DATS 6450
Multivariate Modelling
Professor Reza Jafari

Term Project:
Forecasting Traffic Volume

Sheldon Sebastian
SS

04/24/20

Table of Contents:

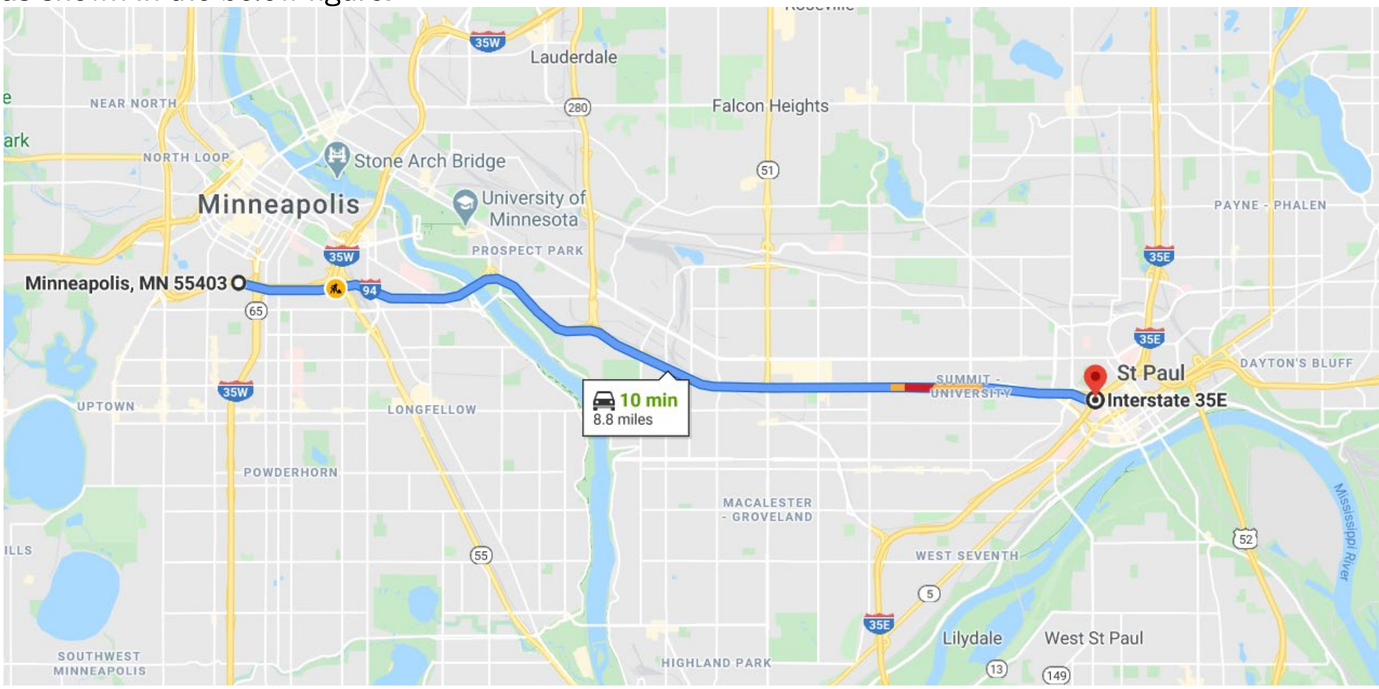
Abstract:

We are predicting the traffic volume per day for the I94 interstate. The traffic volume per day is the number of cars which use the I94 interstate between St. Paul and Minneapolis. To make accurate forecasts, 6 models Average Model, Naïve Model, Drift Model, Holt Winter Model, Multiple linear regression and ARMA were used. The performance of the all models are compared and the best performing model is recommended to forecast traffic volume.

Keywords: Forecasting, Traffic, Average model, Naïve model, Drift Model, Holt Winter, ARMA, Linear Regression

1. Introduction:

We are predicting the number of cars in a day on the I94 interstate between St. Paul and Minneapolis as shown in the below figure:



Business value of project:

1. **Avoid Traffic Congestion**: We can predict the days when there will be heavy traffic congestion and thus take contingencies to avoid them.
2. **Road Maintenance**: Using the traffic volume predictions we can estimate how long before the road needs repairs and we can schedule repairs when there is least traffic volume.

For achieving the goal of predicting traffic volume, we are considering 6 prediction models: Average, Naïve, Drift, Holt Winter, ARMA model and Multiple Linear Regression model.

In average model, all the future predictions are average of the training data.

In naïve model, we predict all the future values by taking the last value of the training dataset.

In drift model, we plot a line from the first point of the data to the last point and extend it to predict all the future values.

In the Holt Winter method, we will find whether traffic volume follows additive or multiplicative trend and then make predictions.

For the Linear Regression Model, we will scale the feature variables and perform data cleaning and then make predictions.

Finally, for ARMA model, we will estimate the order of the ARMA process using GPAC table, estimate the parameters for ARMA and check whether the residuals pass the chi square test or not.

Once all the models are created, we will compare the performance and recommend the best performing model.

2. Description of Dataset:

The dataset has hourly traffic volume from October 2012 to September 2018. Traffic volume is defined as count of cars in an hour on the interstate. As described previously, the hourly traffic volume is tracked between Minneapolis and St Paul, MN.

The dataset is sourced from the following website:
https://archive.ics.uci.edu/ml/datasets/Metro+Interstate+Traffic+Volume

The variables and their description in the dataset are as follows:
Dependent Variables:
1. traffic_volume: The numeric Hourly I-94 ATR 301 reported westbound traffic volume. This is our dependent or target variable.

Independent Variables:

1. holiday: Categorical column containing US National holidays plus regional holidays.
2. temp: Numeric Average temp in kelvin.
3. rain_1h: Numeric Amount in mm of rain that occurred in the hour.
4. snow_1h: Numeric Amount in mm of snow that occurred in the hour.
5. clouds_all: Numeric Percentage of cloud cover.
6. weather_main: Categorical Short textual description of the current weather.
7. weather_description: Categorical Longer textual description of the current weather.
8. date_time: Date Time Hour of the data collected in local CST time.

a. Resampling Data:

For computational purposes and model interpretability the hourly data was **resampled into daily data**. Also, we are focusing on traffic volume data for **September 2016 to September 2018**.

When we perform resampling the following functions were applied to the variables:
- Mean: temp, clouds_all, traffic_volume, rain_1h, snow_1h.
- First: weather_main, holiday.

After resampling the shape of the dataset is as follows:

```
(731, 7)
```

b. Summary Statistics and Data preprocessing:

The summary statistics for numeric columns are as follows:

	temp	clouds_all	traffic_volume	rain_1h	snow_1h
count	731.000000	731.000000	731.000000	731.000000	731.0
mean	281.307359	43.454909	3296.855356	0.030720	0.0
std	12.148368	27.068085	562.172038	0.215786	0.0
min	249.040000	0.000000	1139.050000	0.000000	0.0
25%	272.392500	19.666667	2883.541667	0.000000	0.0
50%	282.432182	41.500000	3478.703704	0.000000	0.0
75%	292.295417	64.594828	3729.833333	0.000000	0.0
max	302.587500	90.227273	4555.170213	3.313594	0.0

We notice that the snow_1h column has all values as zero, thus we remove that column.

The summary statistics for categorical columns are as follows:

	weather_main	holiday
count	731	22
unique	9	11
top	Clear	Veterans Day
freq	361	2

We notice that the holiday column has 22 values only, thus we replace all the other NaN values with “No Holiday” values. After replacing all the holiday NaN columns with 'No Holiday' value we get value counts for holiday column as:

No Holiday	709
Martin Luther King Jr Day	2
New Years Day	2
State Fair	2
Veterans Day	2
Thanksgiving Day	2
Independence Day	2
Washingtons Birthday	2
Christmas Day	2
Columbus Day	2
Memorial Day	2
Labor Day	2

We also notice that the weather_main column contains 9 unique values which are:

Clear	361
Clouds	132
Rain	76
Mist	57
Snow	56
Drizzle	20
Thunderstorm	14
Haze	10
Fog	5

We condense this information as follows:

- Rain additionally covers the values Drizzle, Thunderstorm
- Fog additionally covers the values Mist, Haze, Smoke

Thus, after condensing the value counts are as follows:

Clear	361
Clouds	132
Rain	110
Fog	72
Snow	56

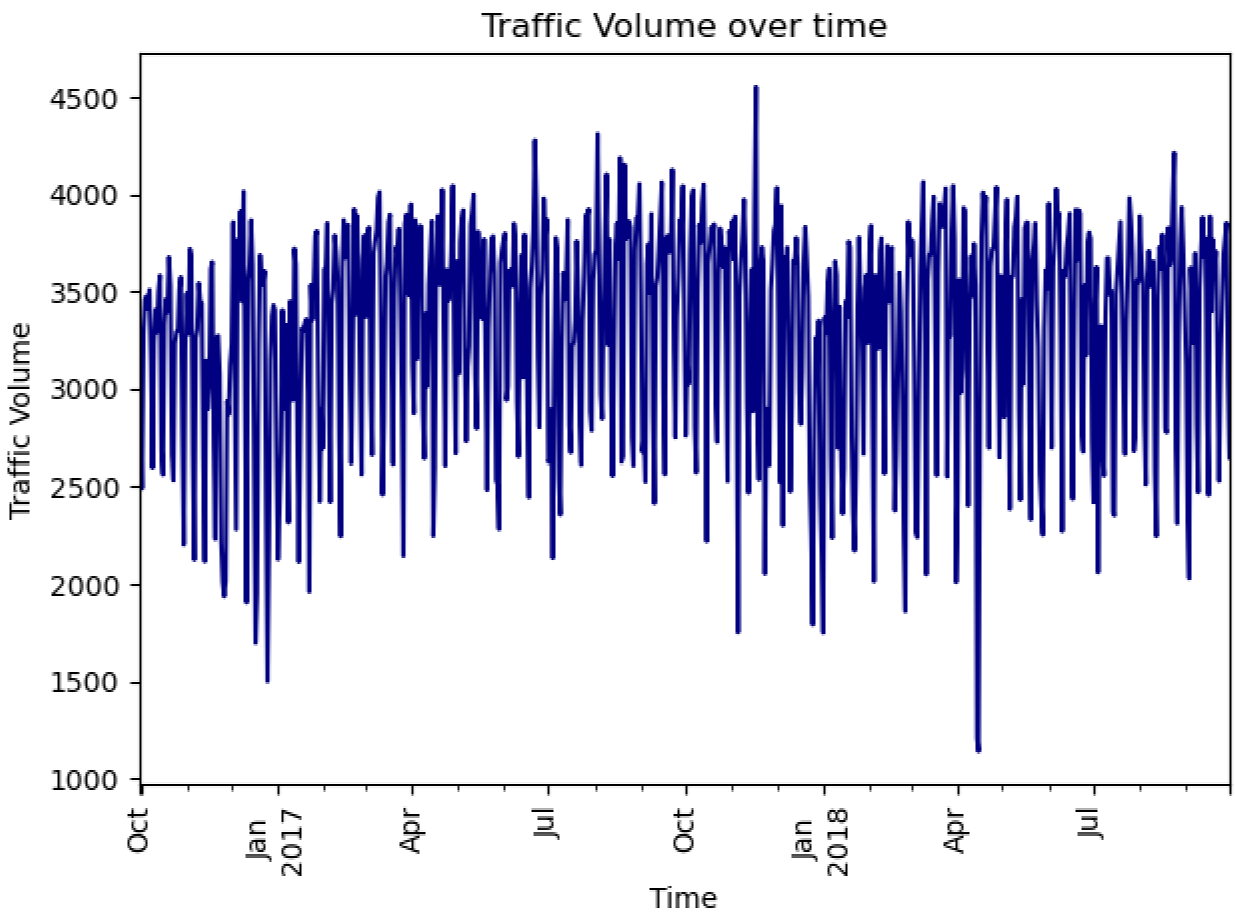
Finally, after resampling and data cleaning the column count with NaN values are:

```
temp            0
clouds_all      0
weather_main    0
traffic_volume  0
holiday         0
rain_1h         0
```

Hence, we do not need to perform any data imputation.

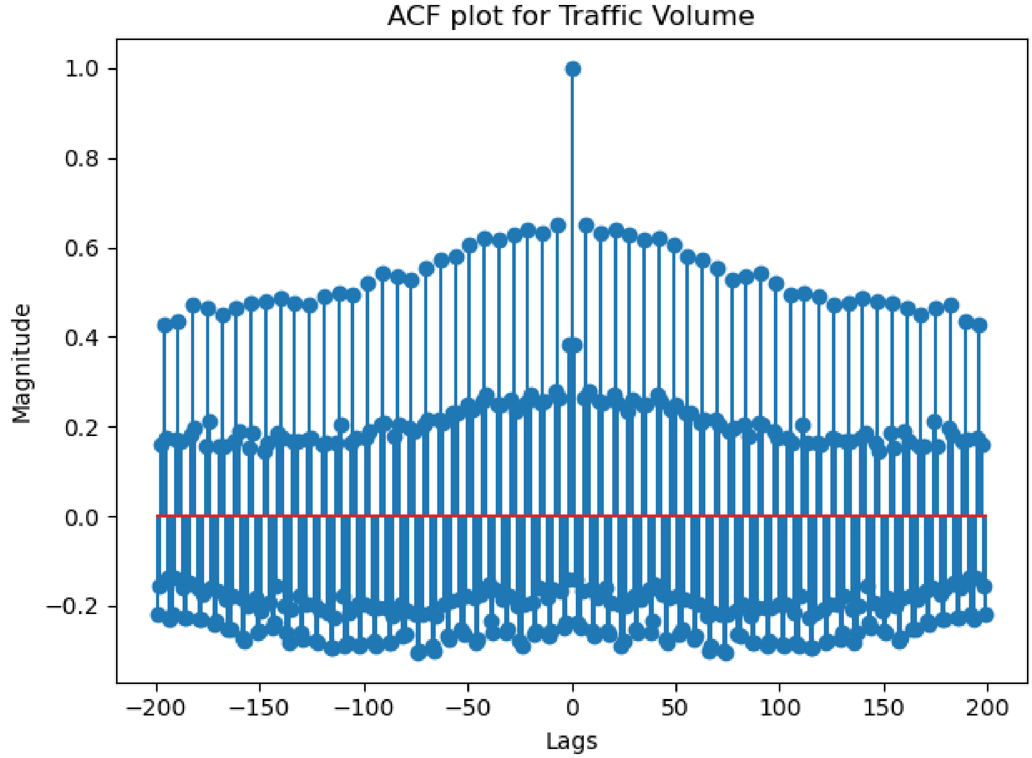
c. Traffic Volume over time:

