

Code link:

[https://colab.research.google.com/drive/1iUqsbWtDc9EyWVleSXast5QFBFt3g1f7#scrollTo=Pjj\\_9ygGTlqZ](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
```

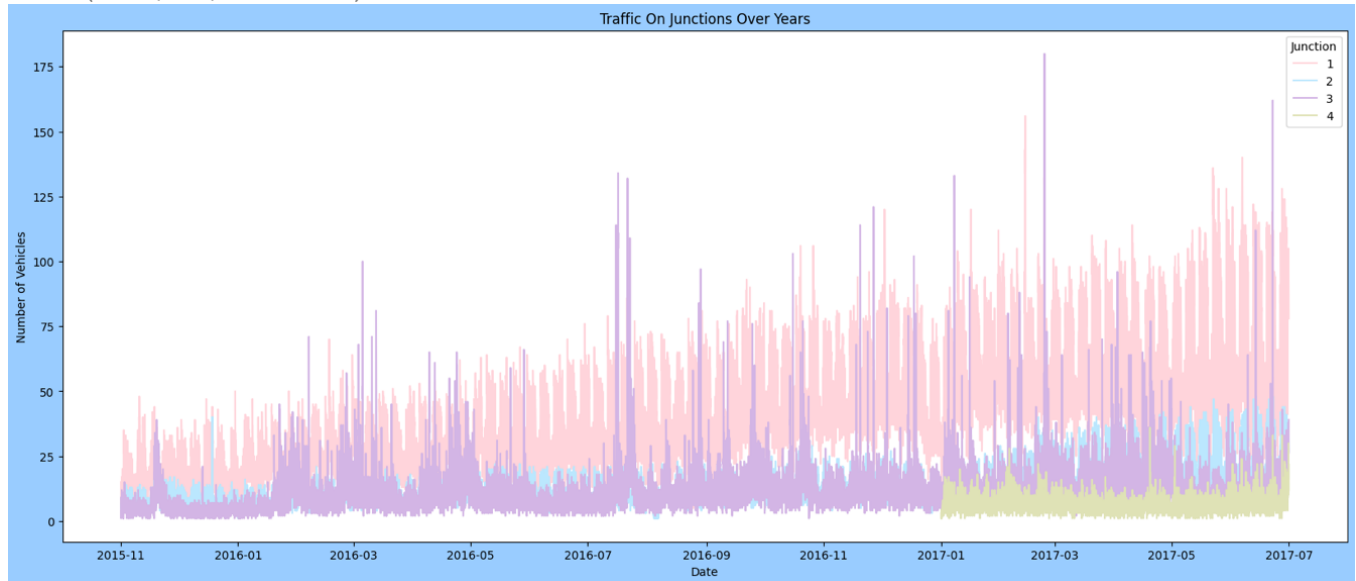
Print a concise summary of a DataFrame.

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')
```



### Key observations from the plot include:

- The initial junction displays a clear upward trend.
- Sparse data for the fourth junction begins only after 2017.
- There is no apparent seasonality depicted in the plot, indicating the need to delve into the datetime composition for further insights.

### ▼ Feature Engineering

During this stage, I am generating several new features based on DateTime. These include:

- 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

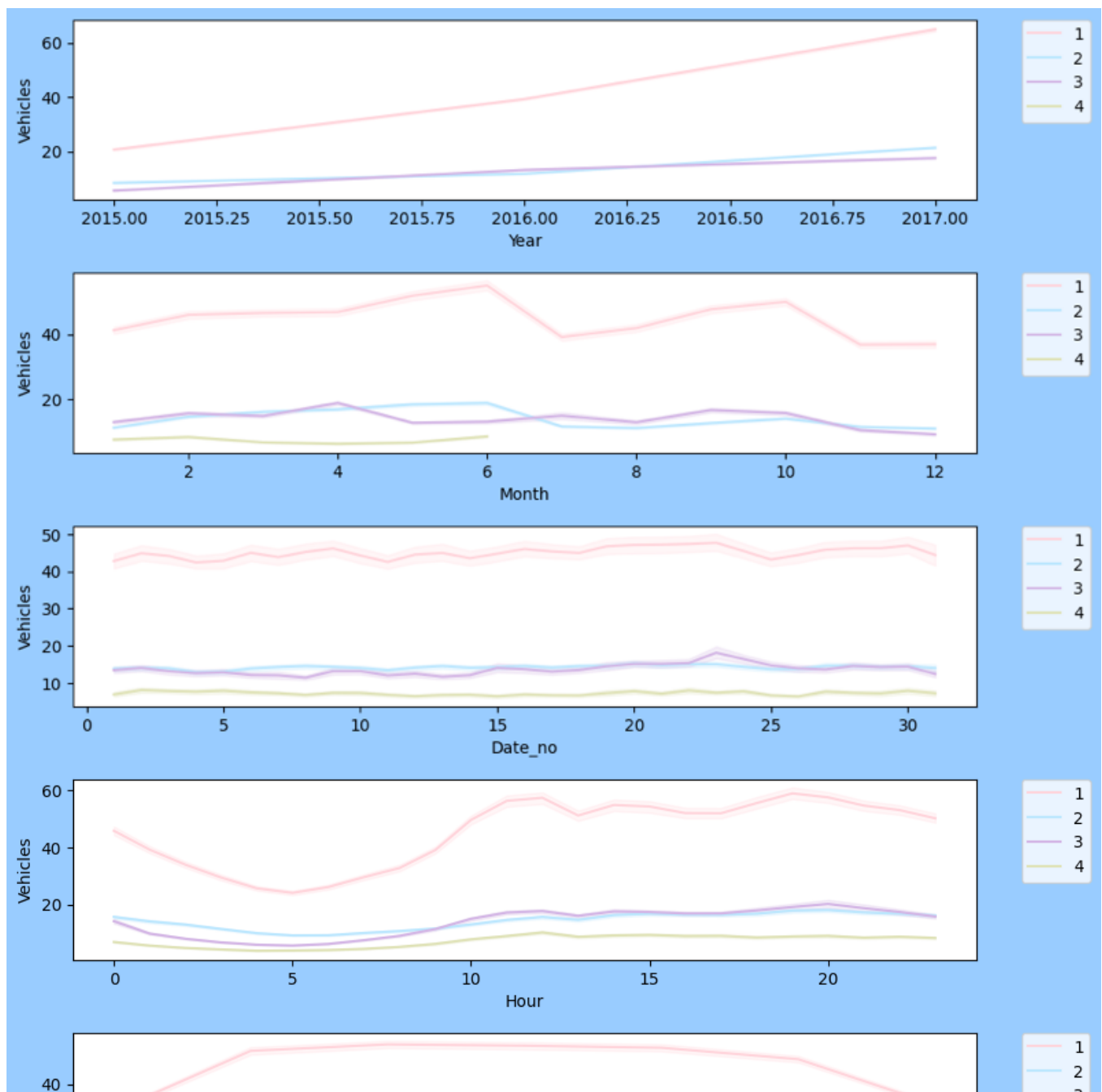
```
new_features = [ "Year","Month", "Date_no", "Hour", "Day"]
```

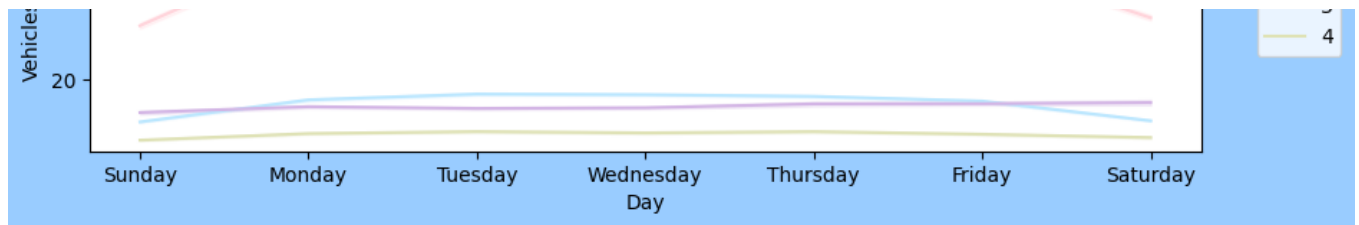
```
for i in new_features:
```

```
    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.)
```



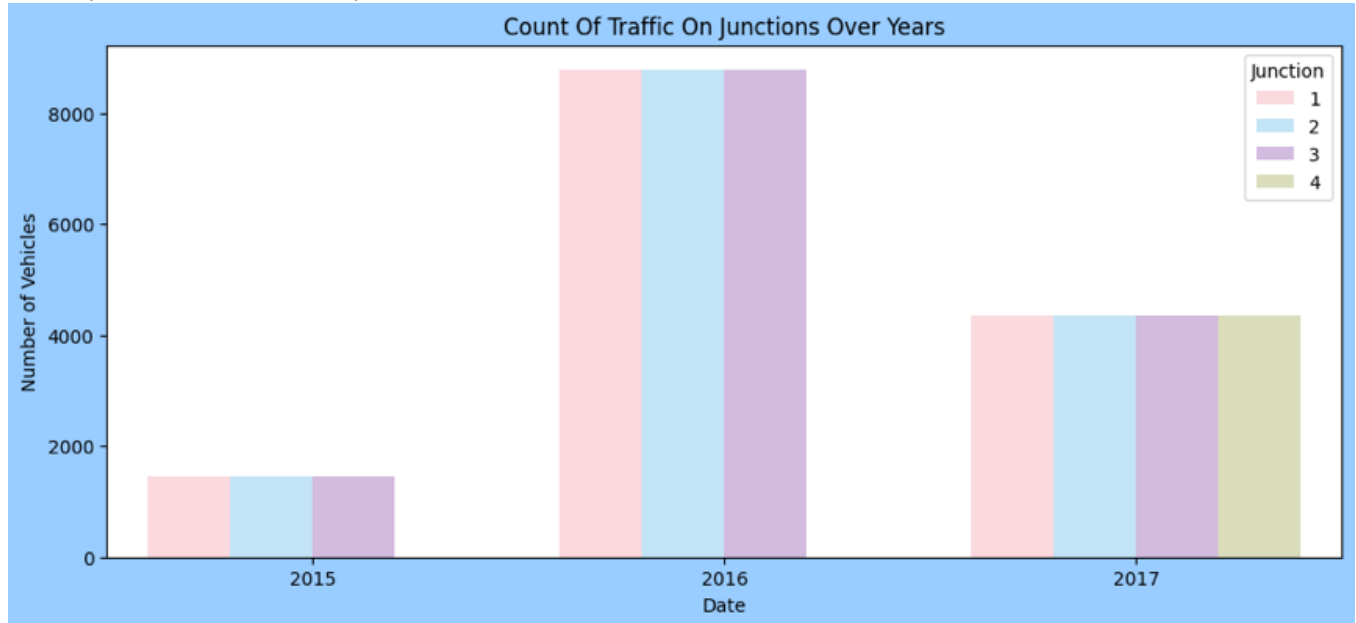


**Based on the above plot, the following conclusions can be drawn:**

- Yearly trends show an upward trajectory for all junctions, except the fourth junction, which lacks sufficient data spanning a year.
- Notably, there is a surge in traffic at the first and second junctions around June, possibly attributed to summer break and related activities.
- Monthly data exhibits consistent patterns across all dates.
- Daily patterns reveal peaks during morning and evening hours, with a decline during nighttime, aligning with expectations.
- Weekly observations indicate smoother traffic on Sundays due to fewer vehicles on the roads, while traffic remains steady from Monday to Friday.

```
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")
```

