise data of lists for error values of four junctions.
Junctions = ["Junction1", "Junction2", "Junction3", "Junction4"]
RMSE = [best_model_rmse_1, best_model_rmse_2, best_model_rmse_3, best_model_rmse_4]
Model = [best_model_name_1, best_model_name_2, best_model_name_3, best_model_name_4]
list_of_tuples = list(zip(Junctions, RMSE, Model))
# Creates pandas DataFrame.
Results = pd.DataFrame(list_of_tuples, columns=["Junction", "RMSE", "MODEL"])
Results.style.background_gradient(cmap="Pastel1")
```

	Junction	RMSE	MODEL
0	Junction1	0.240334	Custom
1	Junction2	0.477646	Custom
2	Junction3	0.565557	Custom
3	Junction4	0.991415	Custom

The Root Mean Square Error is quite a subjective marker for evaluating the performance. Thus, in this project, I am including the outcome plots as well.

✓ Inversing The Transformation Of Data

In this section, I will be inversing transforms that I applied to the datasets to remove the seasonality and trends. Performing this step will make the predictions get back on the accurate scale.

Resource to the inversion process [Link](#)

```
# Functions to inverse transforms and Plot comparative plots
# invert differenced forecast
def inverse_difference(last_ob, value):
    inversed = value + last_ob
    return inversed
#Plotting the comparison
def Sub_Plots2(df_1, df_2,title,m):
    fig, axes = plt.subplots(1, 2, figsize=(18,4), sharey=True,facecolor="#99ccff")
    fig.suptitle(title)

    pl_1=sns.lineplot(ax=axes[0],data=df_1,color=colors[m])
    axes[0].set(ylabel ="Prediction")

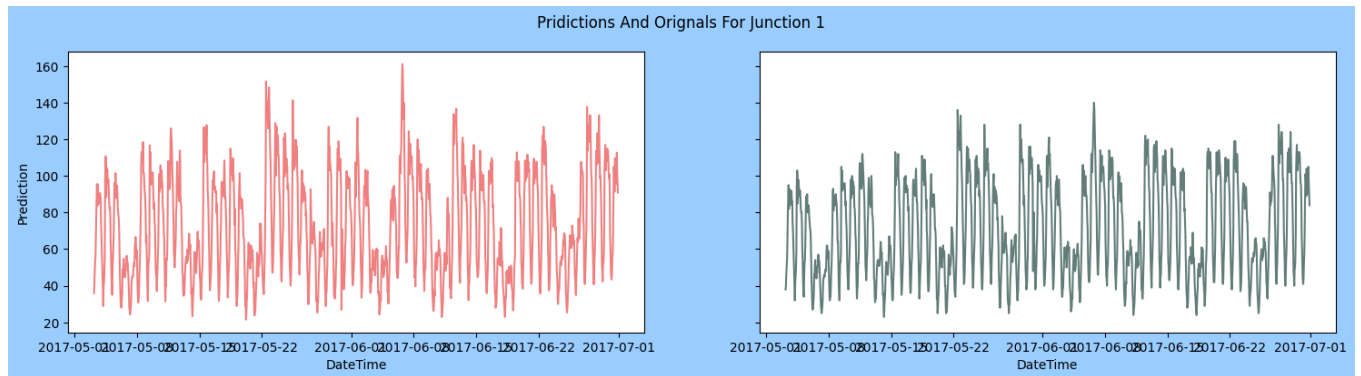
    pl_2=sns.lineplot(ax=axes[1],data=df_2["Vehicles"],color="#627D78")
    axes[1].set(ylabel ="Orignal")
```

The Inverse Transform on the first junction