We plot the traffic volume over time, the traffic volume data is resampled to daily data and the scope of data is from 09/2016 to 09/2018.



d. ACF of traffic volume:

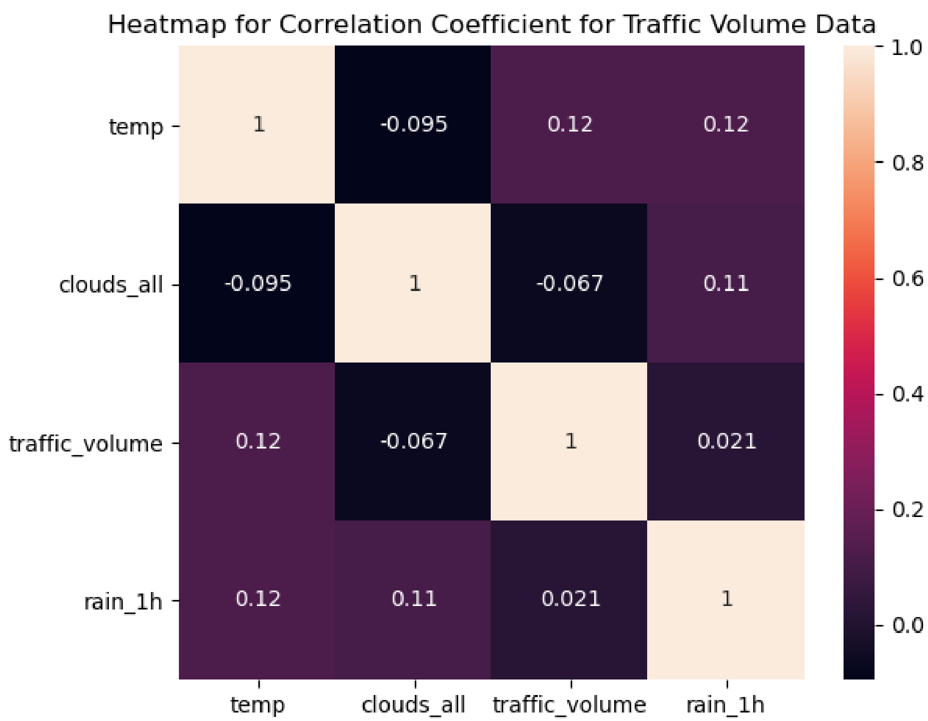
For plotting the ACF plot we have set the value of lag = 200.



We notice that the ACF values show decay at lag = 200.

e. Correlation Coefficient Matrix :

We plot the correlation coefficient matrix for the numerical columns and conclude that there is no multicollinearity. We will explore the correlation coefficient matrix again including the categorical columns in the Linear Model Section 6.b.



f. Train set (80%) and test set (20%) split:

We split the resampled data into train and test datasets. The dimension for train dataset is :

```
(584, 6)
```

The dimension for test dataset is:

```
(147, 6)
```

3.Stationarity:

To check if traffic volume is stationary or not, we perform the ADF test.

```
ADF Test for traffic_volume
ADF Statistic: -3.115765
p-value: 0.025408
Critical Values:
    1%: -3.440
    5%: -2.866
    10%: -2.569

Since p-value is less than 0.05, reject null hypothesis thus time series data is Stationary
```

From ADF test we conclude that traffic volume is stationary.

4.Average Model:

We compute the mean of training data and perform h step predictions to match the size of the test data.

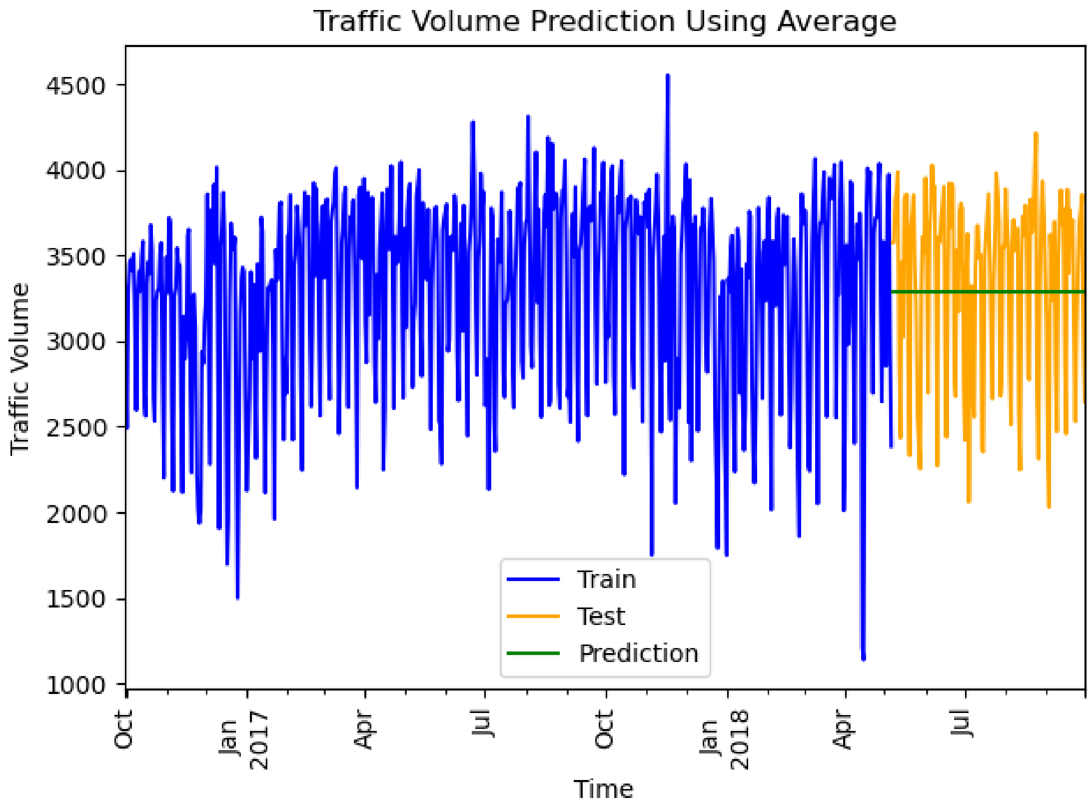
RMSE: The RMSE value is **531.918**

MSE: The MSE value is **282937.565**

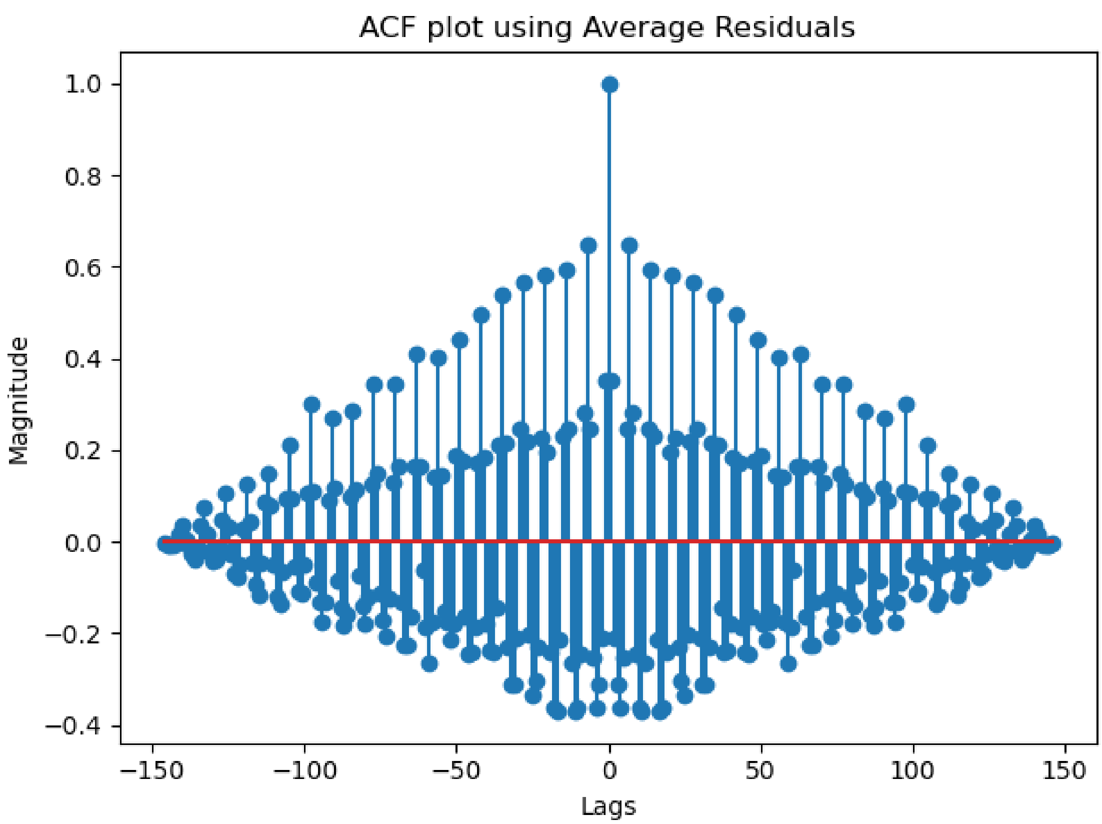
Residual Variance: The residual of variance is **279859.840**

Residual Mean: The residual of mean is **55.477**

Plot of Prediction: The plot of forecasted values with the actual value is shown below:



Plot of ACF: The ACF of residuals is as follows:



We conclude that the ACF plot does not resemble white noise.

5. Naïve Model:

We find the last sample of training data and perform h step predictions to match the size of the test data.