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



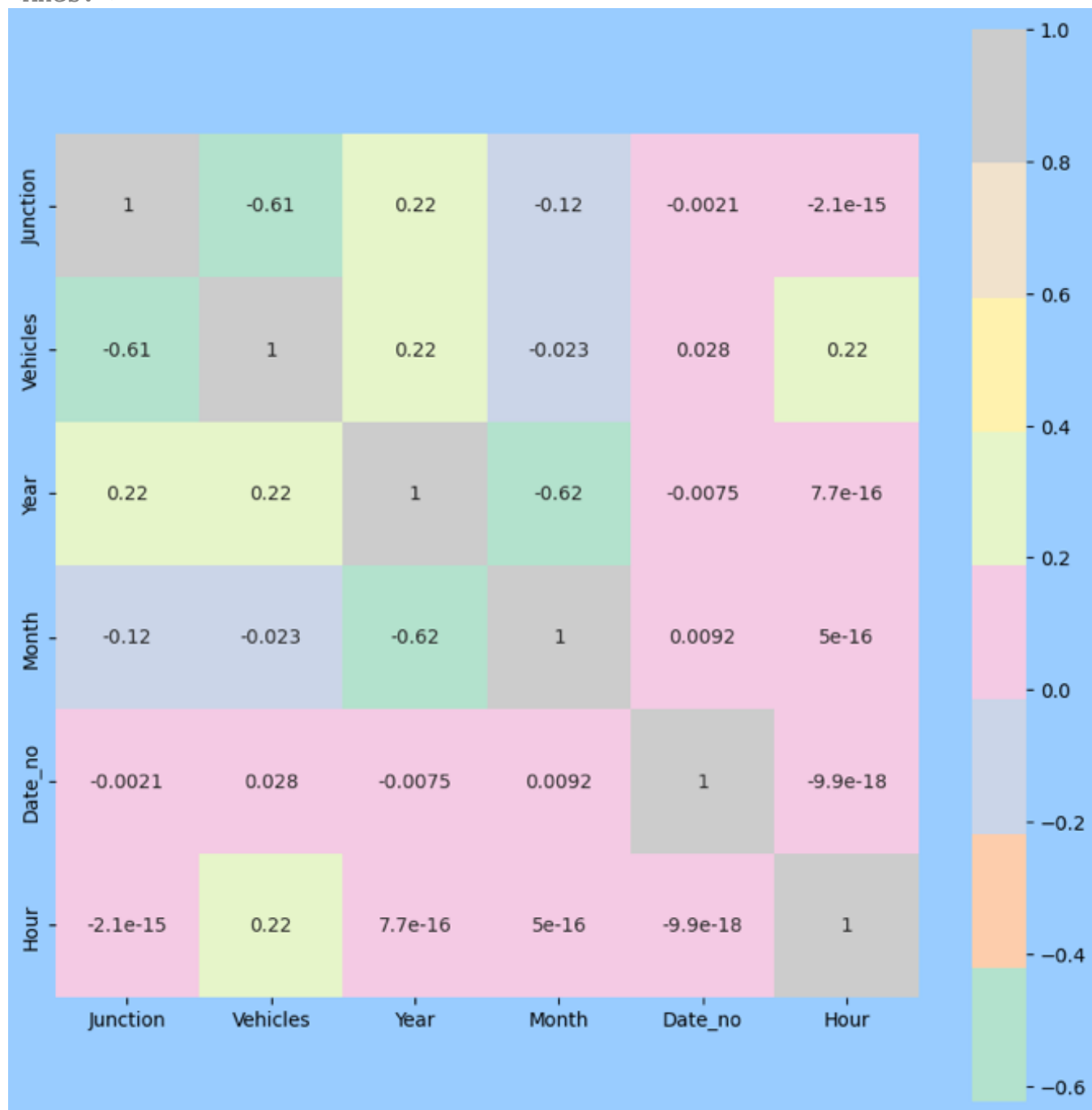
The count plot indicates a rise in vehicle numbers from 2015 to 2016. However, drawing conclusions for 2017 is inconclusive due to limited data available, only extending to the seventh month of the year.

```
# 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)
```

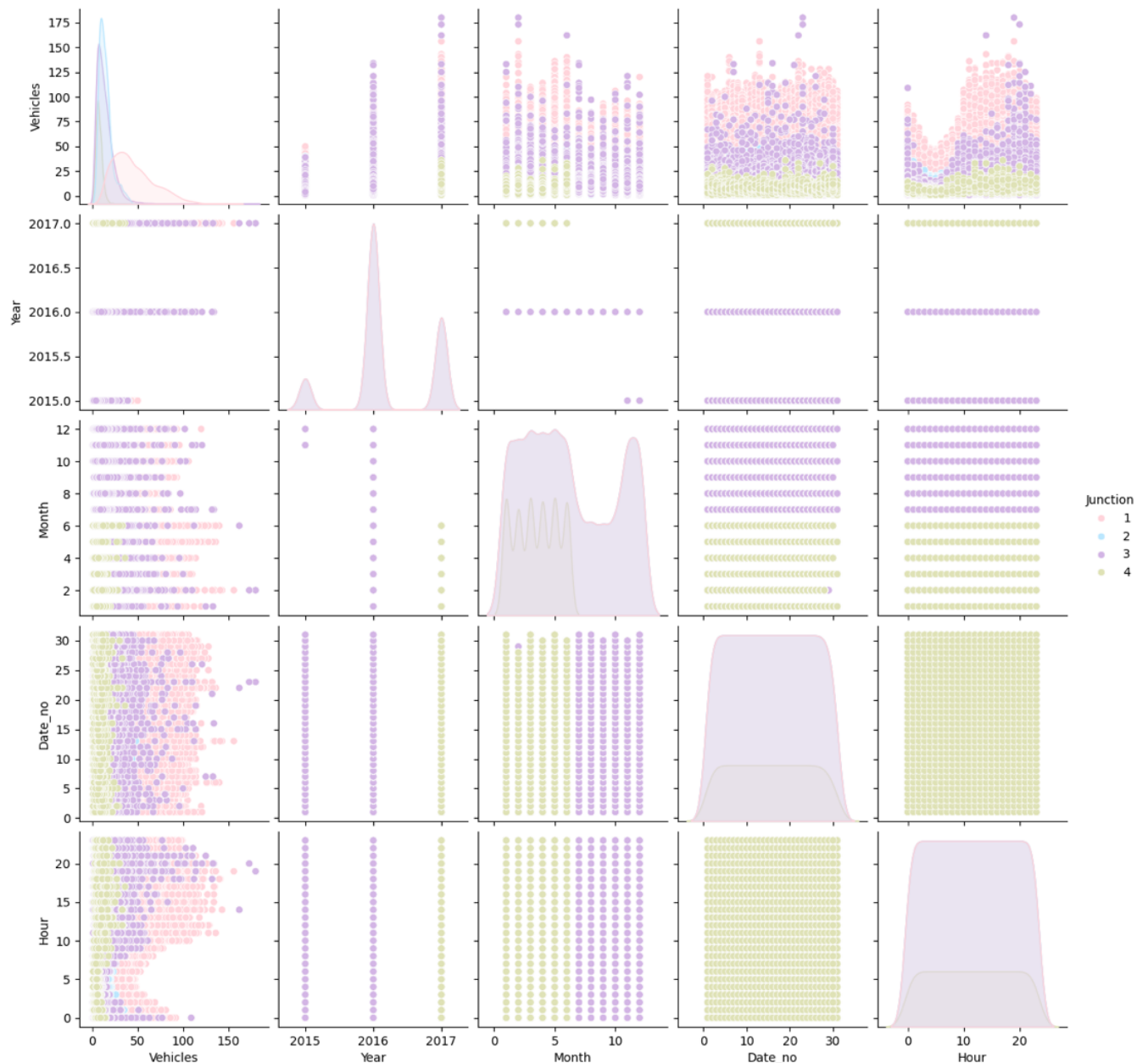
&lt;Axes: &gt;



The preexisting feature exhibits the highest correlation. To conclude my exploratory data analysis (EDA), I will utilize a pair plot, providing a comprehensive visual representation of the data.

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

<seaborn.axisgrid.PairGrid at 0x7f0cc55928c0>



### **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]
```

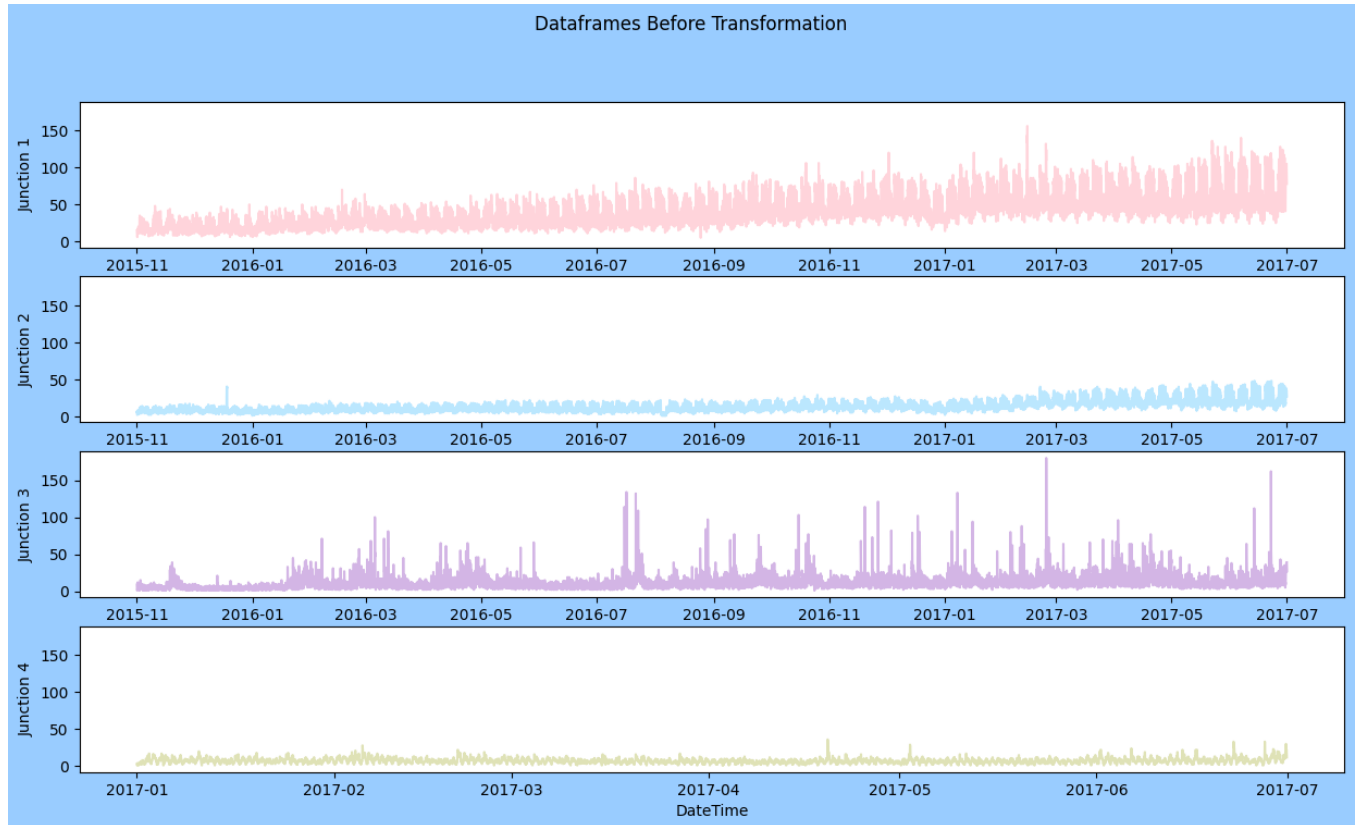
```
for i in list_dfs:
```

```
    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 considered stationary if it lacks both trend and seasonality. However, our exploratory data analysis (EDA) revealed the presence of weekly seasonality and an upward trend over the years. As demonstrated in the above plot, it's reaffirmed that Junctions one and two exhibit an upward trend. Limiting the data span would likely accentuate the weekly seasonality further. However, I'll defer this step for now and proceed with the respective transformations on the 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
```

Based on the aforementioned observations, differencing to remove seasonality should be executed as outlined below:

- For Junction one: Differencing of weekly values will be undertaken.
- For Junction two: Employing the difference of consecutive days is deemed more suitable.
- For Junctions three and four: Differencing of hourly values will suffice for eliminating seasonality.