```

# invert the differenced forecast for Junction 1
recover1 = df_N1.Norm[-1412:-1].to_frame()
recover1["Pred"] = globals()[f"PredJ1_{Model[0]}"]
Transform_reverssed_J1 = inverse_difference(recover1.Norm, recover1.Pred).to_fr
Transform_reverssed_J1.columns = ["Pred_Normed"]
#Invert the normalization J1
Final_J1_Pred = (Transform_reverssed_J1.values * std_J1) + av_J1
Transform_reverssed_J1["Pred_Final"] = Final_J1_Pred
#Plotting the Predictions with originals
Sub_Plots2(Transform_reverssed_J1["Pred_Final"], df_1[-1412:-1], "Pridictions Ar

```

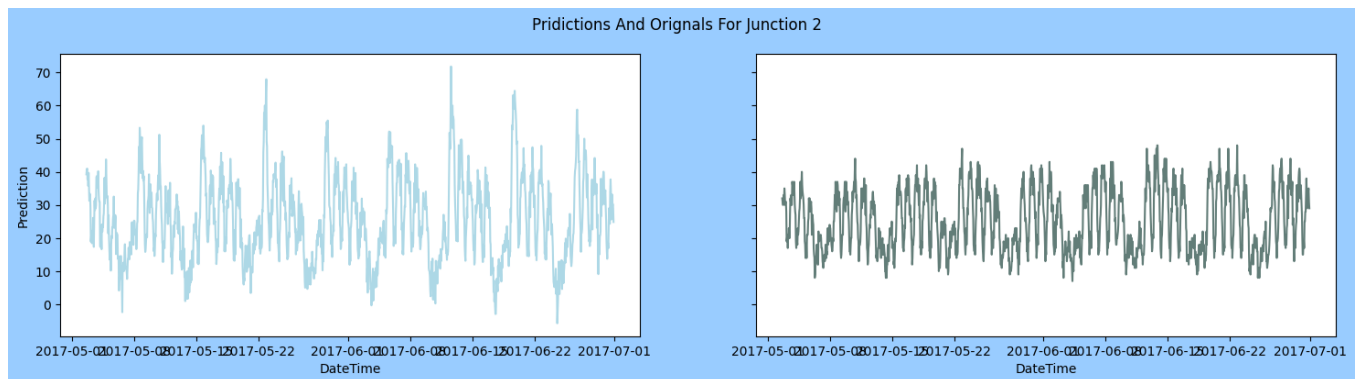


the inverse transform on the second junction

```

#Invert the differenced J2
recover2 = df_N2.Norm[-1426:-1].to_frame() #len as per the diff
recover2["Pred"] = globals()[f"PredJ2_{Model[1]}"]
Transform_reverssed_J2 = inverse_difference(recover2.Norm, recover2.Pred).to_fr
Transform_reverssed_J2.columns = ["Pred_Normed"]
Final_J2_Pred = (Transform_reverssed_J2.values * std_J2) + av_J2
Transform_reverssed_J2["Pred_Final"] = Final_J2_Pred
#Plotting the Predictions with originals
Sub_Plots2(Transform_reverssed_J2["Pred_Final"], df_2[-1426:-1], "Pridictions Ar

```

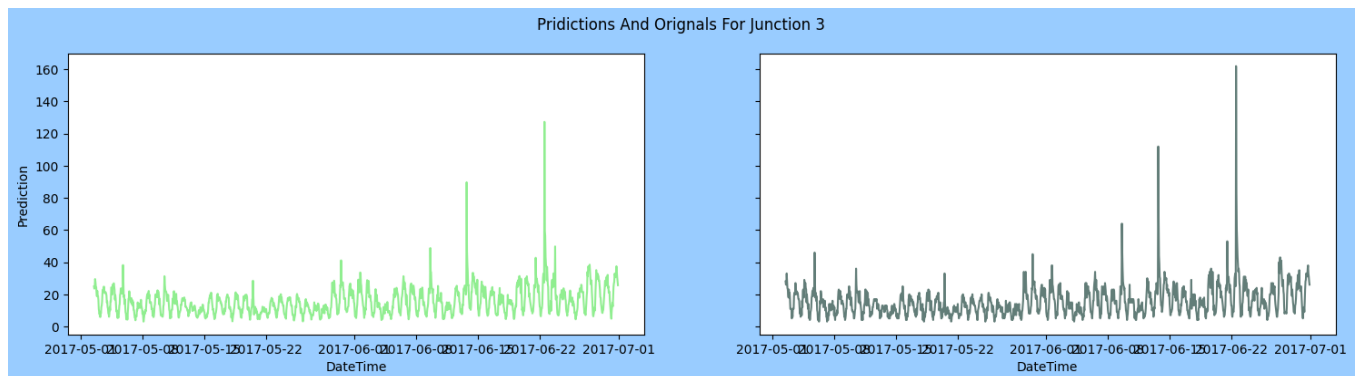


The inverse transform on the third junction