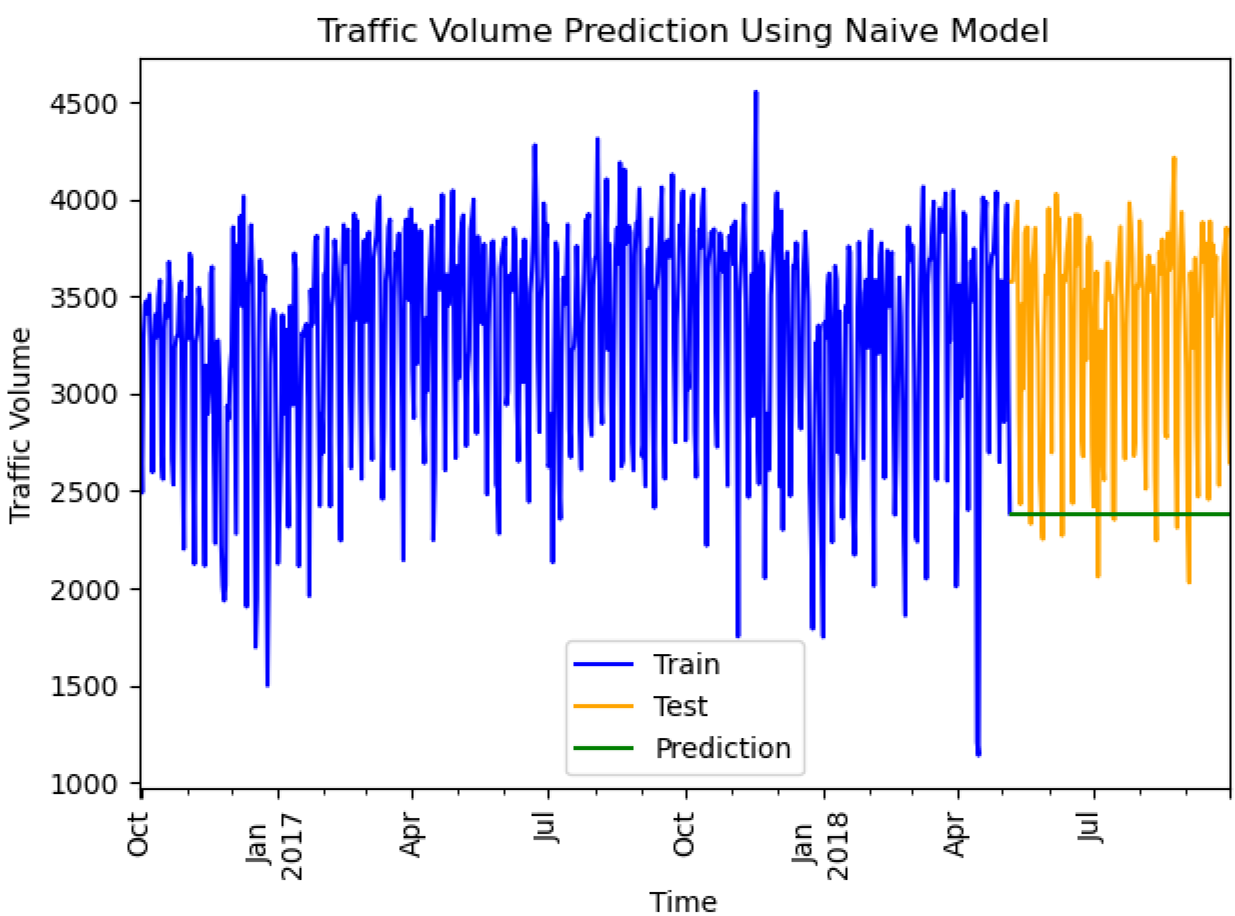
RMSE: The RMSE value is **1091.679**

MSE: The MSE value is **1191763.077**

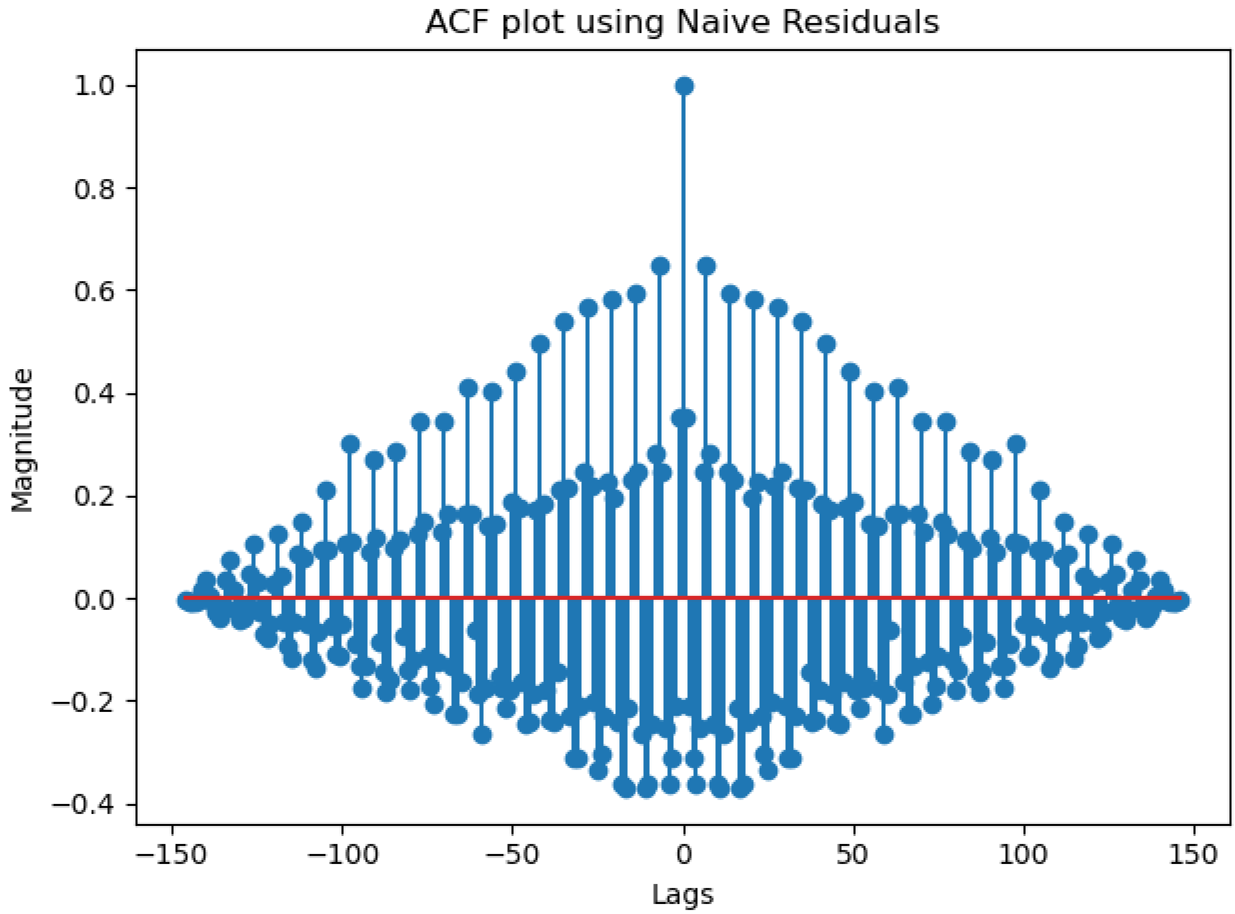
Residual Variance: The residual of variance is **279859.840**

Residual Mean: The residual of mean is **954.936**

Plot of Prediction: The plot of forecasted values with the actual value is shown below:



Plot of ACF: The ACF of residuals is as follows:



We conclude that the ACF plot does not resemble white noise.

6. Drift Model:

The performance measures for the Drift Model are as follows:

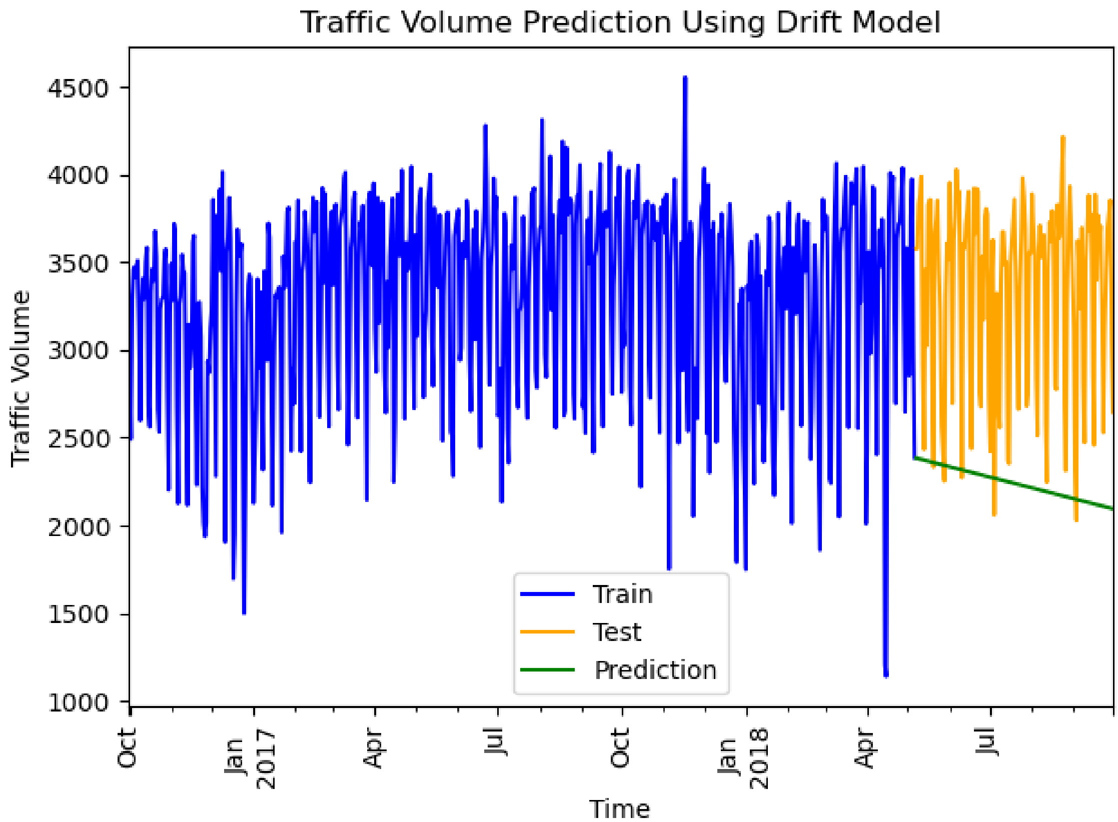
RMSE: The RMSE value is **1223.722**

MSE: The MSE value is **1497497.754**

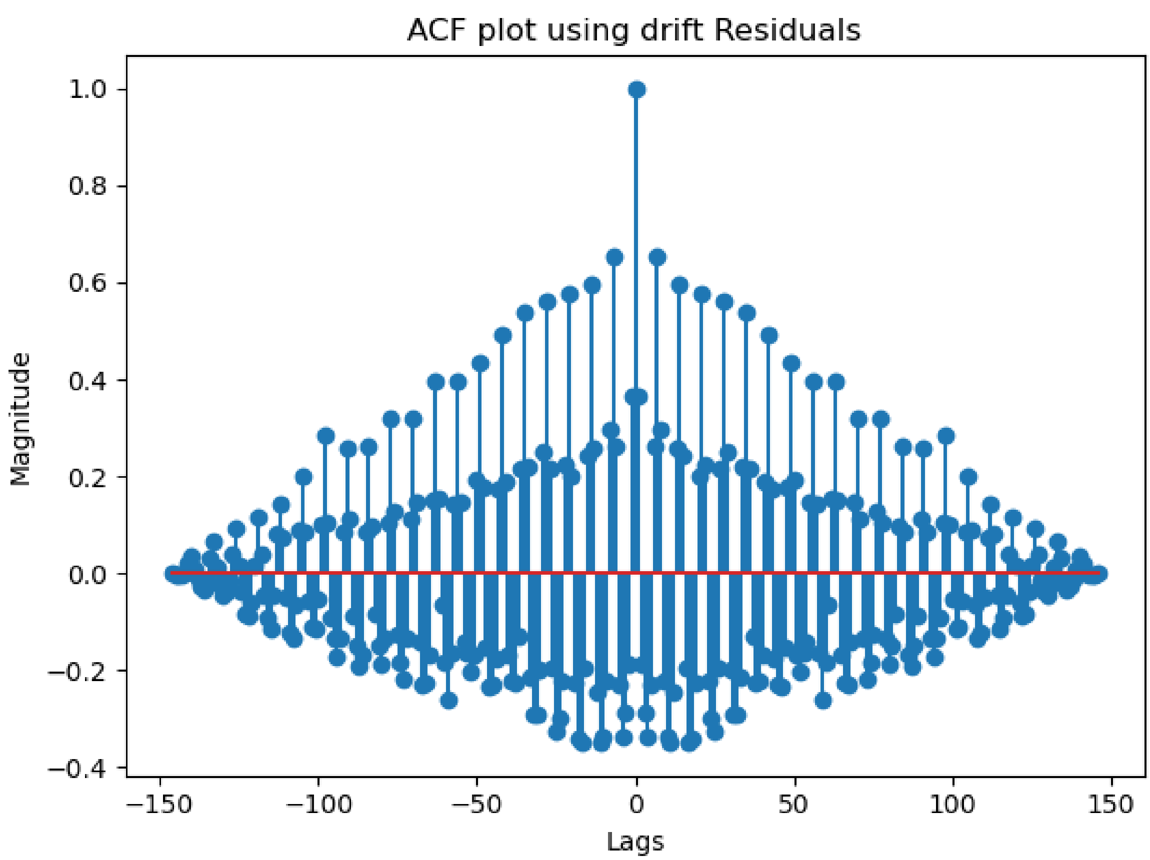
Residual Variance: The residual of variance is **284349.208**

Residual Mean: The residual of mean is **1101.430**

Plot of Prediction: The plot of forecasted values with the actual value is shown below:



Plot of ACF: The ACF of residuals is as follows:



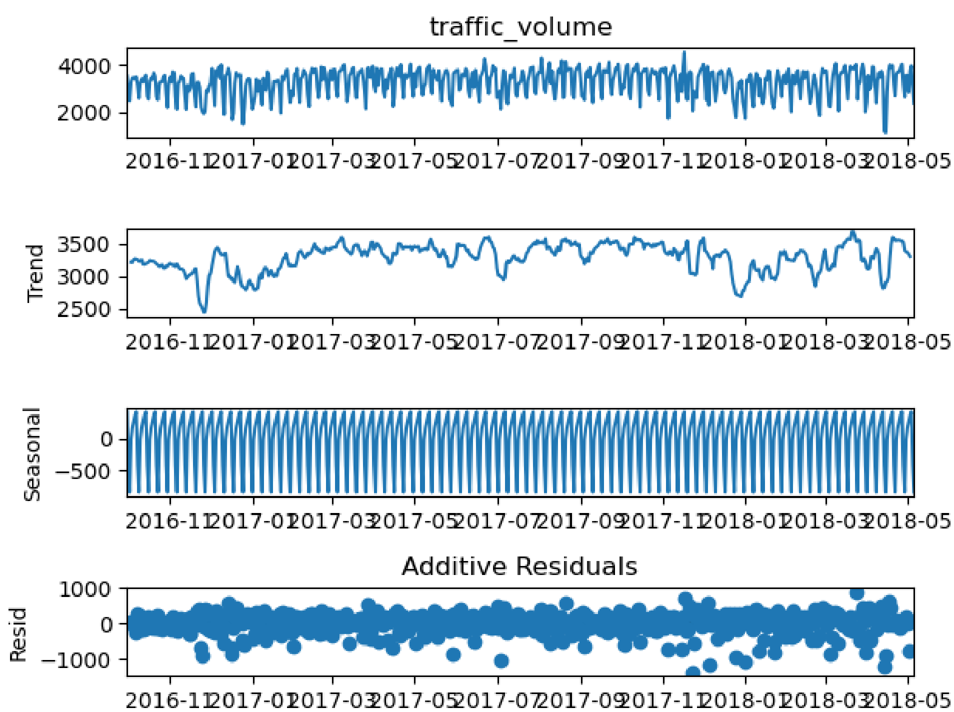
We conclude that the ACF plot does not resemble white noise.

7. Time Series Decomposition:

We will decompose traffic volume to comprehend whether trend and seasonality are additive or multiplicative.

The Multiplicative time series decomposition plot is shown below:

The Additive time series decomposition plot is shown below:



We notice that the additive residuals have high variance and it ranges from +1000 to -1000, whereas all the multiplicative residuals are close to one.

Thus, the multiplicative decomposition best represents the traffic volume data and we see a strong seasonality component but there is no trend visible.

8. Holt-Winters Method:

Based on the time series decomposition we will configure the Holt-Winters parameters for predicting the test data. We will set the **seasonality to be multiplicative** and set **trend to be None**.

The performance metrics for Holt Winter model is given below:

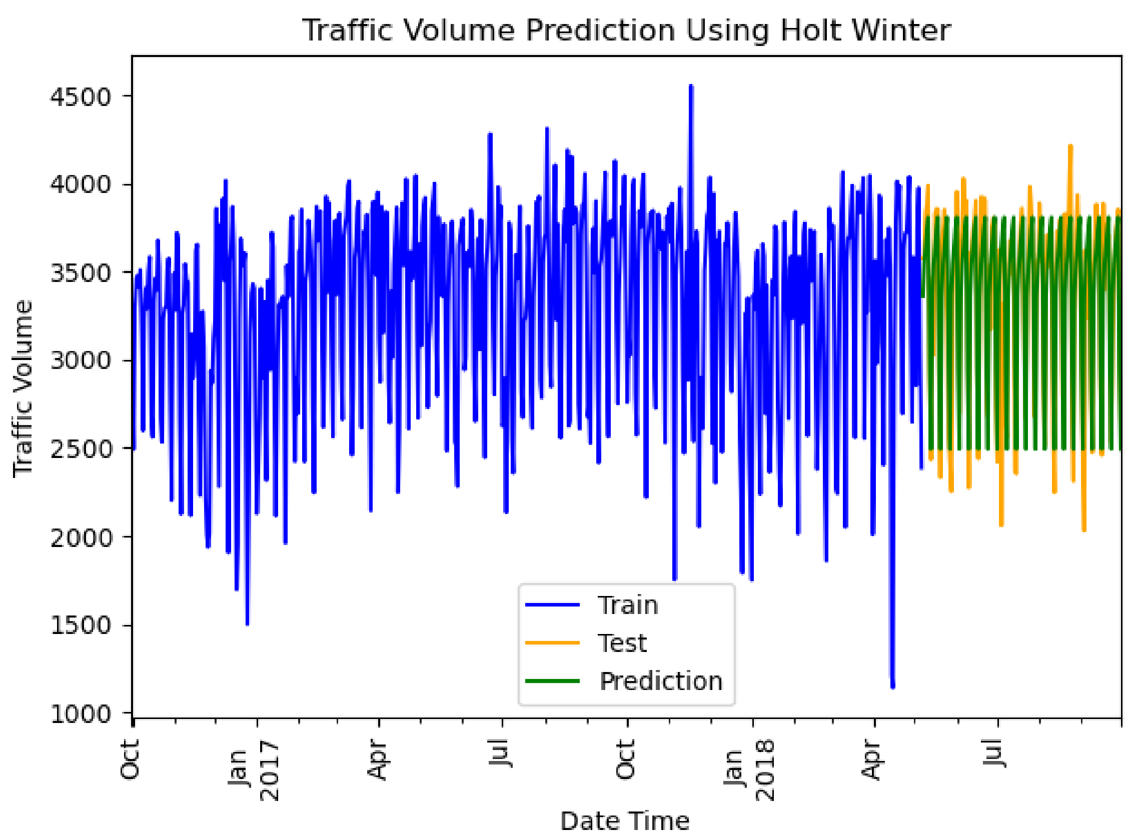
MSE: The MSE of residuals is **84690.827**