```
#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

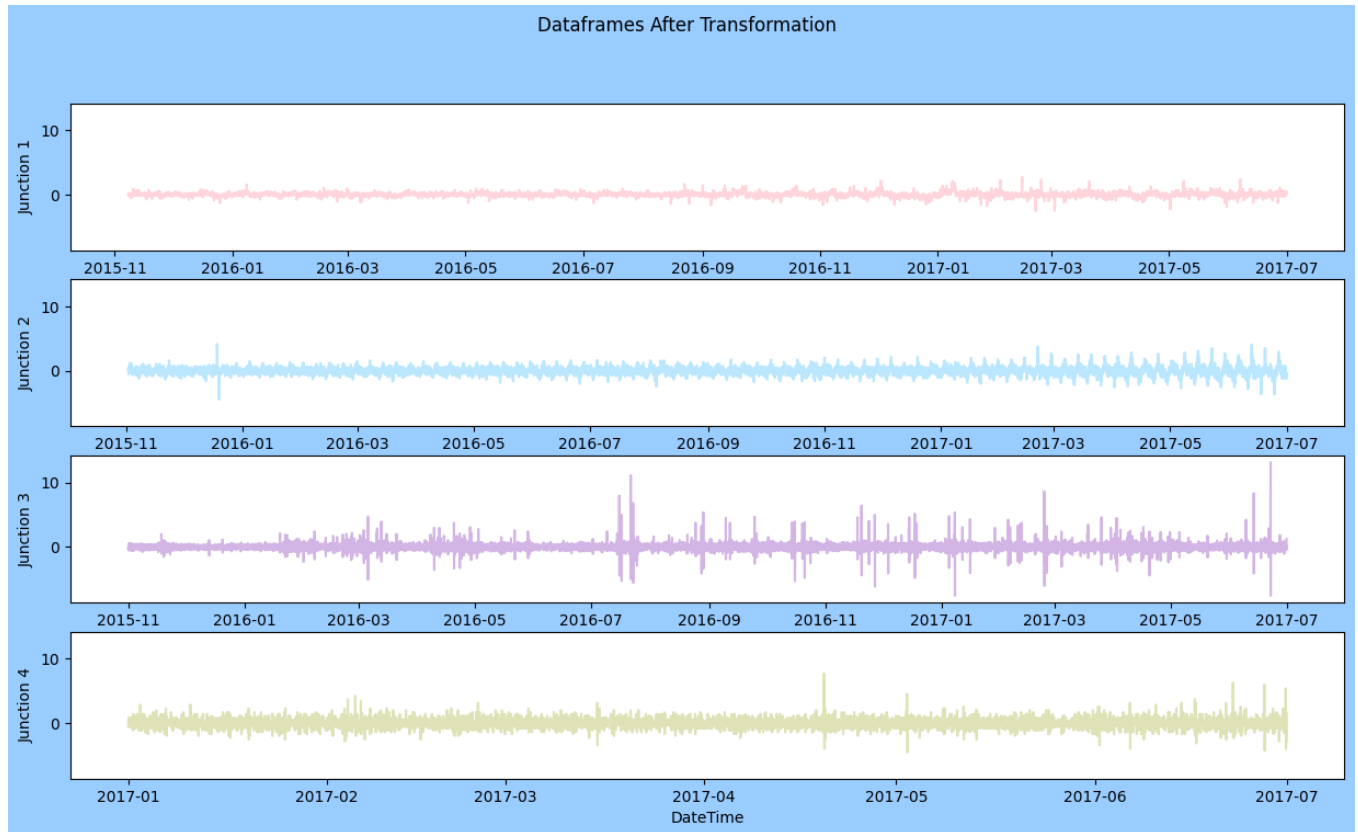
df_N2, av_J2, std_J2 = Normalize(df_2, "Vehicles")
Diff_2 = Difference(df_N2, col="Vehicles", interval=(24)) #taking a day's diffre
df_N2 = df_N2[24:]
df_N2.columns = ["Norm"]
df_N2["Diff"] = Diff_2

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 depicted above appear to exhibit linearity. To verify their stationarity, I will conduct 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: -15.265303390415337
p-value: 4.79853987639816e-28
Critical Values:
    1%: -3.431
    5%: -2.862
   10%: -2.567
Time Series is Stationary
```

```
ADF Statistic: -21.795891026940065
p-value: 0.0
Critical Values:
    1%: -3.431
    5%: -2.862
   10%: -2.567
Time Series is Stationary
```

```
ADF Statistic: -28.001759908832316
p-value: 0.0
Critical Values:
    1%: -3.431
    5%: -2.862
   10%: -2.567
Time Series is Stationary
```

```
ADF Statistic: -17.979092563052276
p-value: 2.778787532595421e-30
Critical Values:
    1%: -3.432
    5%: -2.862
   10%: -2.567
Time Series is Stationary
```

**With the data now stationary, preprocessing for the neural network involves:**

- Dividing the dataset into test and train sets.
- Designating X as the features and y as the target variable.
- Reshaping the data appropriately for the neural network.

```
#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 developing a function that the neural network can utilize to 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

Next, I will fit the transformed training sets of the four junctions to the created model and compare them with 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 [=====] - 1s 6ms/step - loss: 0.0686 - val
Epoch 2/50
108/108 [=====] - 0s 4ms/step - loss: 0.0570 - val
Epoch 3/50
108/108 [=====] - 0s 4ms/step - loss: 0.0535 - val
Epoch 4/50
108/108 [=====] - 0s 4ms/step - loss: 0.0520 - val
Epoch 5/50
108/108 [=====] - 0s 4ms/step - loss: 0.0503 - val
Epoch 6/50
108/108 [=====] - 0s 4ms/step - loss: 0.0502 - val
Epoch 7/50
108/108 [=====] - 0s 4ms/step - loss: 0.0485 - val
Epoch 8/50
108/108 [=====] - 0s 4ms/step - loss: 0.0482 - val
Epoch 9/50
108/108 [=====] - 0s 4ms/step - loss: 0.0486 - val
Epoch 10/50
108/108 [=====] - 0s 4ms/step - loss: 0.0476 - val
Epoch 11/50
108/108 [=====] - 0s 4ms/step - loss: 0.0481 - val
Epoch 12/50
108/108 [=====] - 0s 4ms/step - loss: 0.0471 - val
Epoch 13/50
108/108 [=====] - 0s 3ms/step - loss: 0.0473 - val
Epoch 14/50
108/108 [=====] - 0s 3ms/step - loss: 0.0470 - val
Epoch 15/50
108/108 [=====] - 0s 3ms/step - loss: 0.0471 - val
Epoch 16/50
108/108 [=====] - 0s 4ms/step - loss: 0.0470 - val
Epoch 17/50
108/108 [=====] - 0s 3ms/step - loss: 0.0464 - val
Epoch 18/50
108/108 [=====] - 0s 3ms/step - loss: 0.0465 - val
Epoch 19/50
108/108 [=====] - 1s 6ms/step - loss: 0.0465 - val
45/45 [=====] - 0s 2ms/step
```

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

```
Epoch 1/50
108/108 [=====] - 50s 370ms/step - loss: 0.0749 -
Epoch 2/50
108/108 [=====] - 40s 374ms/step - loss: 0.0656 -
Epoch 3/50
108/108 [=====] - 40s 367ms/step - loss: 0.0638 -
Epoch 4/50
108/108 [=====] - 40s 369ms/step - loss: 0.0624 -
Epoch 5/50
108/108 [=====] - 40s 372ms/step - loss: 0.0612 -
Epoch 6/50
108/108 [=====] - 38s 351ms/step - loss: 0.0593 -
Epoch 7/50
108/108 [=====] - 40s 373ms/step - loss: 0.0576 -
Epoch 8/50
108/108 [=====] - 40s 374ms/step - loss: 0.0573 -
Epoch 9/50
```

```
# 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_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_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