```

#Invert the differenced J3
recover3 = df_N3.Norm[-1429:-1].to_frame() #len as per the diff
recover3["Pred"] = globals()[f"PredJ3_{Model[2]}"]
Transform_reverssed_J3 = inverse_difference(recover3.Norm, recover3.Pred).to_fr
Transform_reverssed_J3.columns = ["Pred_Normed"]
#Invert the normalization J3
Final_J3_Pred = (Transform_reverssed_J3.values * std_J3) + av_J3
Transform_reverssed_J3["Pred_Final"] = Final_J3_Pred
Sub_Plots2(Transform_reverssed_J3["Pred_Final"], df_3[-1429:-1], "Pridictions Ar

```

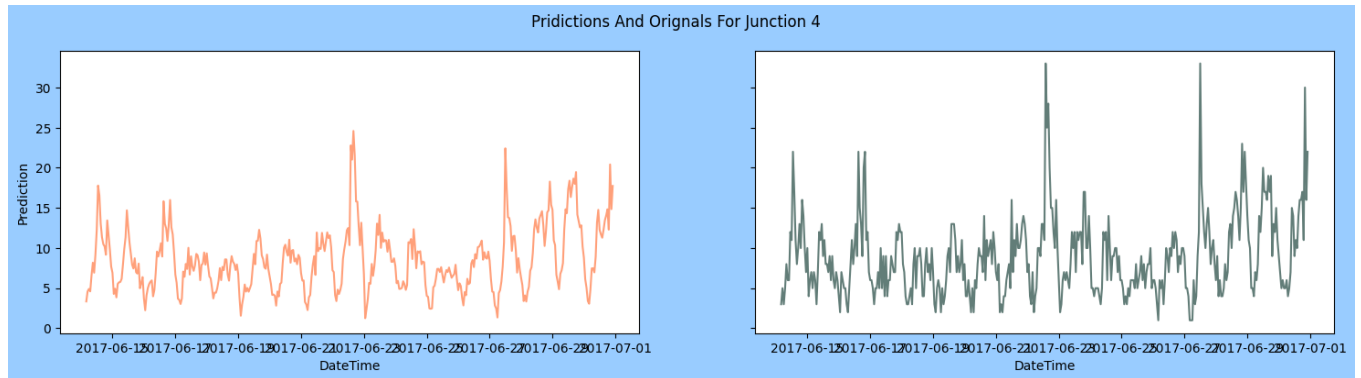


The inverse transform on the fourth junction

```

#Invert the differenced J4
recover4 = df_N4.Norm[-404:-1].to_frame() #len as per the testset
recover4["Pred"] = globals()[f"PredJ4_{Model[3]}"]
Transform_reverssed_J4 = inverse_difference(recover4.Norm, recover4.Pred).to_fr
Transform_reverssed_J4.columns = ["Pred_Normed"]
#Invert the normalization J4
Final_J4_Pred = (Transform_reverssed_J4.values * std_J4) + av_J4
Transform_reverssed_J4["Pred_Final"] = Final_J4_Pred
Sub_Plots2(Transform_reverssed_J4["Pred_Final"], df_4[-404:-1], "Pridictions And

```



Summary

In this project, I trained a GRU, CNN, MLP, LSTM and my custom Model to predicted the traffic on four junctions. I used a normalisation and differencing transform to achieve a stationary timeseries. As the Junctions vary in trends and seasonality, I took diffrent approach for each junction to make it stationary. I applyied the root mean squared error as the evaluation metric for the model. In addition to that I plotted the Predictions alongside the original test values. Take aways from the data analysis:

The Number of vehicles in Junction one is rising more rapidly compaired to junction two and three. The sparsity of data in juction four bars me from making any conclusion on the same.

The Junction one's traffic has a stronger weekly seasonality as well as hourly seasonality. Where as other junctions are significantly linear.

✓ End