RMSE: The RMSE of residuals is **291.017**

Variance: The variance of residuals is **84197.577**

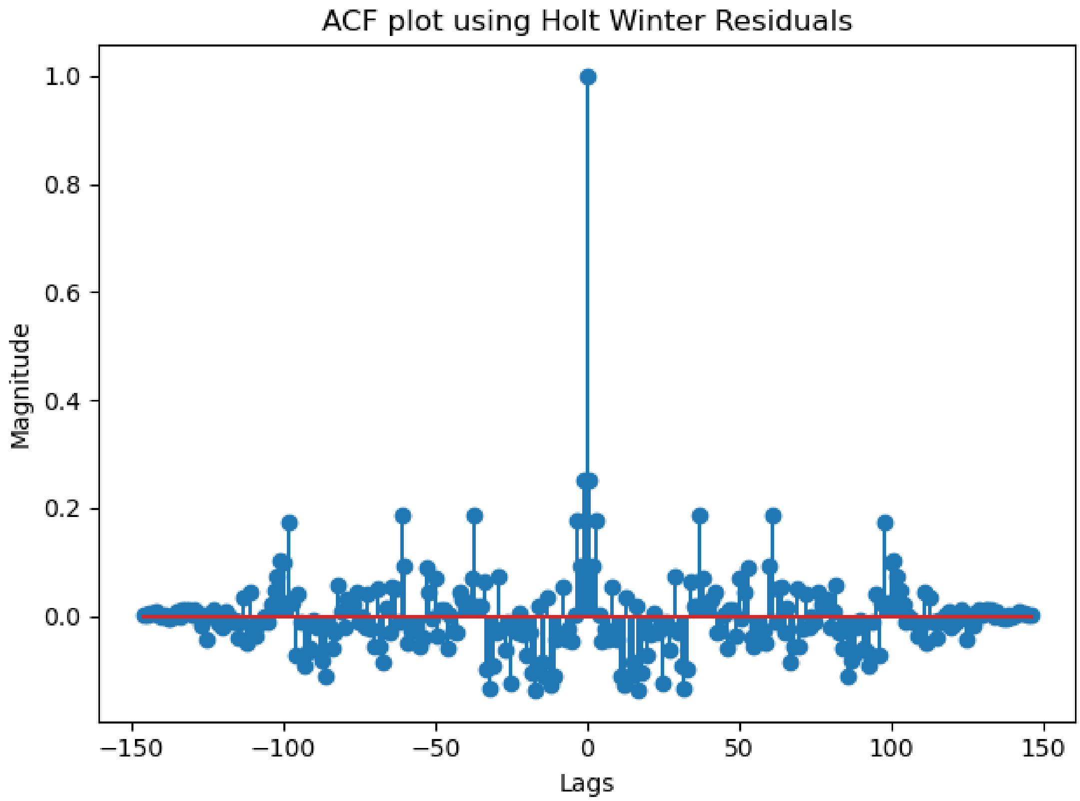
Mean: The mean of residuals is - **-22.209**

The plot for Holt Winter model prediction along with actual predictions is shown below:



We notice from above plot that the Holt Winter model predictions are close to the actual values.

ACF of residuals: We notice that the ACF plot for Holt Winter method resembles the white noise.



9. Multiple Linear Regression:

a. Linear Model with all Features:

We will now perform multiple linear regression and for this we need to **scale** the data and convert the **categorical** columns into numerical columns.

We **scale** the feature variables using sklearn.preprocessing's MinMaxScaler function and then compute the MSE values for the test data and the predicted values.

We convert the **categorical** values into numerical values by using pandas get_dummies(...) function.

The summary of linear model with all variables is:

```
                            OLS Regression Results
==============================================================================
Dep. Variable:         traffic_volume   R-squared:                       0.124
Model:                            OLS   Adj. R-squared:                  0.096
Method:                 Least Squares   F-statistic:                     4.425
Date:                Tue, 21 Apr 2020   Prob (F-statistic):           5.08e-09
Time:                        11:46:59   Log-Likelihood:                -4495.2
No. Observations:                 584   AIC:                             9028.
Df Residuals:                     565   BIC:                             9111.
Df Model:                          18
Covariance Type:            nonrobust
=====================================================================================================
                                        coef    std err          t      P>|t|      [0.025      0.975]
-----------------------------------------------------------------------------------------------------
clouds_all                          -61.4395     84.584     -0.726      0.468    -227.578     104.699
holiday_Christmas Day               129.9489    372.878      0.349      0.728    -602.448     862.346
holiday_Columbus Day               1295.8447    372.887      3.475      0.001     563.430    2028.259
holiday_Independence Day              1.4496    520.315      0.003      0.998   -1020.538    1023.437
holiday_Labor Day                   437.0476    519.342      0.842      0.400    -583.030    1457.125
holiday_Martin Luther King Jr Day   948.4226    373.371      2.540      0.011     215.058    1681.787
holiday_Memorial Day                182.3774    521.083      0.350      0.726    -841.118    1205.873
holiday_New Years Day               272.5889    371.823      0.733      0.464    -457.736    1002.914
holiday_No Holiday                 1323.4723     92.532     14.303      0.000    1141.724    1505.221
holiday_State Fair                 1918.1943    520.430      3.686      0.000     895.980    2940.409
holiday_Thanksgiving Day            175.2211    372.322      0.471      0.638    -556.084     906.526
holiday_Veterans Day               1567.7924    371.292      4.223      0.000     838.511    2297.074
holiday_Washingtons Birthday        844.9261    375.600      2.250      0.025     107.183    1582.669
rain_1h                             188.9808    510.526      0.370      0.711    -813.779    1191.741
temp                                255.4504    111.651      2.288      0.023      36.149     474.752
weather_main_Clear                 1911.3618    103.585     18.452      0.000    1707.903    2114.821
weather_main_Clouds                1964.4687    115.367     17.028      0.000    1737.869    2191.069
weather_main_Fog                   1835.0392    126.200     14.541      0.000    1587.160    2082.918
weather_main_Rain                  1777.7041    131.567     13.512      0.000    1519.284    2036.125
weather_main_Snow                  1608.7123    122.898     13.090      0.000    1367.319    1850.105
==============================================================================
```

Conclusions based on above summary:

F-test : The F-test passes since the Prob(F-statistics) is less than 0.05 and thus our model performs better than null model.

AIC: The AIC value is 9028

BIC: The BIC value is 9111

T-test: There are variables which fail the t-test, to fix this we drop the variables which fail the t-test.

b. Linear Model after Feature Selection:

Since there are variables which fail the t-test when we use all variables model, we remove these variables and repeat the linear model process until all the variables pass the t-test.

The summary of linear model after feature selection is:

```
                            OLS Regression Results
==============================================================================
Dep. Variable:          traffic_volume   R-squared:                       0.115
Model:                             OLS   Adj. R-squared:                  0.101
Method:                  Least Squares   F-statistic:                     8.276
Date:                 Tue, 21 Apr 2020   Prob (F-statistic):           1.24e-11
Time:                         11:46:59   Log-Likelihood:                -4498.1
No. Observations:                  584   AIC:                             9016.
Df Residuals:                      574   BIC:                             9060.
Df Model:                            9
Covariance Type:             nonrobust
=========================================================================================
                            coef    std err          t      P>|t|      [0.025      0.975]
-----------------------------------------------------------------------------------------
holiday_Columbus Day    898.9201    411.956      2.182      0.030      89.796    1708.044
holiday_No Holiday      921.0062    151.983      6.060      0.000     622.496    1219.516
holiday_State Fair     1524.0124    564.420      2.700      0.007     415.432    2632.593
holiday_Veterans Day   1174.0708    411.281      2.855      0.004     366.272    1981.870
temp                    257.2118    109.683      2.345      0.019      41.783     472.641
weather_main_Clear     2288.5133    161.279     14.190      0.000    1971.745    2605.282
weather_main_Clouds    2337.3717    164.616     14.199      0.000    2014.049    2660.694
weather_main_Fog       2207.2229    172.184     12.819      0.000    1869.035    2545.411
weather_main_Rain      2133.5156    175.297     12.171      0.000    1789.213    2477.818
weather_main_Snow      1974.6909    165.619     11.923      0.000    1649.397    2299.985
==============================================================================
```

Conclusions based on above model:

F-test: The Prob(F-statistics) < 0.05 thus our model performs better than the null model and it passes the F-test.

AIC: The AIC value is 9016 which is lower than all variables model.

BIC: The BIC value is 9060 which is lower than all variables model.

T-test: The t-test passes for all variables since the P(t) < 0.05 for all variables.

R-Squared: The R-squared value is 0.115

Adjusted R-squared: The adjusted R-squared value is 0.101 which is better than all variables model.

MSE: The MSE of residuals is **256424.508**

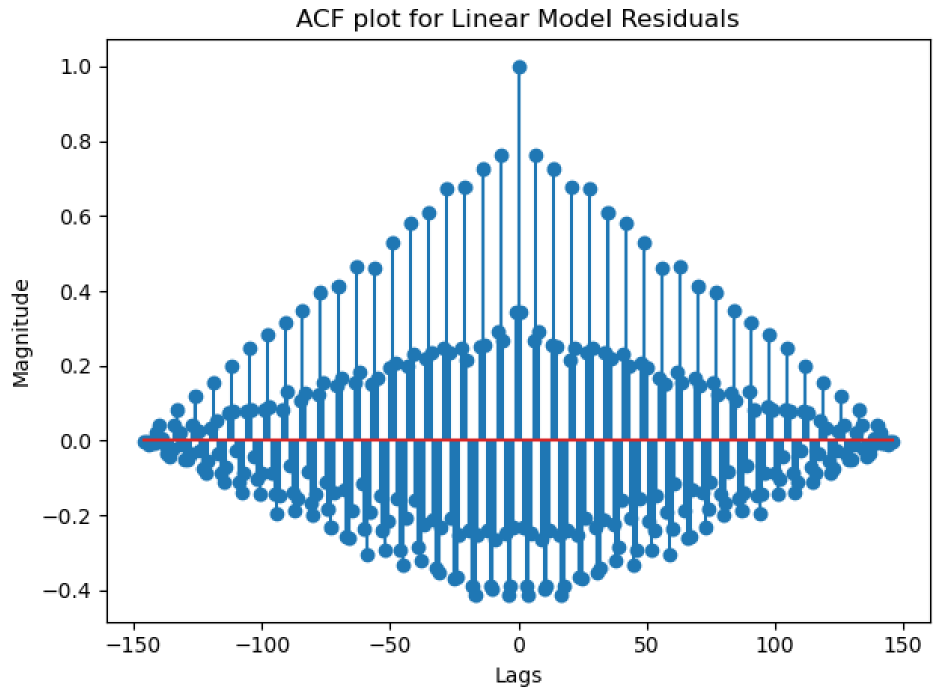
RMSE: The RMSE of residuals is **506.384**

Variance: The variance of residuals is **255166.372**

Mean: The mean of residuals is **-35.470**

ACF of residuals:

We observe the residuals are decaying in ACF plot, but they do not resemble a white noise.



Q value:

The Q value of residuals of Linear Model is: **1333.365**

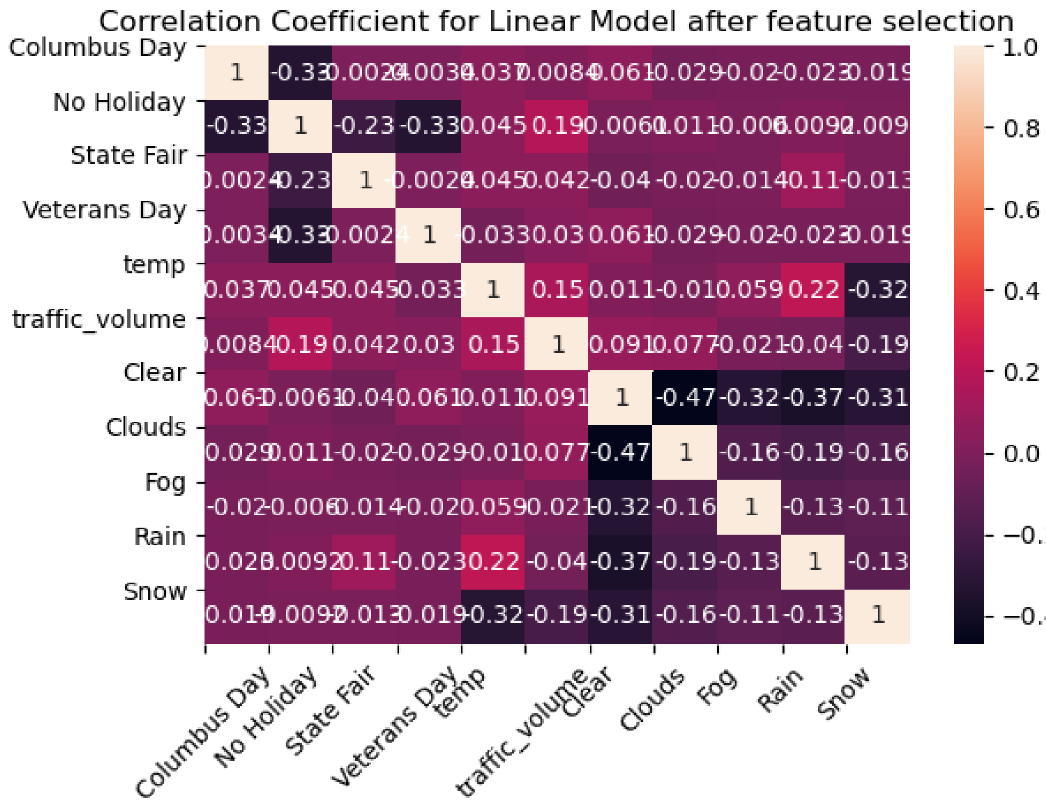
Correlation Coefficient Matrix:

The correlation coefficient matrix includes only those variables which have been used for final linear model. After feature selection the final independent features are:

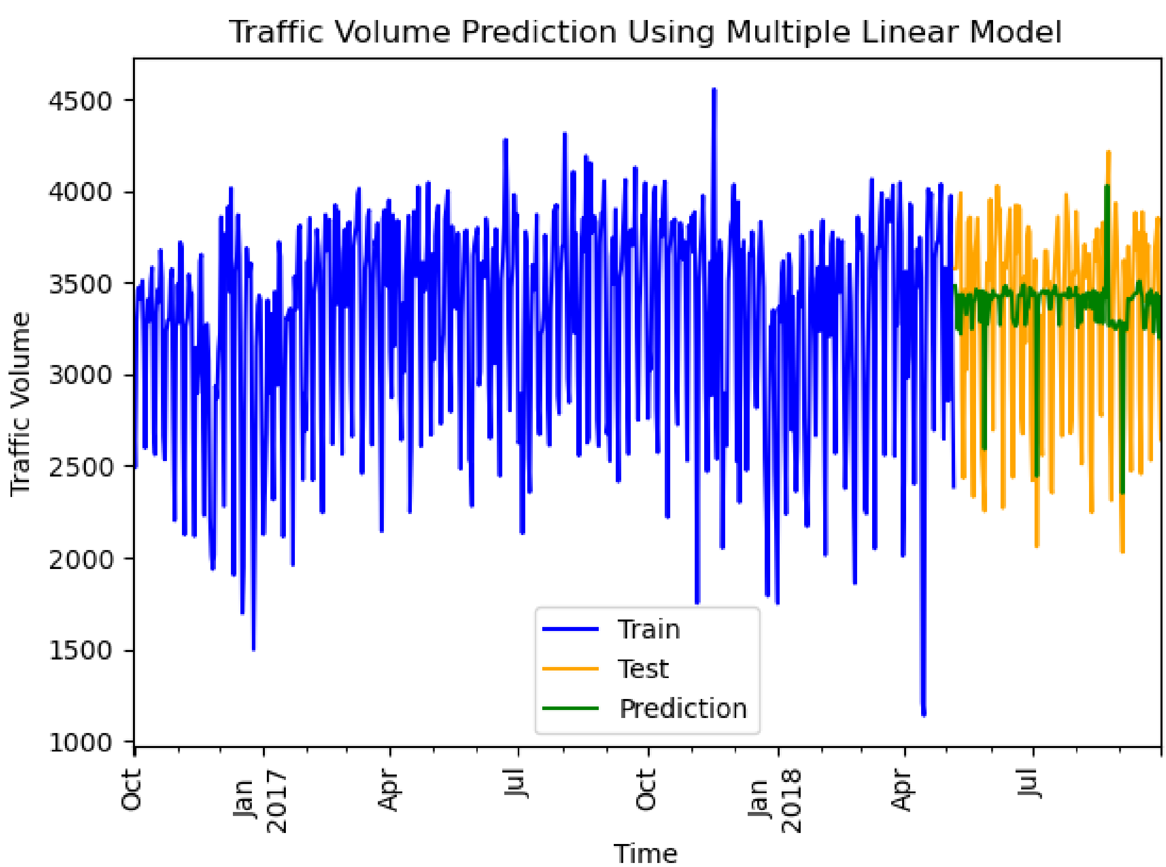
holiday_Columbus Day, holiday_No Holiday, holiday_State Fair, holiday_Veterans Day, temp, weather_main_Clear, weather_main_Clouds, weather_main_Fog, weather_main_Rain, and weather_main_Snow.

We notice there is no strong relationship between the variables and thus no multicollinearity is present.

The correlation coefficient matrix is as follows:



The plot for linear model prediction along with actual predictions is shown below:



10. ARMA Model:

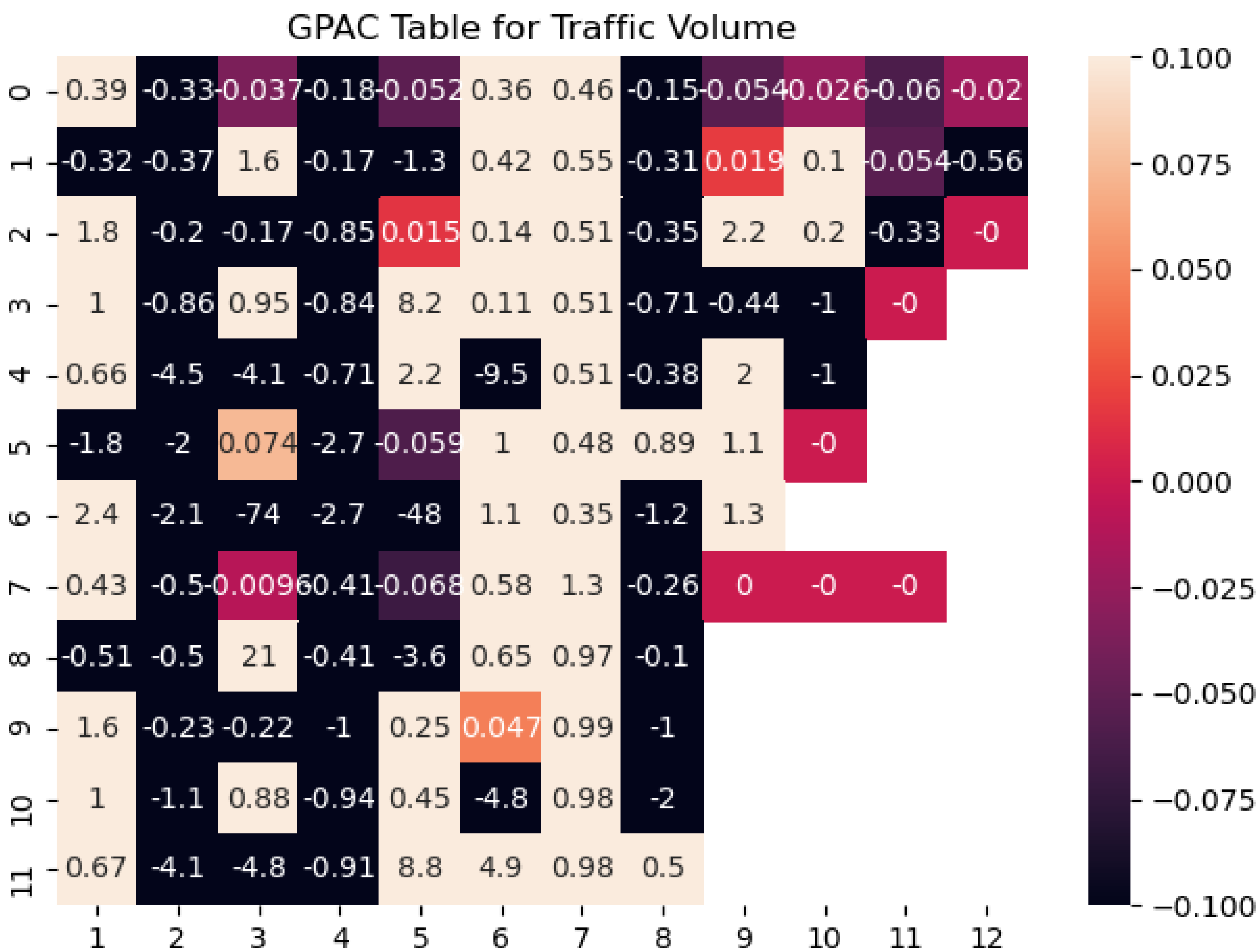
We will now predict the test data using the ARMA process. For this we will create the GPAC table and then find potential order of the ARMA process. After we find the order of ARMA process we will estimate its parameters.

We computed the mean of training data and subtracted it from the training data. This is done to relax the ARMA constraint. Since the ARMA model does not include an intercept, it might be a challenge to fit data with non-zero mean.

Once the order and parameters are estimated we will forecast the values and add the mean of training data. After adding mean, we check whether the residuals of forecasted and actual values are significant or not using the chi squared diagnostic test.

a. GPAC Table:

The GPAC table with j=12, and k=12 is shown below:



From the GPAC table we consider the following orders for ARMA parameter estimation.

	1	2	3	4	5	6	7	8	9	10	11	12
0	0.39100	-0.32803	-0.03682	-0.18165	-0.05222	0.36151	0.46031	-0.15261	-0.05415	-0.02552	-0.06009	-0.01996
1	-0.31969	-0.36721	1.57930	-0.17137	-1.30602	0.41931	0.55477	-0.31260	0.01869	0.10204	-0.05357	-0.55556
2	1.76800	-0.20306	-0.16998	-0.84983	0.01496	0.14300	0.50638	-0.34625	2.25000	0.20000	-0.33333	-0.00000
3	1.01810	-0.86486	0.95419	-0.83737	8.22222	0.10772	0.51498	-0.70896	-0.44444	-1.00000	-0.00000	NaN
4	0.65778	-4.52455	-4.06623	-0.70682	2.18919	-9.50000	0.51247	-0.37895	2.00000	-1.00000	NaN	NaN
5	-1.77703	-2.02319	0.07410	-2.67107	-0.05864	1.02456	0.47748	0.88889	1.12500	-0.00000	NaN	NaN
6	2.43726	-2.05907	-74.12637	-2.66469	-47.97368	1.05479	0.35094	-1.18750	1.33333	inf	inf	NaN
7	0.43370	-0.49830	-0.00964	-0.40720	-0.06802	0.58442	1.29032	-0.26316	0.00000	-0.00000	-0.00000	NaN
8	-0.51079	-0.50466	21.04615	-0.40565	-3.56452	0.65278	0.96667	-0.10000	NaN	NaN	NaN	NaN
9	1.64085	-0.23287	-0.21930	-1.02472	0.24887	0.04681	0.99138	-1.00000	NaN	NaN	NaN	NaN
10	1.04292	-1.06522	0.88000	-0.94408	0.45455	-4.81818	0.98261	-2.00000	NaN	NaN	NaN	NaN
11	0.67078	-4.14428	-4.84280	-0.90941	8.78000	4.88679	0.98230	0.50000	NaN	NaN	NaN	NaN

(n_a, n_b) = [(2, 5), (2, 7), (4, 0), (4, 2), (4, 5), (4, 7), (6, 5), (10, 3)]

We noticed that **none of the identified ARMA order from the GPAC table pass the chi squared test.**

Thus, we try for **all possible combinations of orders from the GPAC table in a brute force manner;** the **ARMA(4,6)** passes the Chi Square test, but shows no pattern in GPAC table; this might be possible since we have only **584 samples** in the training data.

b. Chi Square Test:

After trying a brute force approach for all possible order combinations for GPAC table, the ARMA(4,6) passes the chi square test. And as mentioned previously it shows no pattern in the GPAC table, a possible reason could be the small training size of 584 data points.

c. Parameter Estimation:

The estimated parameters based on n_a=4 and n_b=6 is:

```
ar.L1.traffic_volume     0.8028
ar.L2.traffic_volume    -1.4451
ar.L3.traffic_volume     0.8026
ar.L4.traffic_volume    -0.9998
ma.L1.traffic_volume    -0.4844
ma.L2.traffic_volume     1.3628
ma.L3.traffic_volume    -0.4779
ma.L4.traffic_volume     0.9992
ma.L5.traffic_volume     0.1718
ma.L6.traffic_volume     0.1858
```

d. Summary of ARMA(4,6) model:

The summary of ARMA(4,6) model is as follows:

```
                              ARMA Model Results
==============================================================================
Dep. Variable:          traffic_volume   No. Observations:                  584
Model:                      ARMA(4, 6)   Log Likelihood               -4251.314
Method:                        css-mle   S.D. of innovations            345.904
Date:                 Fri, 24 Apr 2020   AIC                           8524.628
Time:                         13:15:06   BIC                           8572.697
Sample:                     09-30-2016   HQIC                          8543.363
                          - 05-06-2018
========================================================================================
                           coef    std err          z      P>|z|      [0.025      0.975]
----------------------------------------------------------------------------------------
ar.L1.traffic_volume     0.8028      0.000   2081.965      0.000       0.802       0.804
ar.L2.traffic_volume    -1.4451   4.46e-06  -3.24e+05      0.000      -1.445      -1.445
ar.L3.traffic_volume     0.8026      0.000   2081.809      0.000       0.802       0.803
ar.L4.traffic_volume    -0.9998   2.06e-07  -4.85e+06      0.000      -1.000      -1.000
ma.L1.traffic_volume    -0.4844      0.046    -10.593      0.000      -0.574      -0.395
ma.L2.traffic_volume     1.3628      0.048     28.136      0.000       1.268       1.458
ma.L3.traffic_volume    -0.4779      0.060     -7.966      0.000      -0.595      -0.360
ma.L4.traffic_volume     0.9992      0.070     14.318      0.000       0.862       1.136
ma.L5.traffic_volume     0.1718      0.047      3.666      0.000       0.080       0.264
ma.L6.traffic_volume     0.1858      0.040      4.634      0.000       0.107       0.264
                                    Roots
=============================================================================
                  Real          Imaginary           Modulus         Frequency
-----------------------------------------------------------------------------
AR.1            0.6237           -0.7816j            1.0000           -0.1428
AR.2            0.6237           +0.7816j            1.0000            0.1428
AR.3           -0.2224           -0.9750j            1.0001           -0.2857
AR.4           -0.2224           +0.9750j            1.0001            0.2857
MA.1            0.6246           -0.7810j            1.0001           -0.1426
MA.2            0.6246           +0.7810j            1.0001            0.1426
MA.3           -0.2225           -0.9818j            1.0066           -0.2855
MA.4           -0.2225           +0.9818j            1.0066            0.2855
MA.5           -0.8645           -2.1361j            2.3045           -0.3112
MA.6           -0.8645           +2.1361j            2.3045            0.3112
-----------------------------------------------------------------------------
```

e. Simplification of Model:

We will now simplify the ARMA(4,6) model by checking if zeros are included in confidence interval or not.

The confidence interval for the parameters are as follows:

	coef	[0.025	0.975]
ar.L1.traffic_volume	0.8028	0.802	0.804
ar.L2.traffic_volume	-1.4451	-1.445	-1.445
ar.L3.traffic_volume	0.8026	0.802	0.803
ar.L4.traffic_volume	-0.9998	-1.000	-1.000
ma.L1.traffic_volume	-0.4844	-0.574	-0.395
ma.L2.traffic_volume	1.3628	1.268	1.458
ma.L3.traffic_volume	-0.4779	-0.595	-0.360
ma.L4.traffic_volume	0.9992	0.862	1.136
ma.L5.traffic_volume	0.1718	0.080	0.264
ma.L6.traffic_volume	0.1858	0.107	0.264

We notice there are no zeros in the confidence interval band. Thus, no simplification needed.