# Plot the bar graph for model names and RMSE values with transparency and light
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)

```

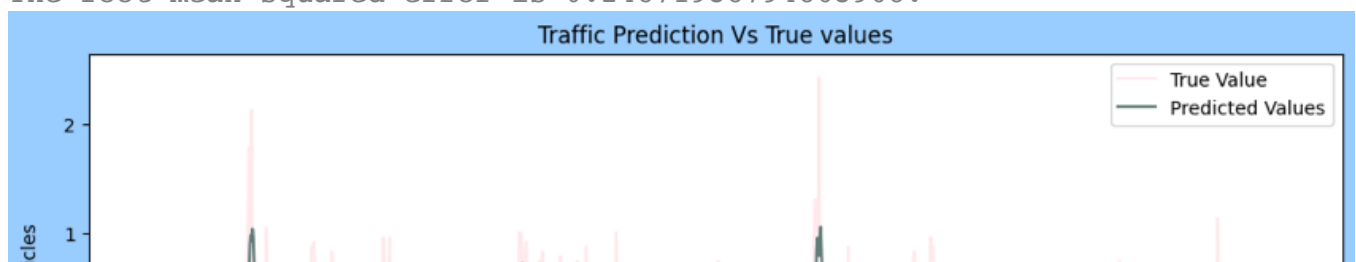
#####-----Custom Model-----#####

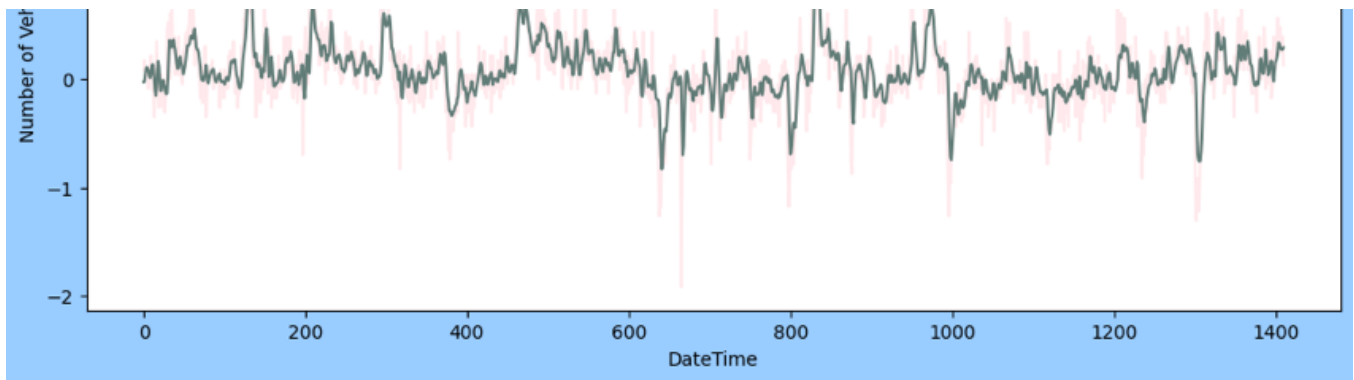
The root mean squared error is 0.24280191448043165.



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

The root mean squared error is 0.24871938794863968.





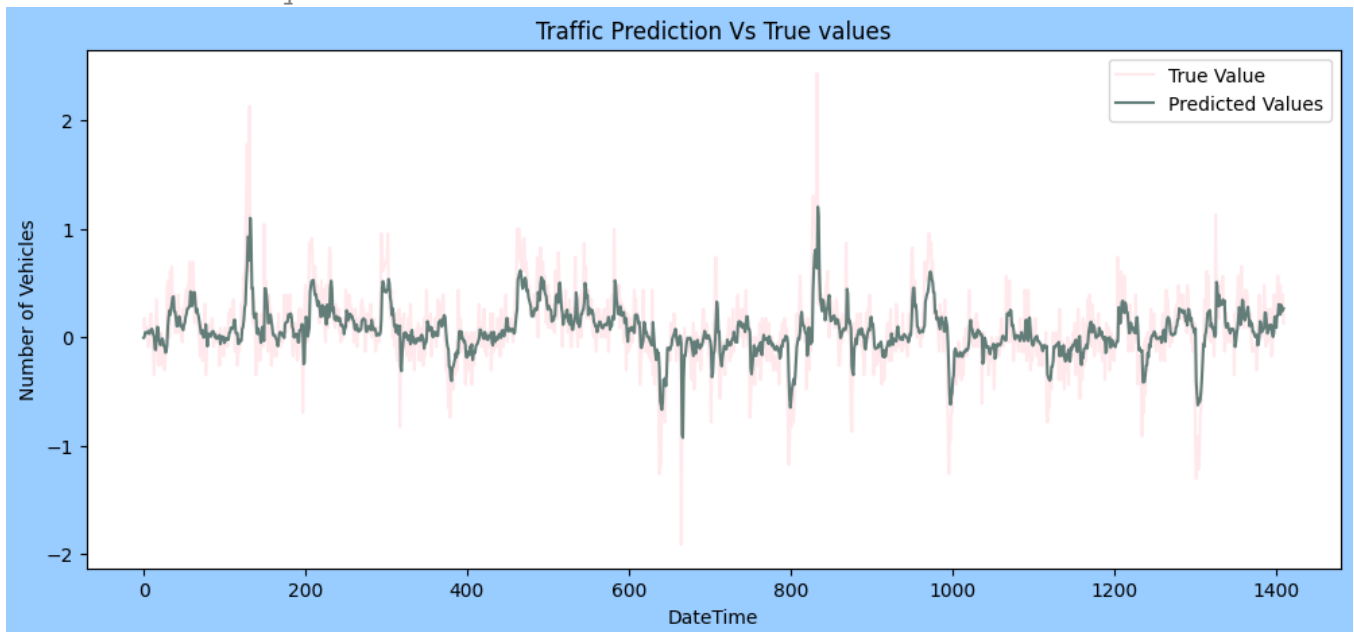
#####-----LSTM Model-----#####

The root mean squared error is 0.2701606590390193.



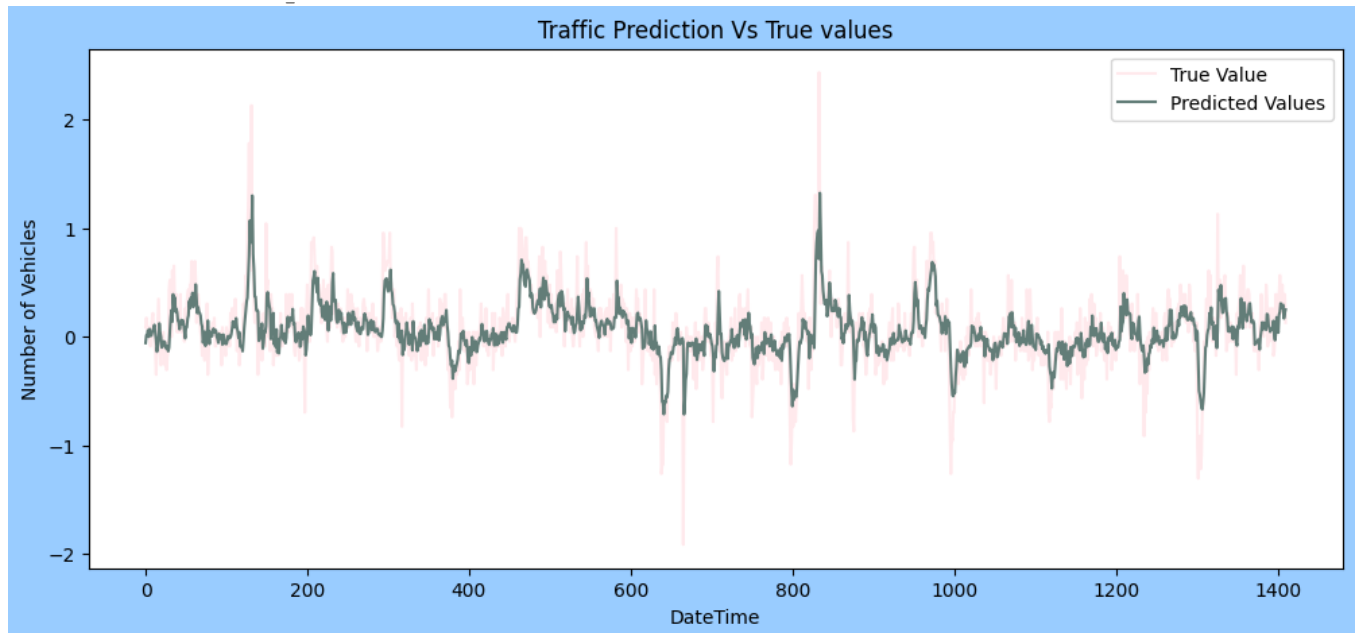
#####-----CNN Model-----#####

The root mean squared error is 0.24563708664795814.

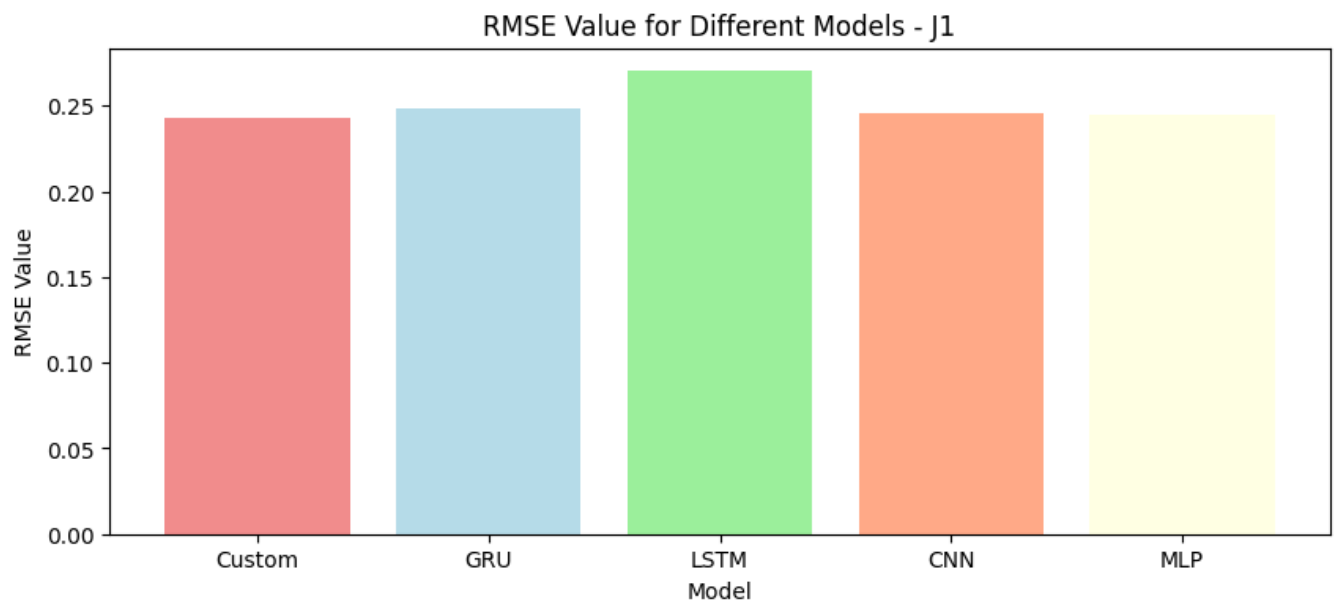


#####-----MLP Model-----#####

The root mean squared error is 0.24445737269962944.



**Best Model: Custom - RMSE: 0.24280191448043165**



	MODEL	RMSE
0	Custom	0.242802
1	GRU	0.248719
2	LSTM	0.270161
3	CNN	0.245637
4	MLP	0.244457



```
# Predictions For Second Junction
```

```

# Predictions for second function
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: 0.2126 - val
Epoch 2/50
109/109 [=====] - 0s 4ms/step - loss: 0.1695 - val
Epoch 3/50
109/109 [=====] - 0s 4ms/step - loss: 0.1587 - val
Epoch 4/50
109/109 [=====] - 0s 4ms/step - loss: 0.1532 - val
Epoch 5/50
109/109 [=====] - 0s 4ms/step - loss: 0.1492 - val
Epoch 6/50
109/109 [=====] - 0s 4ms/step - loss: 0.1485 - val
Epoch 7/50
109/109 [=====] - 0s 4ms/step - loss: 0.1479 - val
Epoch 8/50
109/109 [=====] - 0s 4ms/step - loss: 0.1484 - val
Epoch 9/50
109/109 [=====] - 1s 5ms/step - loss: 0.1454 - val
Epoch 10/50
109/109 [=====] - 0s 4ms/step - loss: 0.1433 - val
Epoch 11/50
109/109 [=====] - 0s 4ms/step - loss: 0.1431 - val
Epoch 12/50
109/109 [=====] - 1s 5ms/step - loss: 0.1419 - val
Epoch 13/50
109/109 [=====] - 1s 6ms/step - loss: 0.1423 - val
Epoch 14/50
109/109 [=====] - 1s 6ms/step - loss: 0.1405 - val
Epoch 15/50
109/109 [=====] - 1s 6ms/step - loss: 0.1408 - val
Epoch 16/50
109/109 [=====] - 1s 6ms/step - loss: 0.1390 - val
Epoch 17/50
109/109 [=====] - 1s 7ms/step - loss: 0.1394 - val

```

```

Epoch 18/50
109/109 [=====] - 1s 6ms/step - loss: 0.1381 - val
Epoch 19/50
109/109 [=====] - 0s 4ms/step - loss: 0.1398 - val
Epoch 20/50
109/109 [=====] - 0s 4ms/step - loss: 0.1380 - val
Epoch 21/50
109/109 [=====] - 0s 3ms/step - loss: 0.1388 - val
Epoch 22/50
109/109 [=====] - 0s 4ms/step - loss: 0.1380 - val
Epoch 23/50
109/109 [=====] - 0s 4ms/step - loss: 0.1386 - val
Epoch 24/50
109/109 [=====] - 0s 4ms/step - loss: 0.1372 - val
Epoch 25/50
109/109 [=====] - 0s 4ms/step - loss: 0.1366 - val
Epoch 26/50
109/109 [=====] - 0s 4ms/step - loss: 0.1360 - val
Epoch 27/50
109/109 [=====] - 1s 5ms/step - loss: 0.1365 - val
Epoch 28/50
109/109 [=====] - 0s 4ms/step - loss: 0.1349 - val
Epoch 29/50
109/109 [=====] - 0s 4ms/step - loss: 0.1357 - val

```

# 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]

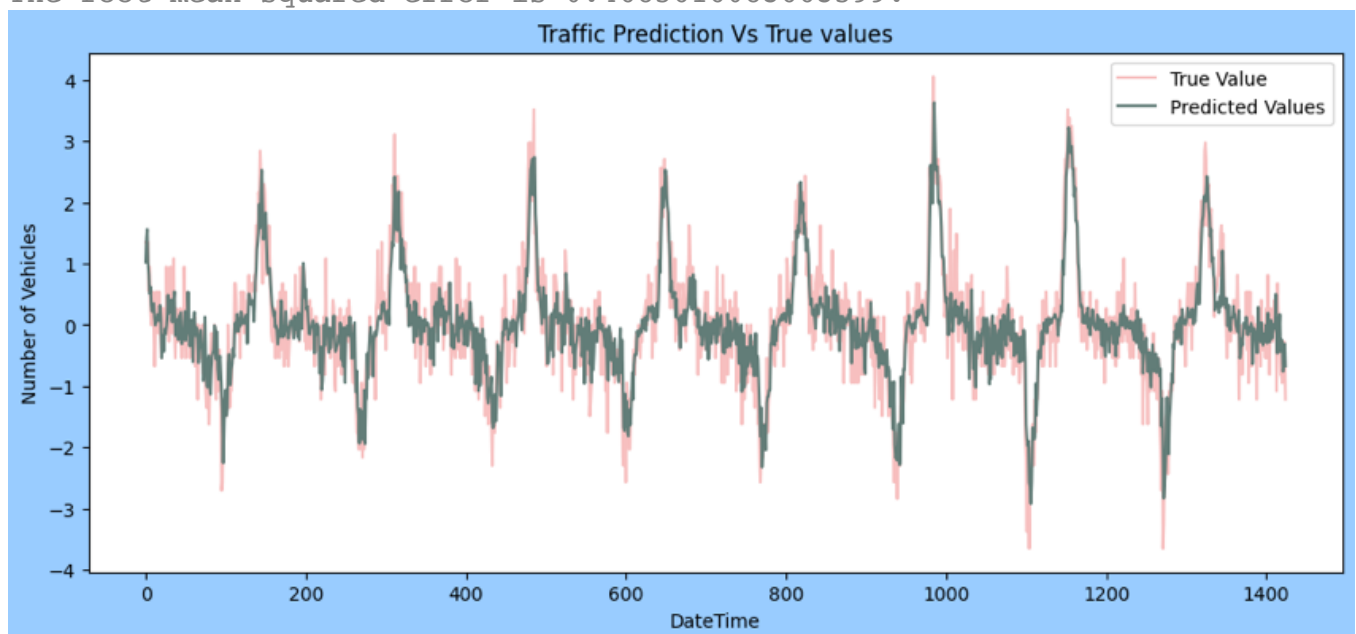
# Print the best model name and its RMSE value
print("\n\033[1;31;4mBest Model: {} - RMSE: {}\n".format(best_model_name_2, best_model_rmse_2))

# 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)
```

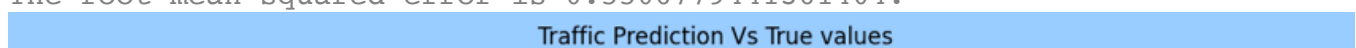
#####-----Custom Model-----#####

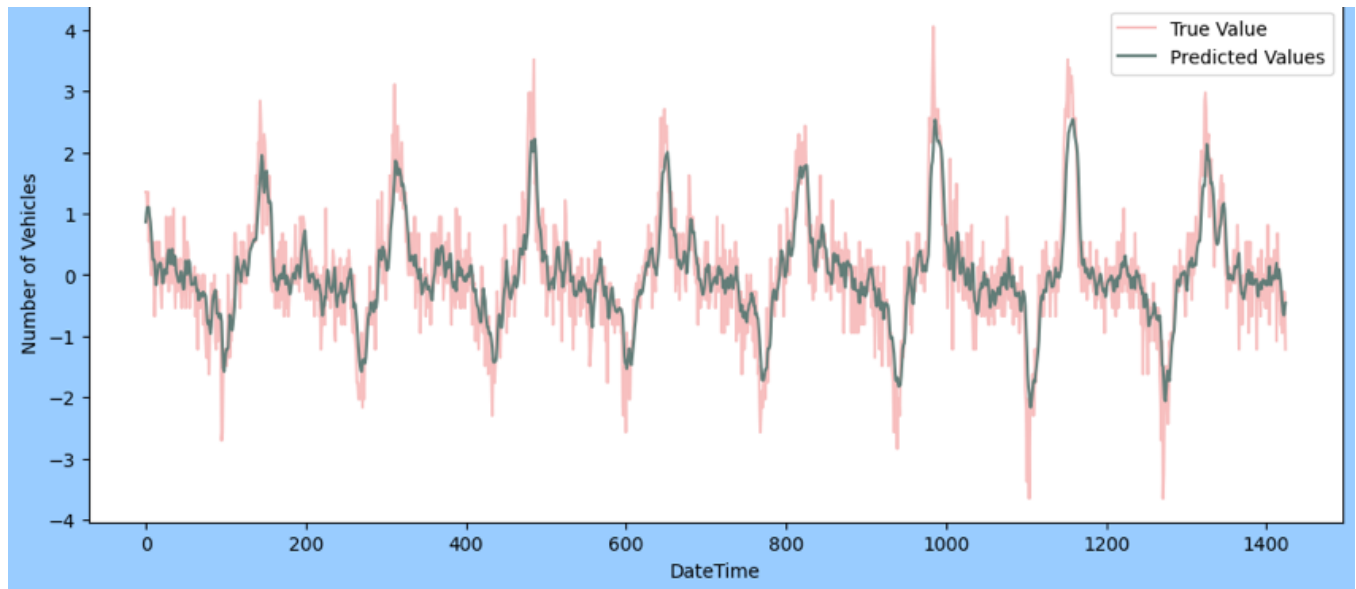
The root mean squared error is 0.4685616083663599.



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

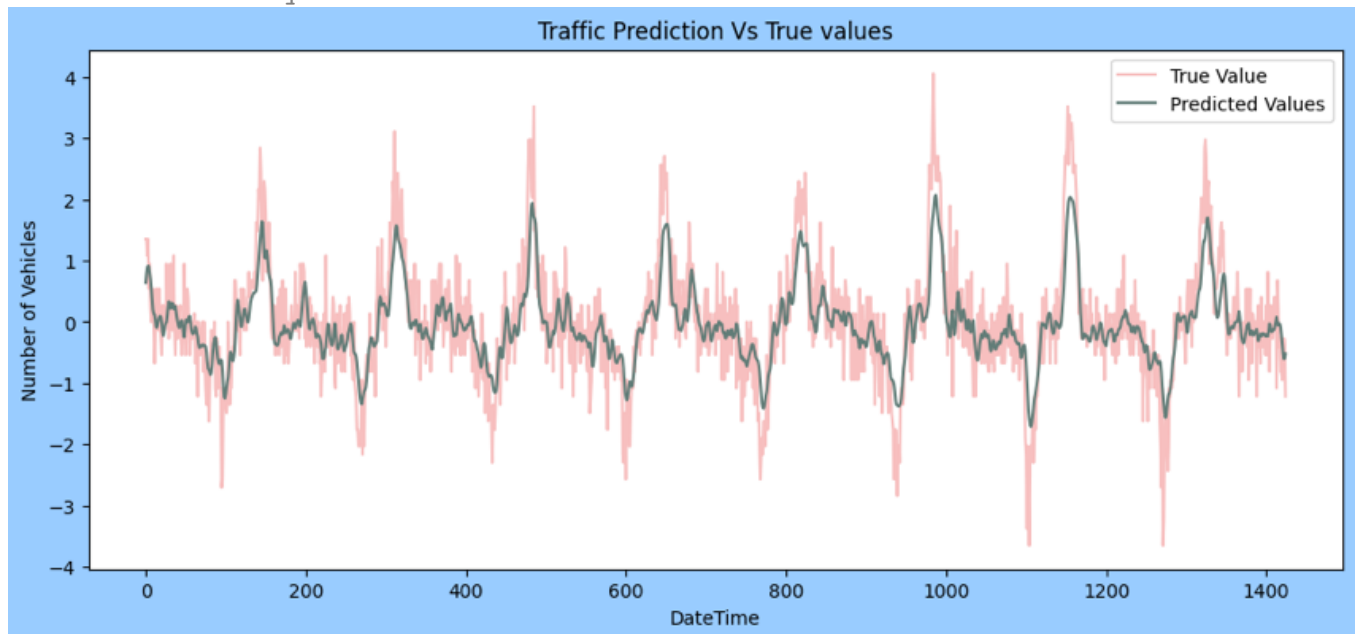
The root mean squared error is 0.5506779441361404.