We will also simplify model based on zero/pole cancellation by checking the roots of numerator and denominator. The roots of the AR and MA process are:

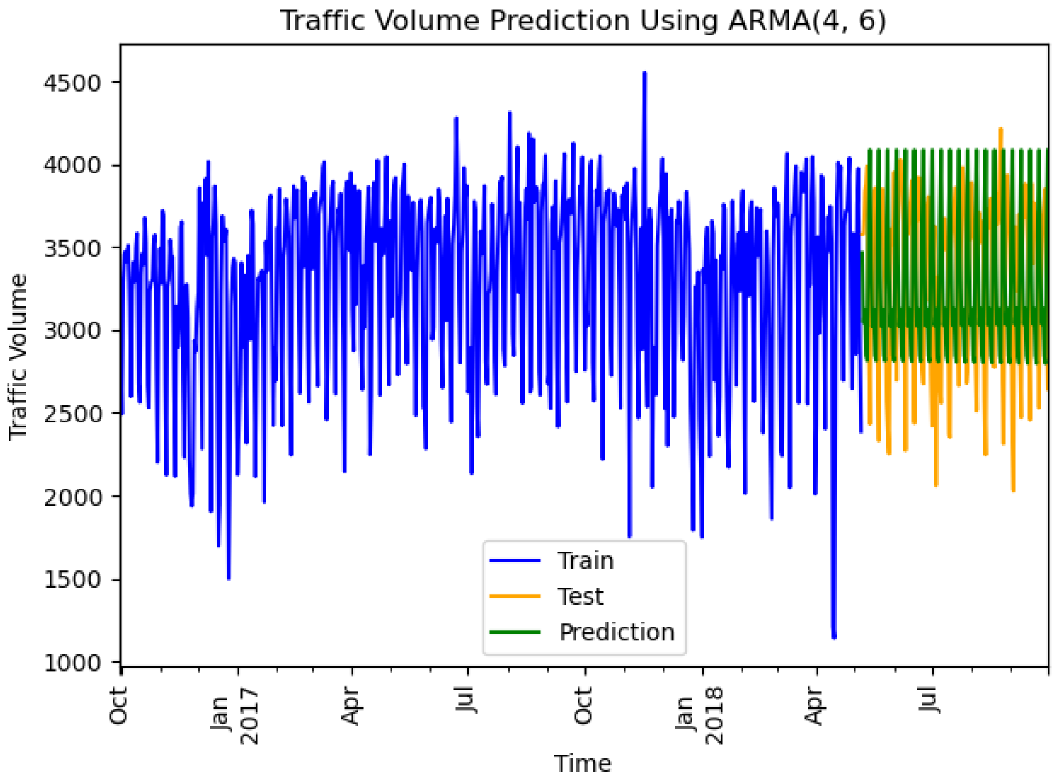
	Real	Imaginary	Modulus	Frequency
AR.1	0.6237	-0.7816j	1.0000	-0.1428
AR.2	0.6237	+0.7816j	1.0000	0.1428
AR.3	-0.2224	-0.9750j	1.0001	-0.2857
AR.4	-0.2224	+0.9750j	1.0001	0.2857
MA.1	0.6246	-0.7810j	1.0001	-0.1426
MA.2	0.6246	+0.7810j	1.0001	0.1426
MA.3	-0.2225	-0.9818j	1.0066	-0.2855
MA.4	-0.2225	+0.9818j	1.0066	0.2855
MA.5	-0.8645	-2.1361j	2.3045	-0.3112
MA.6	-0.8645	+2.1361j	2.3045	0.3112

None of the roots are same, thus no zero/pole cancellation required.

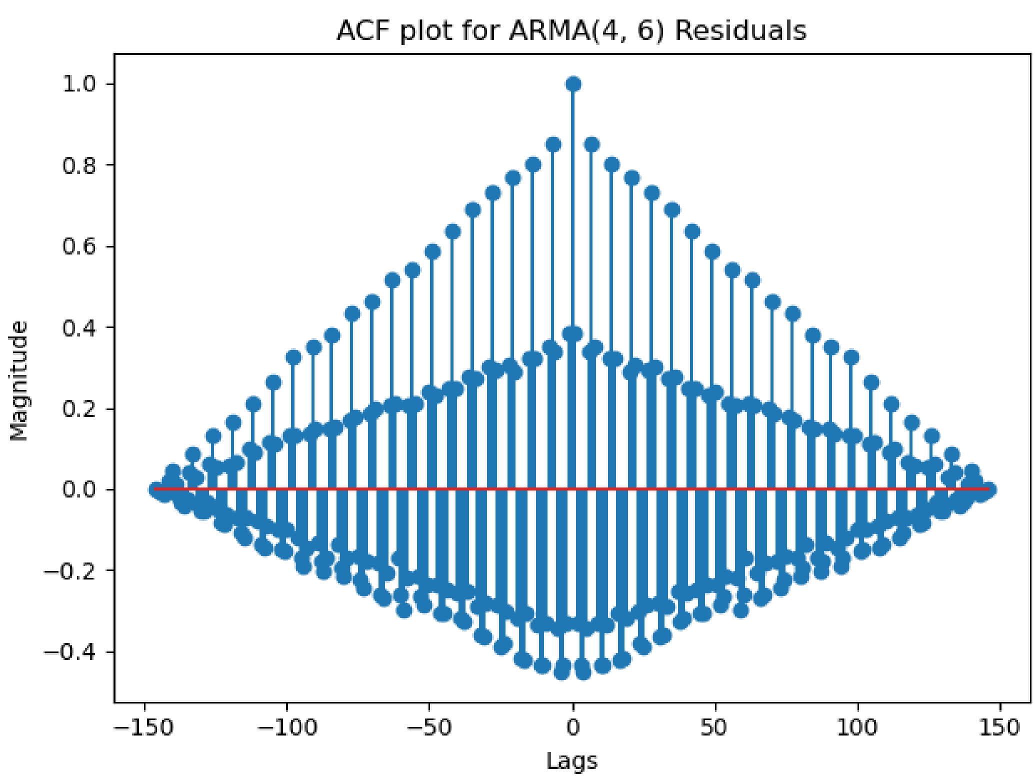
Hence the final ARMA model after simplification is ARMA(4,6).

f. Performance Measures:

Plot of Prediction: The plot of forecasted values with the actual value is shown below:



ACF of residual: The ACF of residuals for ARMA(4,6) is,



We conclude that the ACF plot does not resemble white noise.

A possible reason for poor performance of ARMA model is that our data contains seasonality.

MSE: The MSE for ARMA model is **845648.87**

RMSE: The RMSE for ARMA model is **919.591**

Mean of Residual Errors: The mean of residual for ARMA model is **55.185**

Variance of Residual Errors: The Variance of residual for ARMA model is **842603.408**

Biased or Unbiased models: Since the absolute value of mean of the residuals is greater than 0.05, we say that the model is biased. We can remove the bias by adding the mean to all the predictions.

Variance of error of estimated parameters:

Estimated variance of error for na = 4 and nb = 6 is **119649.321**

Covariance of estimated Parameters:

```
                     ar.L1.traffic_volume  ar.L2.traffic_volume  \
ar.L1.traffic_volume          1.486909e-07         -4.966135e-11
ar.L2.traffic_volume         -4.966135e-11          1.985626e-11
ar.L3.traffic_volume          1.487044e-07         -9.990091e-12
ar.L4.traffic_volume          8.187837e-11         -3.372619e-11
ma.L1.traffic_volume          2.220250e-07          5.023372e-10
ma.L2.traffic_volume         -3.055364e-07          4.958985e-10
ma.L3.traffic_volume         -1.293999e-07         -2.090309e-11
ma.L4.traffic_volume         -5.149855e-07         -4.430079e-10
ma.L5.traffic_volume         -1.754801e-07          9.044757e-10
ma.L6.traffic_volume         -6.591942e-08         -8.708043e-10
```

```
                     ar.L3.traffic_volume  ar.L4.traffic_volume  \
ar.L1.traffic_volume          1.487044e-07          8.187837e-11
ar.L2.traffic_volume         -9.990091e-12         -3.372619e-11
ar.L3.traffic_volume          1.486388e-07          8.183092e-11
ar.L4.traffic_volume          8.183092e-11          4.256228e-14
ma.L1.traffic_volume          2.272536e-07          5.810047e-11
ma.L2.traffic_volume         -3.048396e-07         -9.731181e-11
ma.L3.traffic_volume         -1.176657e-07         -9.497587e-11
ma.L4.traffic_volume         -5.277886e-07         -1.821608e-10
ma.L5.traffic_volume         -1.771354e-07         -9.716955e-11
ma.L6.traffic_volume         -7.974345e-08         -4.625718e-11
```

	ma.L1.traffic_volume	ma.L2.traffic_volume \
ar.L1.traffic_volume	2.220250e-07	-3.055364e-07
ar.L2.traffic_volume	5.023372e-10	4.958985e-10
ar.L3.traffic_volume	2.272536e-07	-3.048396e-07
ar.L4.traffic_volume	5.810047e-11	-9.731181e-11
ma.L1.traffic_volume	2.090933e-03	-1.333717e-03
ma.L2.traffic_volume	-1.333717e-03	2.345981e-03
ma.L3.traffic_volume	2.127871e-03	-2.459461e-03
ma.L4.traffic_volume	-1.591908e-03	3.231706e-03
ma.L5.traffic_volume	1.132418e-03	-1.792696e-03
ma.L6.traffic_volume	1.779648e-04	1.392947e-03

	ma.L3.traffic_volume	ma.L4.traffic_volume \
ar.L1.traffic_volume	-1.293999e-07	-5.149855e-07
ar.L2.traffic_volume	-2.090309e-11	-4.430079e-10
ar.L3.traffic_volume	-1.176657e-07	-5.277886e-07
ar.L4.traffic_volume	-9.497587e-11	-1.821608e-10
ma.L1.traffic_volume	2.127871e-03	-1.591908e-03
ma.L2.traffic_volume	-2.459461e-03	3.231706e-03
ma.L3.traffic_volume	3.598217e-03	-2.854042e-03
ma.L4.traffic_volume	-2.854042e-03	4.870101e-03
ma.L5.traffic_volume	2.632552e-03	-2.044845e-03
ma.L6.traffic_volume	-7.655813e-04	2.247116e-03

	ma.L5.traffic_volume	ma.L6.traffic_volume
ar.L1.traffic_volume	-1.754801e-07	-6.591942e-08
ar.L2.traffic_volume	9.044757e-10	-8.708043e-10
ar.L3.traffic_volume	-1.771354e-07	-7.974345e-08
ar.L4.traffic_volume	-9.716955e-11	-4.625718e-11
ma.L1.traffic_volume	1.132418e-03	1.779648e-04
ma.L2.traffic_volume	-1.792696e-03	1.392947e-03
ma.L3.traffic_volume	2.632552e-03	-7.655813e-04
ma.L4.traffic_volume	-2.044845e-03	2.247116e-03
ma.L5.traffic_volume	2.197303e-03	-8.408590e-04
ma.L6.traffic_volume	-8.408590e-04	1.607583e-03

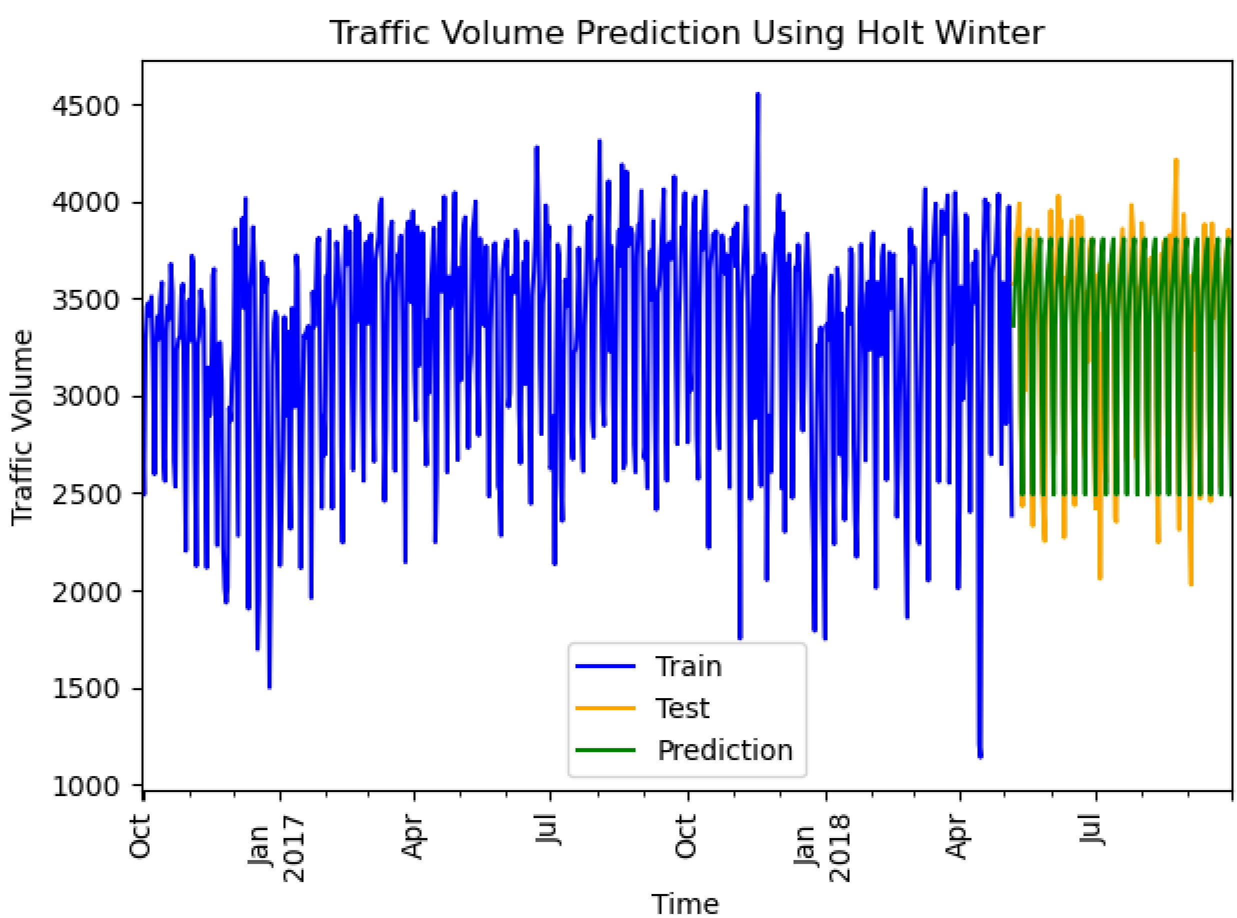
11. Final Model Selection:

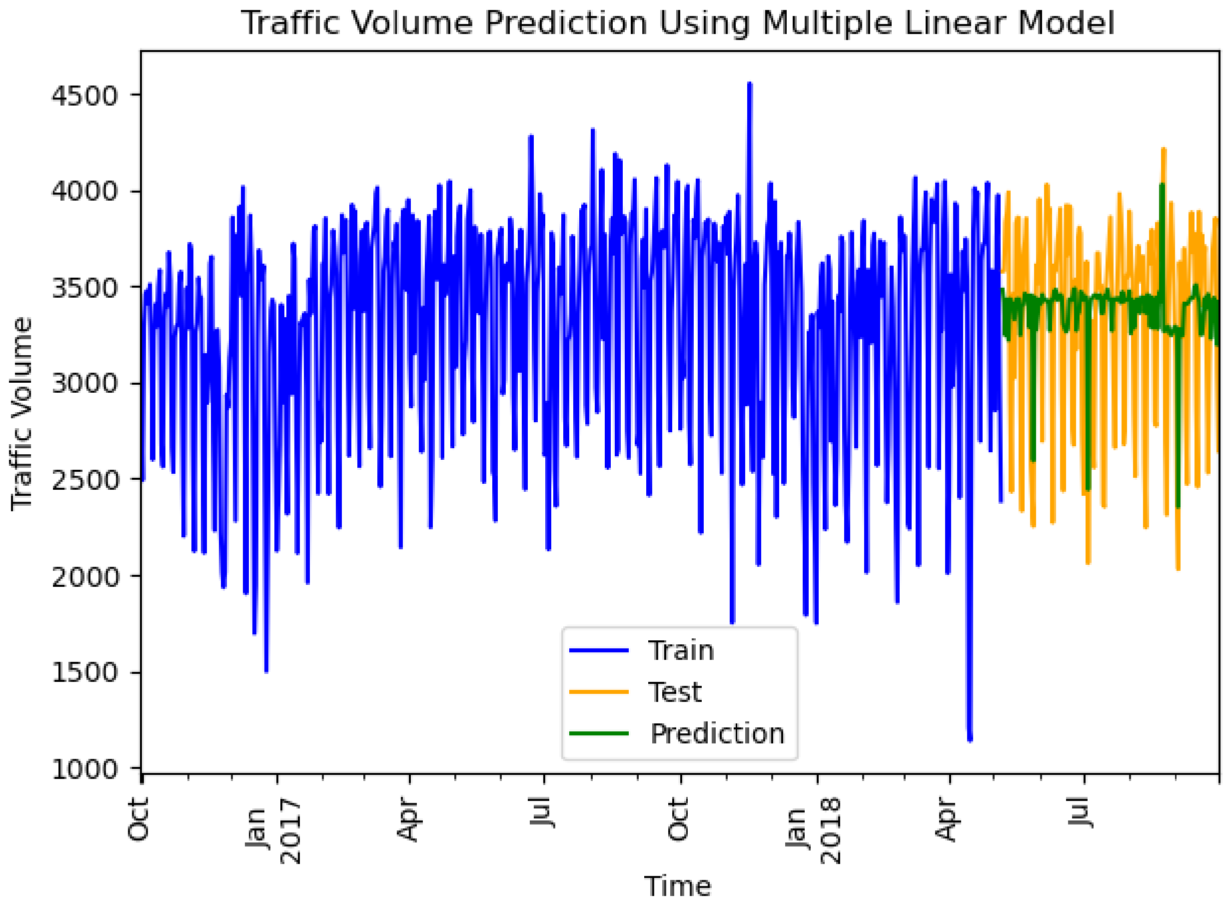
In our analysis for predicting traffic volume we have used the Average Method, Naïve Method, Drift Method, Holt Winter Model, Multiple Linear Regression model and the ARMA model. We will now compare the outputs of these models and provide conclusions.

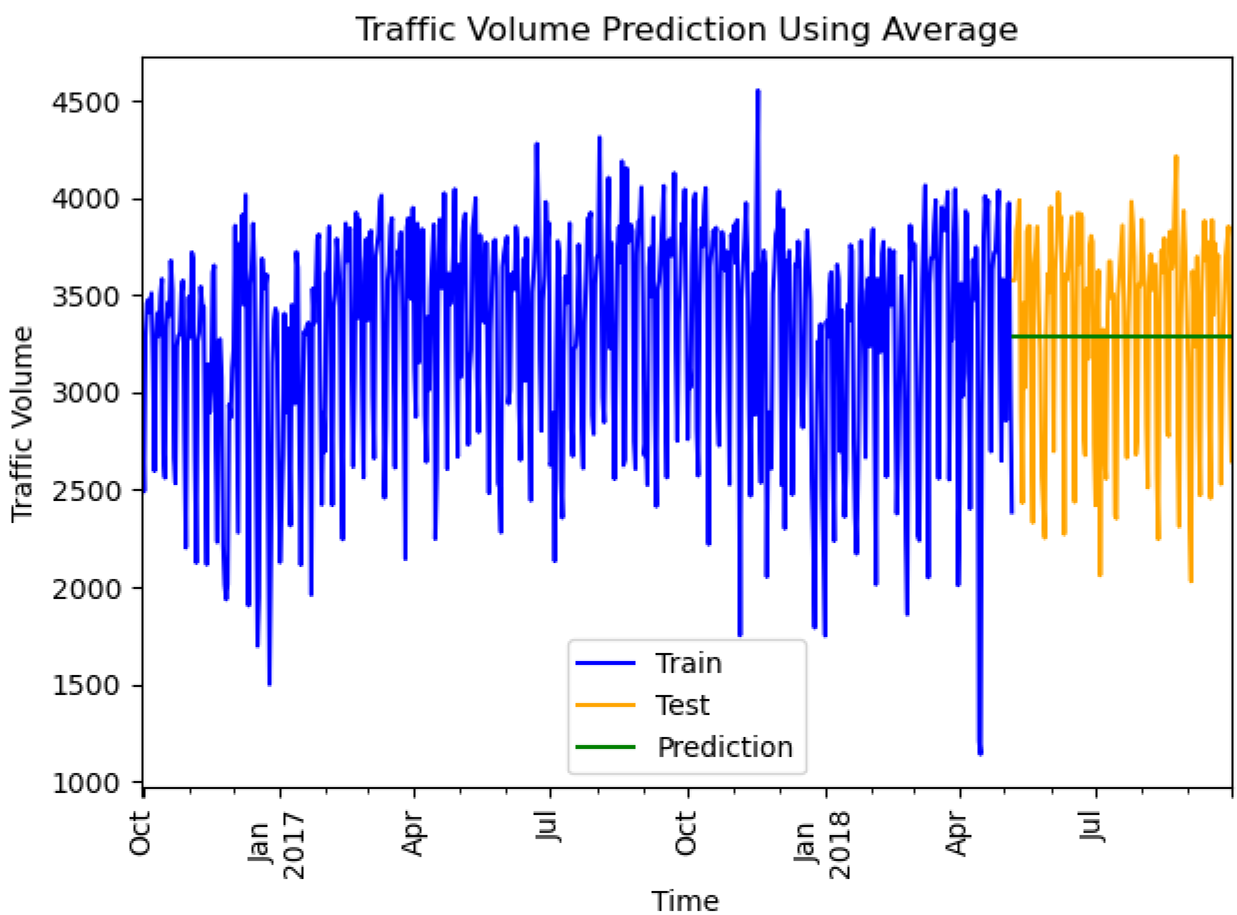
Model	MSE	RMSE	Residual Mean	Residual Variance
Holt Winter Model	84690.827	291.016884	-22.209225	84197.577336
Multiple Linear Regression Model	256424.508	506.383756	-35.470217	255166.372084
Average Model	282937.565	531.918758	55.477248	279859.840159
ARMA(4, 6) Model	845648.870	919.591687	55.185695	842603.408654
Naive Model	1191763.077	1091.679017	954.936248	279859.840159
Drift Model	1497497.754	1223.722907	1101.430227	284349.208489

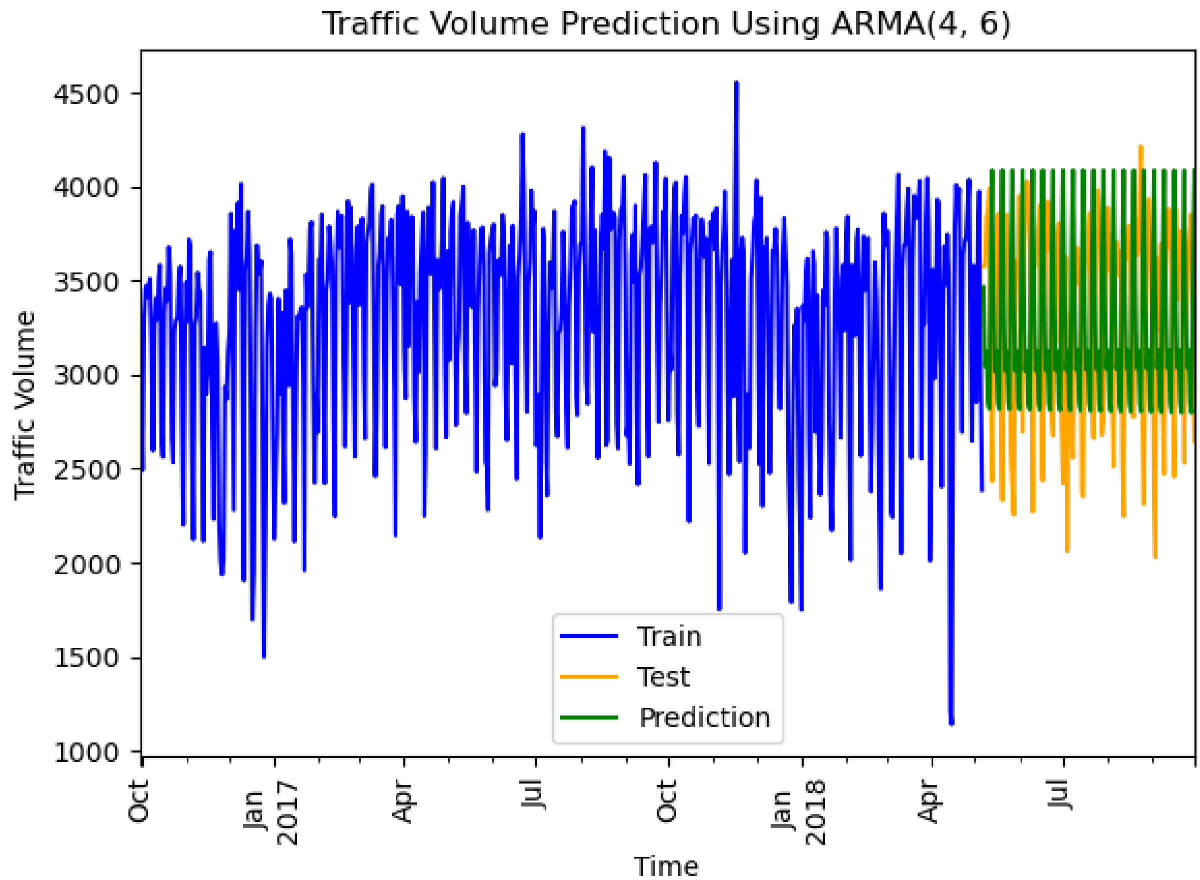
When we consider RMSE to be performance metric, we conclude that Holt Winter has the smallest RMSE and thus performs the best.
Hence for traffic volume prediction problem the Holt Winter model is the best model.

The plots for predicted and the actual values for all models are shown:

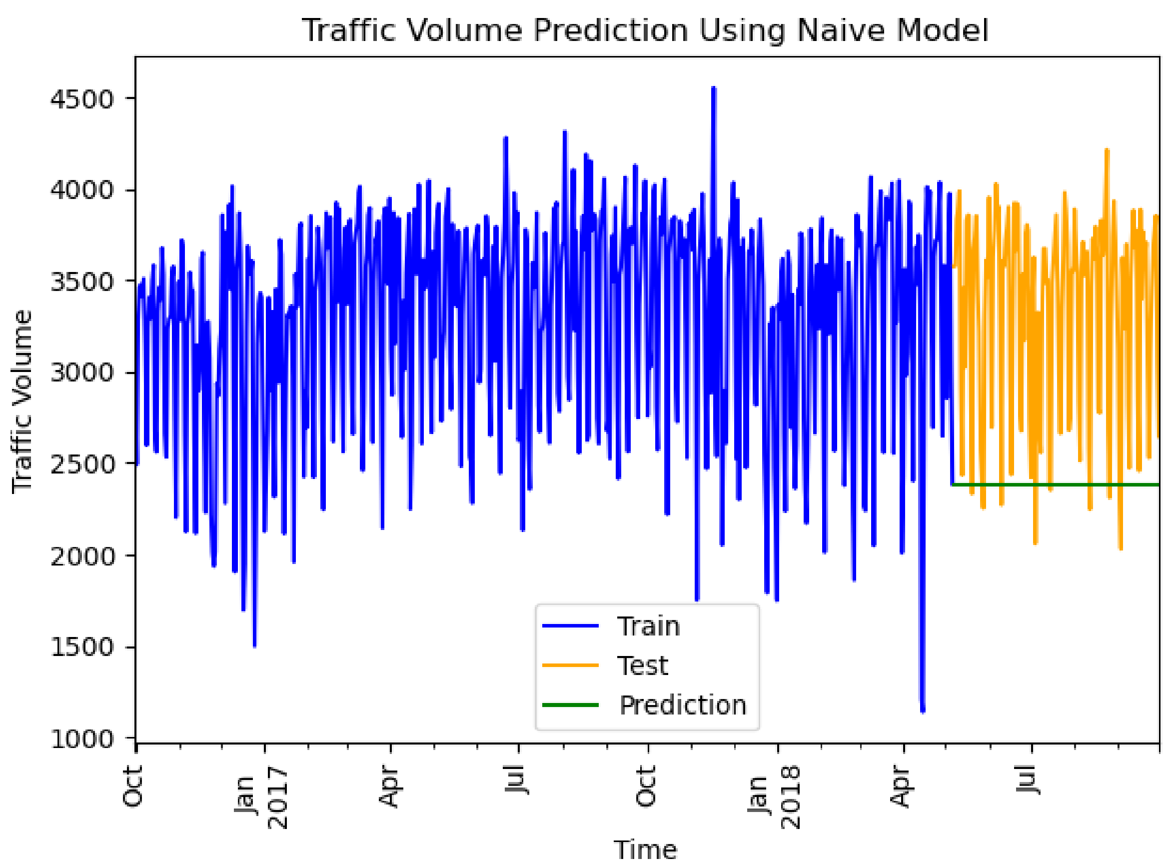



Traffic Volume Prediction Using Multiple Linear Model
Traffic Volume
Time
Train
Test
Prediction
4500
4000
3500
3000
2500
2000
1500
1000
Oct
Jan
2017
Apr
Jul
Oct
Jan
2018
Apr
Jul