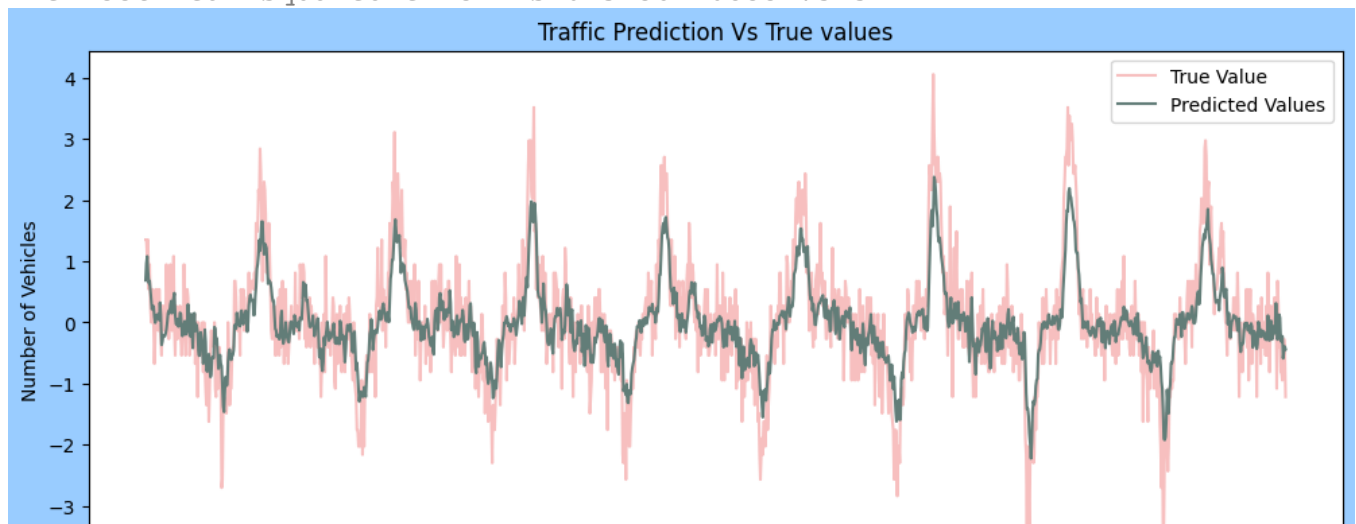
#####-----LSTM Model-----#####

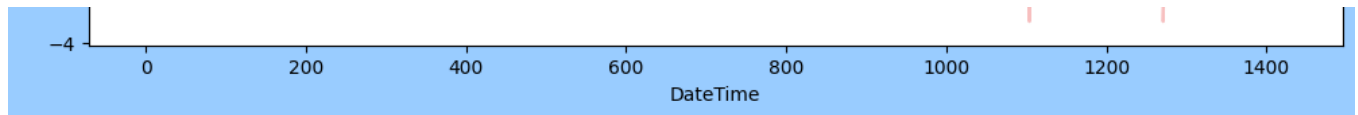
The root mean squared error is 0.5859443577976761.



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

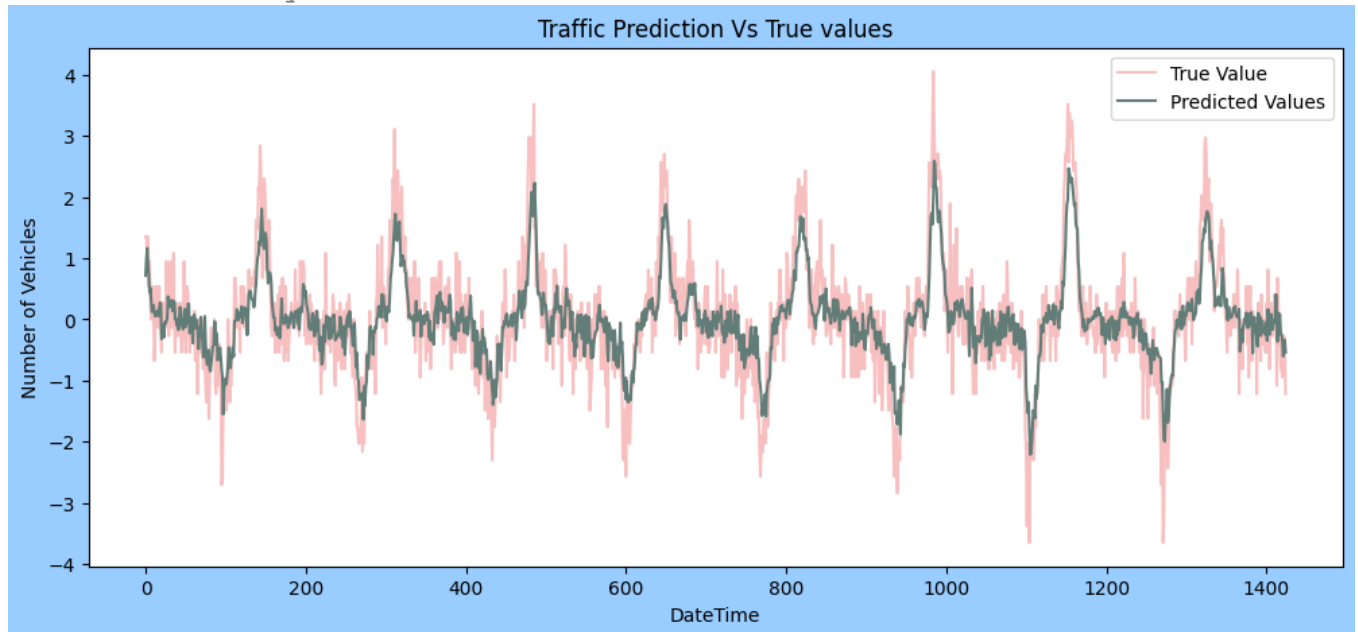
The root mean squared error is 0.5436110833275152.



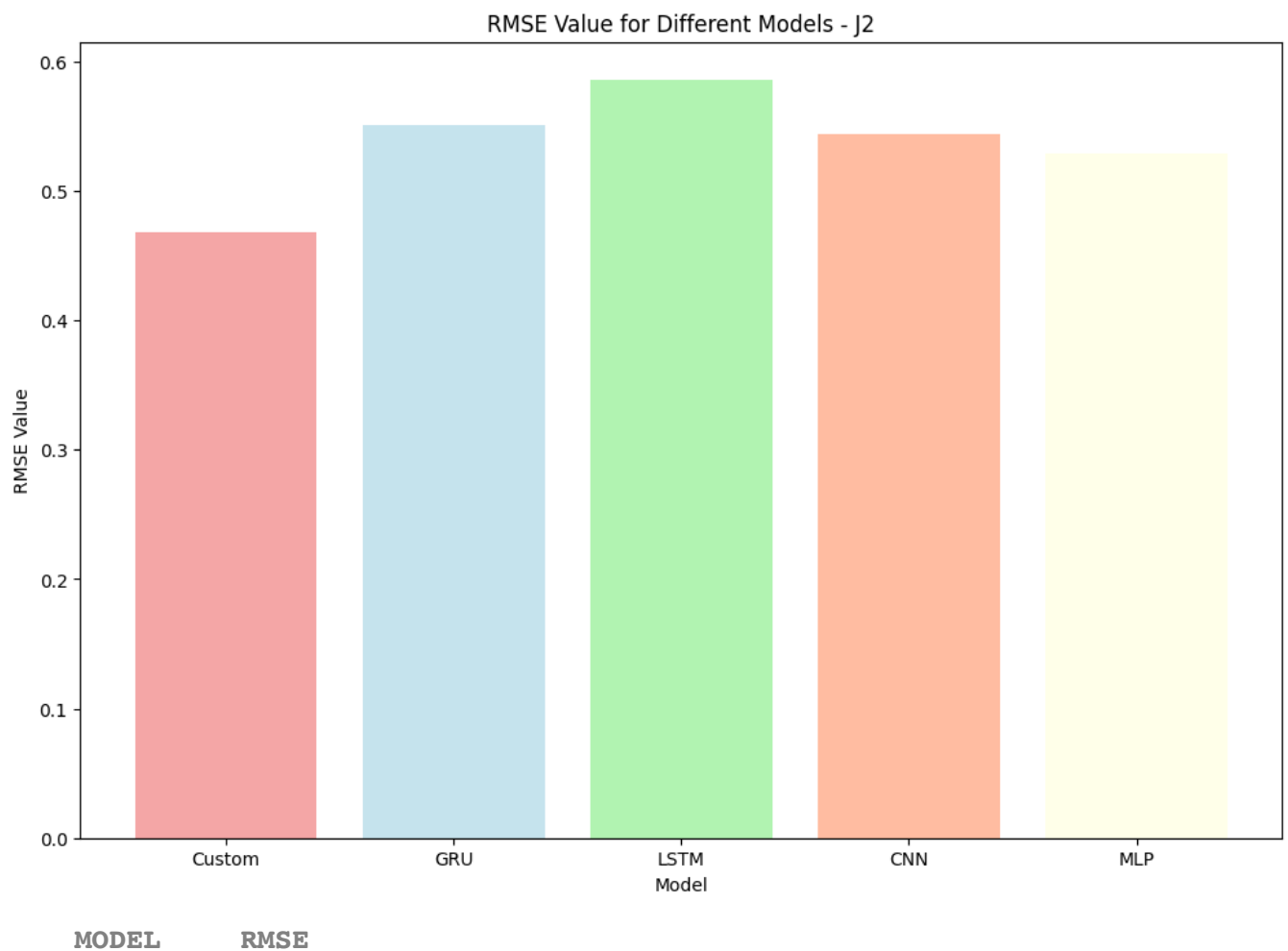


#####-----MLP Model-----#####

The root mean squared error is 0.5295006153075514.



**Best Model: Custom - RMSE: 0.4685616083663599**



---

0	Custom	0.468562
1	GRU	0.550678
2	LSTM	0.585944
3	CNN	0.543611
4	MLP	0.529501

```

# 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 9ms/step - loss: 0.3015 - val

Epoch 2/50

110/110 [=====] - 1s 7ms/step - loss: 0.2808 - val

Epoch 3/50

110/110 [=====] - 0s 4ms/step - loss: 0.2655 - val

Epoch 4/50

110/110 [=====] - 0s 3ms/step - loss: 0.2596 - val

Epoch 5/50



```

110/110 [=====] - 0s 4ms/step - loss: 0.2573 - val
Epoch 6/50
110/110 [=====] - 1s 5ms/step - loss: 0.2529 - val
Epoch 7/50
110/110 [=====] - 0s 4ms/step - loss: 0.2493 - val
Epoch 8/50
110/110 [=====] - 0s 4ms/step - loss: 0.2451 - val
Epoch 9/50
110/110 [=====] - 0s 4ms/step - loss: 0.2432 - val
Epoch 10/50
110/110 [=====] - 0s 4ms/step - loss: 0.2425 - val
Epoch 11/50
110/110 [=====] - 0s 4ms/step - loss: 0.2400 - val
Epoch 12/50
110/110 [=====] - 0s 4ms/step - loss: 0.2382 - val
Epoch 13/50
110/110 [=====] - 0s 4ms/step - loss: 0.2364 - val
Epoch 14/50
110/110 [=====] - 0s 4ms/step - loss: 0.2327 - val
Epoch 15/50
110/110 [=====] - 0s 4ms/step - loss: 0.2351 - val
45/45 [=====] - 0s 2ms/step

```

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

```

Epoch 1/50
110/110 [=====] - 52s 389ms/step - loss: 0.2977 -
Epoch 2/50
110/110 [=====] - 42s 378ms/step - loss: 0.2914 -
Epoch 3/50
110/110 [=====] - 41s 374ms/step - loss: 0.2942 -
Epoch 4/50
110/110 [=====] - 40s 365ms/step - loss: 0.2920 -
Epoch 5/50
110/110 [=====] - 39s 354ms/step - loss: 0.2899 -
Epoch 6/50
110/110 [=====] - 39s 354ms/step - loss: 0.2915 -
Epoch 7/50
110/110 [=====] - 41s 369ms/step - loss: 0.2888 -
Epoch 8/50
110/110 [=====] - 40s 365ms/step - loss: 0.2861 -
Epoch 9/50
110/110 [=====] - 40s 364ms/step - loss: 0.2862 -
Epoch 10/50
110/110 [=====] - 41s 370ms/step - loss: 0.2853 -
Epoch 11/50
110/110 [=====] - 40s 360ms/step - loss: 0.2836 -
Epoch 12/50
110/110 [=====] - 39s 355ms/step - loss: 0.2822 -
Epoch 13/50

```

# 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))

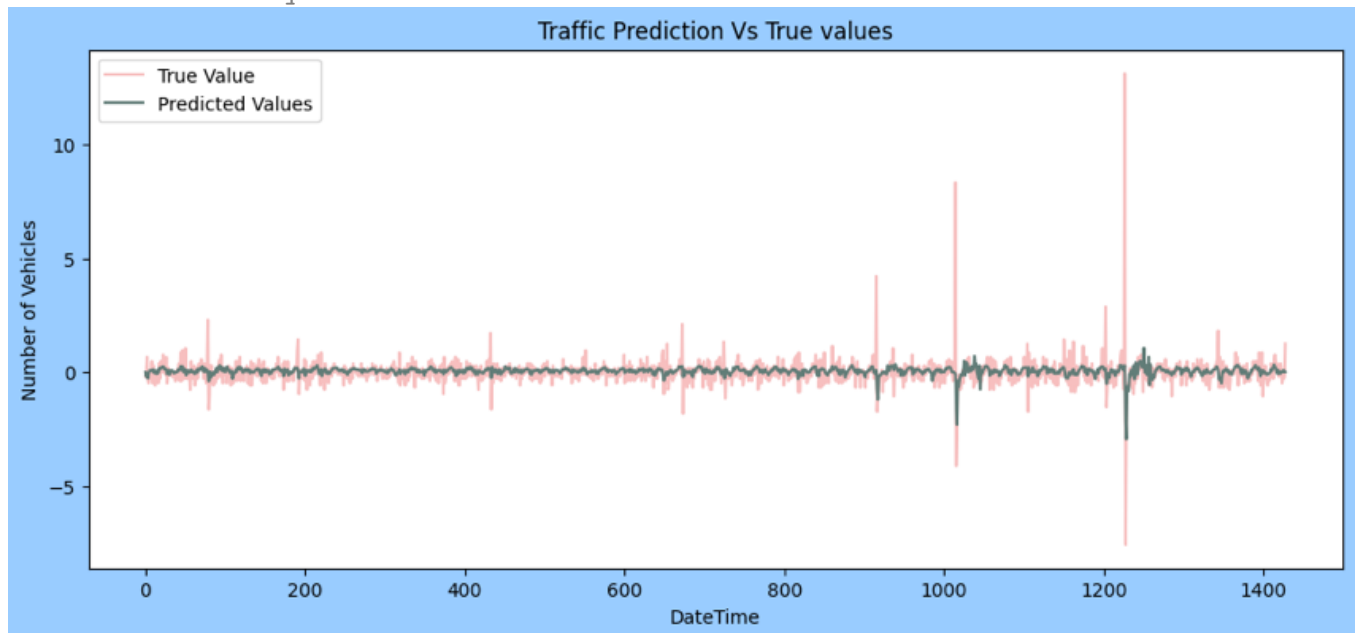
# Plot the bar graph for model names and RMSE values with transparency and light colors
colors = ['lightcoral', 'lightblue', 'lightgreen', 'lightsalmon', 'lightyellow']
alpha = 0.7
fig, ax = plt.subplots(figsize=(12, 8)) # Set the figure size (width, height)
ax.bar(model_names, rmse_values, color=colors,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

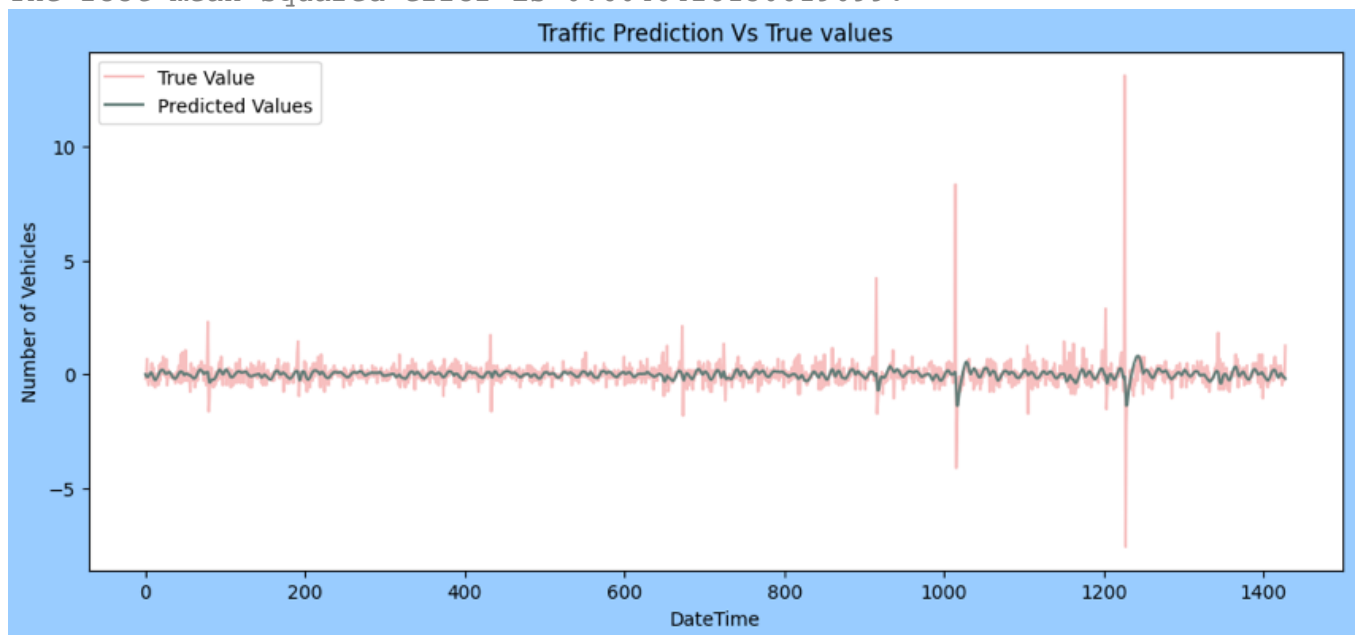
```
#####-----Custom Model-----#####
```

The root mean squared error is 0.579506020280626.



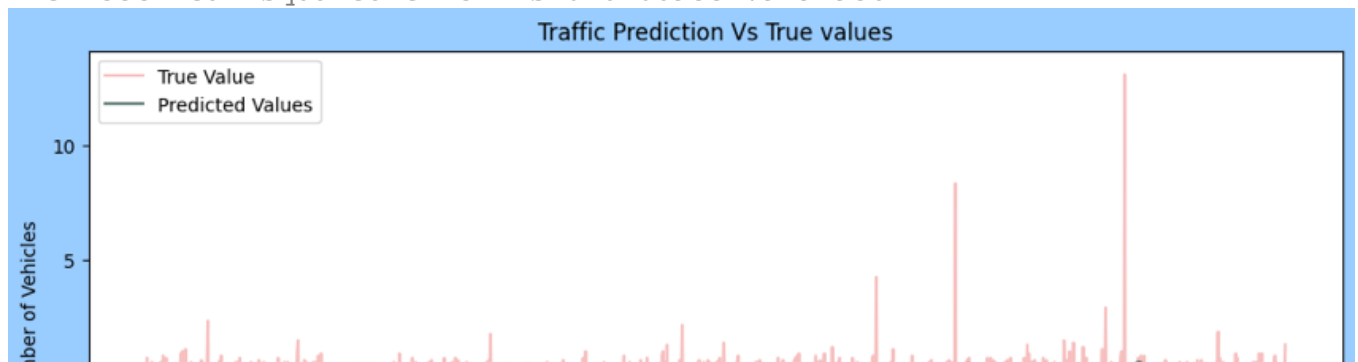
```
#####-----GRU Model-----#####
```

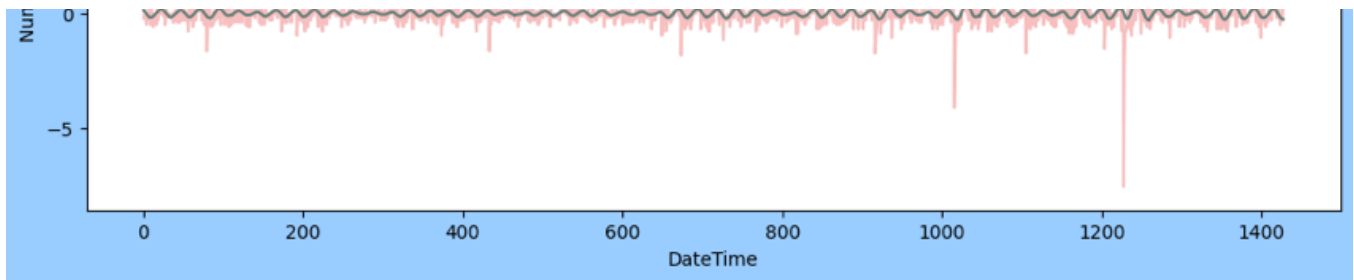
The root mean squared error is 0.6046428130619099.



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

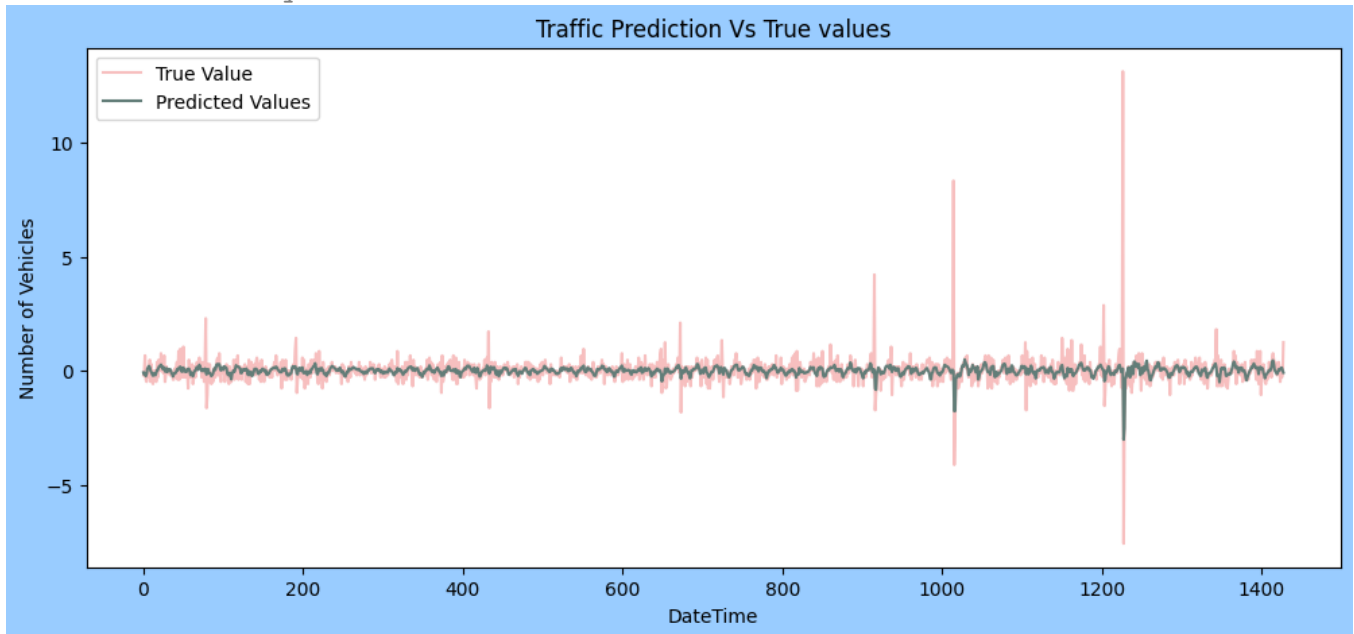
The root mean squared error is 0.6206933479192936.





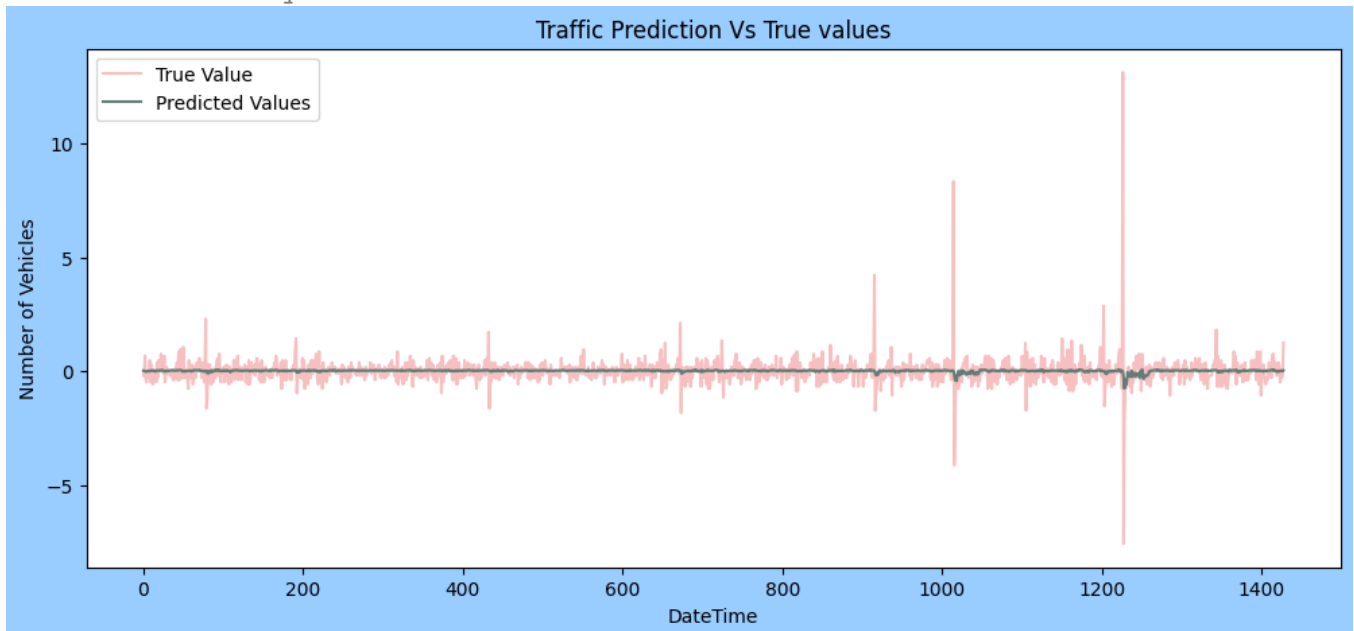
#####-----CNN Model-----#####

The root mean squared error is 0.5708027221386098.



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

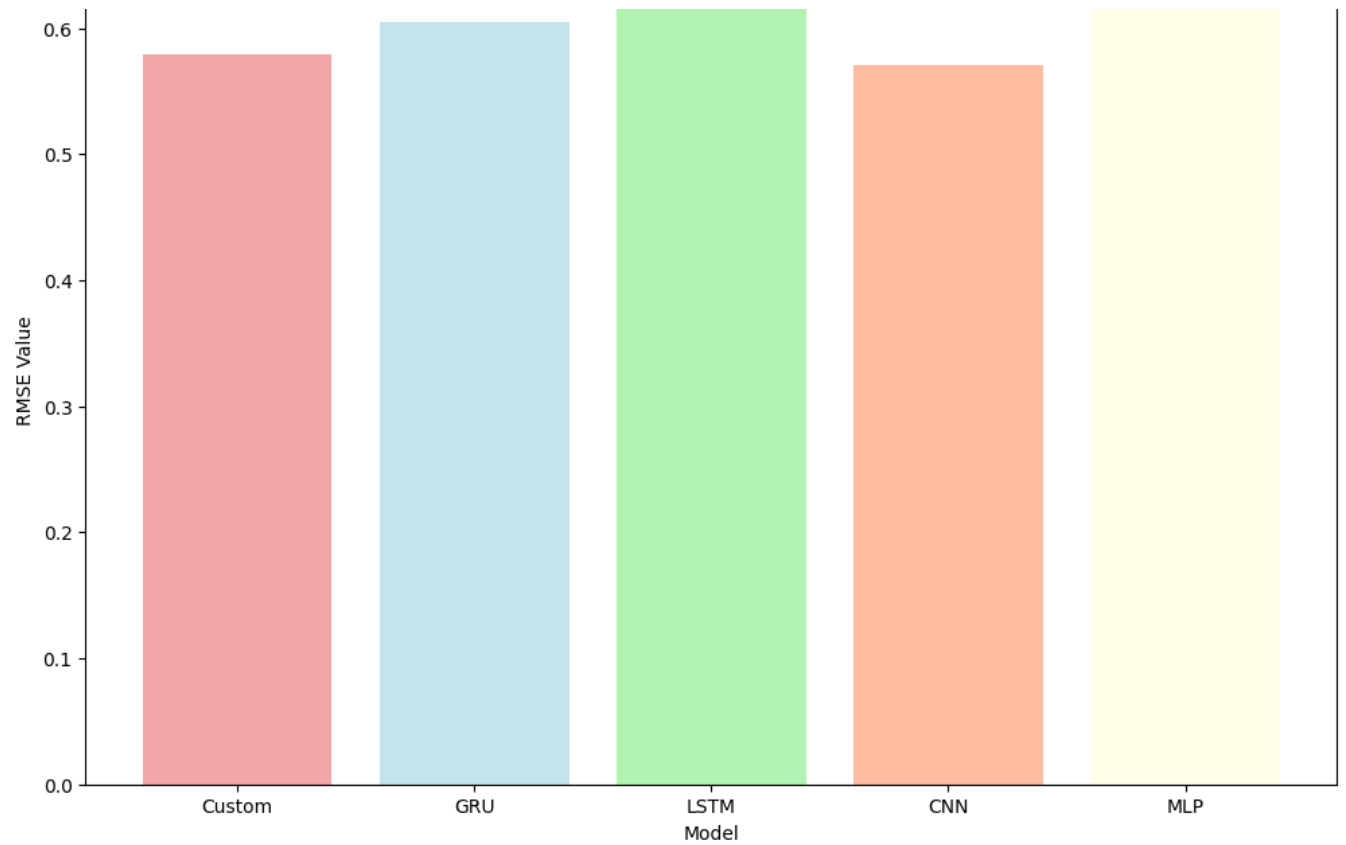
The root mean squared error is 0.6261025649204597.



**Best Model: CNN - RMSE: 0.5708027221386098**

RMSE Value for Different Models - J3





	MODEL	RMSE
0	Custom	0.579506
1	GRU	0.604643
2	LSTM	0.620693
3	CNN	0.570803
4	MLP	0.626103



```

# 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)

# 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)

```

## 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")

```

**The Root Mean Square Error serves as a somewhat subjective metric for evaluating performance. Therefore, in this project, I am also incorporating outcome plots for a more comprehensive assessment.**



## ✓ Inversing The Transformation Of Data

In this section, I will reverse the transformations applied to the datasets to eliminate seasonality and trends. This step is essential for ensuring that the predictions return to their 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 ="Original")
```

## 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_frame()
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],"Predictions 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 Anc
```

## Summary

*In this project, I trained several models including GRU, CNN, MLP, LSTM, and a custom model to predict traffic on four junctions. I employed normalization and differencing transforms to achieve stationary time series. Considering the varying trends and seasonality among the junctions, I adopted different approaches to make each stationary. The root mean squared error served as the evaluation metric for the models. Additionally, I plotted the predictions alongside the original test values. Key insights from the data analysis are as follows:*

*Junction one exhibits a more rapid increase in the number of vehicles compared to junctions two and three. However, the sparse data in junction four prevents conclusive analysis.*

*Junction one's traffic displays a stronger weekly and hourly seasonality, whereas other junctions appear more linear.*

▼ End

```
# Assuming you have a new dataset in CSV format, you can use pandas to load it
import pandas as pd

# Load the new dataset
new_dataset = pd.read_csv('/content/drive/MyDrive/DEEP LEARNING/traffic_test.csv')

# Preprocess the new dataset if necessary
# For example, you might need to perform feature scaling or normalization

# Assuming X_test_new and y_test_new represent your features and target variable
X_test_new = new_dataset.drop(columns=['Vehicles']) # Drop the target column
y_test_new = new_dataset['Vehicles']

# Now you can use your trained models to make predictions on this new dataset
# You may need to preprocess X_test_new further if it requires reshaping or any

# Use your models to make predictions
# For example, if you have a trained Custom_model, you can use it like this:
PredJ1_Custom = Custom_model(X_train, y_train, X_test_new, y_test_new)

# Evaluate the performance of your models on the new dataset using appropriate
# For example, you can calculate the RMSE for each model
RMSE_Custom_new = RMSE_Value(y_test_new, PredJ1_Custom)

# You can also visualize the predictions using your defined plotting functions
PredictionsPlot(y_test_new, PredJ1_Custom, 0)

# Repeat the above steps for each of your trained models (GRU_model, LSTM_model)
```

```
# Invoke models with test data
PredJ1_Custom = Custom_model(X_train, y_train, X_test_new, y_test_new)
PredJ1_GRU = GRU_model(X_train, y_train, X_test_new, y_test_new)
PredJ1_LSTM = LSTM_model(X_train, y_train, X_test_new, y_test_new)
PredJ1_CNN = CNN_model(X_train, y_train, X_test_new, y_test_new)
PredJ1_MLP = MLP_model(X_train, y_train, X_test_new, y_test_new)

# Evaluate predictions and visualize
RMSE_J1_Custom = RMSE_Value(y_test_new, PredJ1_Custom)
PredictionsPlot(y_test_new, PredJ1_Custom, 0)

RMSE_J1_GRU = RMSE_Value(y_test_new, PredJ1_GRU)
PredictionsPlot(y_test_new, PredJ1_GRU, 0)

RMSE_J1_LSTM = RMSE_Value(y_test_new, PredJ1_LSTM)
PredictionsPlot(y_test_new, PredJ1_LSTM, 0)

RMSE_J1_CNN = RMSE_Value(y_test_new, PredJ1_CNN)
PredictionsPlot(y_test_new, PredJ1_CNN, 0)

RMSE_J1_MLP = RMSE_Value(y_test_new, PredJ1_MLP)
PredictionsPlot(y_test_new, PredJ1_MLP, 0)

# Repeat the same process for other junctions (J2, J3, J4)
# You can reuse the above code with respective test data (X_testJ2, y_testJ2),
```