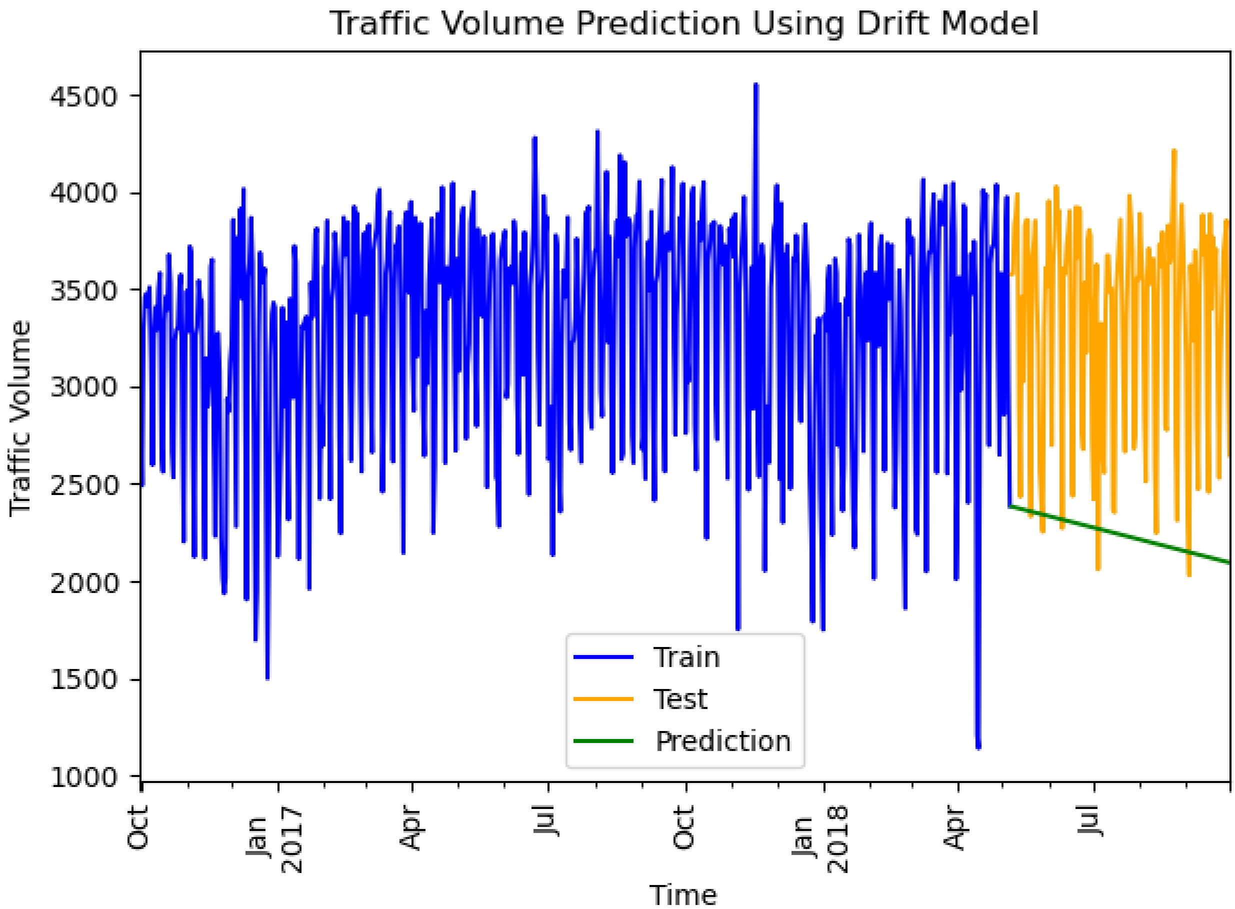

Traffic Volume Prediction Using Average
Traffic Volume
Time
Train
Test
Prediction
4500
4000
3500
3000
2500
2000
1500
1000
Oct
Jan
2017
Apr
Jul
Oct
Jan
2018
Apr
Jul



Traffic Volume Prediction Using ARMA(4, 6)
Traffic Volume
4500
4000
3500
3000
2500
2000
1500
1000
Train
Test
Prediction
Oct
Jan
2017
Apr
Jul
Oct
Jan
2018
Apr
Jul
Time



Traffic Volume Prediction Using Naive Model
Traffic Volume
4500
4000
3500
3000
2500
2000
1500
1000
Train
Test
Prediction
Oct
Jan
2017
Apr
Jul
Oct
Jan
2018
Apr
Jul
Time



12. Conclusion:

In conclusion based on the RMSE values, the Holt Winters model is recommended for traffic volume prediction.

For future scope we may want to explore other models like SARIMA or recurrent neural networks.

Appendix:

TermProject.py

```python
import warnings

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

from ToolBox import split_df_train_test, cal_auto_correlation, create_gpac_table, adf_cal,
plot_line_using_pandas_index, \
    plot_acf, plot_heatmap, plot_seasonal_decomposition, generic_average_method, \
    cal_mse, cal_forecast_errors, plot_multiline_chart_pandas_using_index, generic_naive_method,
generic_drift_method, \
    generic_holt_linear_winter, normal_equation_using_statsmodels,
normal_equation_prediction_using_statsmodels, \
    box_pierce_test, gpac_order_chi_square_test, statsmodels_estimate_parameters,
statsmodels_predict_ARMA_process, \
    statsmodels_print_covariance_matrix, statsmodels_print_variance_error

if __name__ == "__main__":
    # pandas print options
    pd.set_option("max_columns", 10)

    # ignore warnings
    warnings.filterwarnings("ignore")

    # read the original data
    traffic_raw = pd.read_csv("./data/Metro_Interstate_Traffic_Volume.csv")

    # replace all the None values with nan
    traffic_copy = traffic_raw.replace("None", np.nan)

    # convert date string column to date time object
    traffic_copy["date_time"] = pd.to_datetime(traffic_copy["date_time"], format="%Y-%m-%d")
    traffic_copy = traffic_copy.set_index(traffic_copy["date_time"])

    # focus on data from 09/2016 to 09/2018
    traffic_clipped = traffic_copy.loc['2016-09-30':'2018-09-30'].copy(deep=True)

    # resample based on daily data
    traffic_resampled = traffic_clipped.groupby(pd.Grouper(freq="D")).aggregate(
        {"temp": "mean", "clouds_all": "mean", "weather_main": "first", "traffic_volume": "mean", "holiday":
"first",
         "rain_1h": "mean", "snow_1h": "mean"})

    print()
    print("The dimension of the resampled data is as follows:")
    print(traffic_resampled.shape)

    print()
```

```
    print("The summary statistics of numeric data after resampling to daily data is:")
    print(traffic_resampled.describe(include=["float64"]))

    # drop the snow_1h column
    traffic_resampled.drop(["snow_1h"], axis=1, inplace=True)

    print()
    print("The summary statistics of categorical data after resampling to daily data is:")
    print(traffic_resampled.describe(include=["object"]))

    traffic_resampled["holiday"] = traffic_resampled["holiday"].replace({np.nan: "No Holiday"})
    print()
    print("After replacing all the holiday NaN columns with 'No Holiday' value we get value counts for holiday
column "
          "as:")
    print(traffic_resampled["holiday"].value_counts())

    # weather_main column value counts before condensing
    print()
    print("The value counts for weather_main column before condensing it:")
    print(traffic_resampled["weather_main"].value_counts())

    # condense the weather main categorical values
    traffic_resampled = traffic_resampled.replace(
        {"weather_main": {"Drizzle": "Rain", "Thunderstorm": "Rain", "Mist": "Fog", "Haze": "Fog", "Smoke":
"Fog"}})

    # weather_main column value counts after condensing
    print()
    print("The value counts for weather_main column after condensing it:")
    print(traffic_resampled["weather_main"].value_counts())

    # after resampling we find these many NaN rows
    print()
    print("After resampling and data cleaning, column count with NaN values are:")
    print(traffic_resampled.isnull().sum())

    # Plot of the dependent variable versus time.
    plot_line_using_pandas_index(traffic_resampled, "traffic_volume", "Traffic Volume over time", "Navy",
"Time",
                                 y_axis_label="Traffic Volume")

    # ACF of the dependent variable.
    autocorrelation = cal_auto_correlation(list(traffic_resampled["traffic_volume"]), 200)
    plot_acf(autocorrelation, "ACF plot for Traffic Volume")

    # Correlation Matrix with seaborn heatmap and pearson's correlation coefficient
    corr = traffic_resampled.corr()
    plot_heatmap(corr, "Heatmap for Correlation Coefficient for Traffic Volume Data")

    # split into train and test(20%) dataset
    train, test = split_df_train_test(traffic_resampled, 0.2)
```

```python
# dimension of train data
print()
print("The dimension of train data is:")
print(train.shape)

# dimension of test data
print()
print("The dimension of test data is:")
print(test.shape)

# combining train and test data
combined_data = train.append(test)

# Stationarity
print()
adf_cal(combined_data, "traffic_volume")

# Time series Decomposition
# the train dataframe already has DateTimeIndex as index which specified the frequency as 'D'
plot_seasonal_decomposition(train["traffic_volume"], None, "Multiplicative Residuals", "multiplicative")
plot_seasonal_decomposition(train["traffic_volume"], None, "Additive Residuals", "additive")

# to keep track of performance for all the models
result_performance = pd.DataFrame(
    {"Model": [], "MSE": [], "RMSE": [], "Residual Mean": [], "Residual Variance": []})

print("-------------------------------------------- Average Model --------------------------------------------")

# average model
average_predictions = generic_average_method(train["traffic_volume"], len(test["traffic_volume"]))

avg_mse = cal_mse(test["traffic_volume"], average_predictions)
print()
print("The MSE for Average model is:")
print(avg_mse)

avg_rmse = np.sqrt(avg_mse)
print()
print("The RMSE for Average model is:")
print(avg_rmse)

# forecast errors for average model
residuals_avg = cal_forecast_errors(test["traffic_volume"], average_predictions)

# average residual variance
avg_variance = np.var(residuals_avg)
print()
print("The Variance of residual for Average model is:")
print(avg_variance)

# Average residual mean
```

```
avg_mean = np.mean(residuals_avg)
print()
print("The Mean of residual for Average model is:")
print(avg_mean)

# Average residual ACF
residual_autocorrelation_average = cal_auto_correlation(residuals_avg, len(average_predictions))
plot_acf(residual_autocorrelation_average, "ACF plot using Average Residuals")

# add the results to common dataframe
result_performance = result_performance.append(
    pd.DataFrame(
        {"Model": ["Average Model"], "MSE": [avg_mse], "RMSE": [avg_rmse],
         "Residual Mean": [avg_mean], "Residual Variance": [avg_variance]}))

# plot the predicted vs actual data
average_df = test.copy(deep=True)
average_df["traffic_volume"] = average_predictions

plot_multiline_chart_pandas_using_index([train, test, average_df], "traffic_volume",
                                        ["Train", "Test", "Prediction"], ["Blue", "Orange", "Green"],
                                        "Time", "Traffic Volume",
                                        "Traffic Volume Prediction Using Average",
                                        rotate_xticks=True)

print("------------------------------ Naive Model ------------------------------")

# naive model
naive_predictions = generic_naive_method(train["traffic_volume"], len(test["traffic_volume"]))

naive_mse = cal_mse(test["traffic_volume"], naive_predictions)
print()
print("The MSE for Naive model is:")
print(naive_mse)

naive_rmse = np.sqrt(naive_mse)
print()
print("The RMSE for Naive model is:")
print(naive_rmse)

# forecast errors for naive model
residuals_naive = cal_forecast_errors(test["traffic_volume"], naive_predictions)

# naive residual variance
naive_variance = np.var(residuals_naive)
print()
print("The Variance of residual for Naive model is:")
print(naive_variance)

# naive residual mean
naive_mean = np.mean(residuals_naive)
print()
```

```
print("The Mean of residual for Naive model is:")
print(naive_mean)

# naive residual ACF
residual_autocorrelation_naive = cal_auto_correlation(residuals_naive, len(naive_predictions))
plot_acf(residual_autocorrelation_naive, "ACF plot using Naive Residuals")

# add the results to common dataframe
result_performance = result_performance.append(
    pd.DataFrame(
        {"Model": ["Naive Model"], "MSE": [naive_mse], "RMSE": [naive_rmse],
         "Residual Mean": [naive_mean], "Residual Variance": [naive_variance]}))

# plot the predicted vs actual data
naive_df = test.copy(deep=True)
naive_df["traffic_volume"] = naive_predictions

plot_multiline_chart_pandas_using_index([train, test, naive_df], "traffic_volume",
                                        ["Train", "Test", "Prediction"], ["Blue", "Orange", "Green"],
                                        "Time", "Traffic Volume",
                                        "Traffic Volume Prediction Using Naive Model",
                                        rotate_xticks=True)

print("--------------------------------------- Drift Model ---------------------------------------")

# drift model
drift_predictions = generic_drift_method(train["traffic_volume"], len(test["traffic_volume"]))

drift_mse = cal_mse(test["traffic_volume"], drift_predictions)
print()
print("The MSE for drift model is:")
print(drift_mse)

drift_rmse = np.sqrt(drift_mse)
print()
print("The RMSE for Drift model is:")
print(drift_rmse)

# forecast errors for drift model
residuals_drift = cal_forecast_errors(test["traffic_volume"], drift_predictions)

# drift residual variance
drift_variance = np.var(residuals_drift)
print()
print("The Variance of residual for Drift model is:")
print(drift_variance)

# drift residual mean
drift_mean = np.mean(residuals_drift)
print()
print("The Mean of residual for drift model is:")
print(drift_mean)
```

```
    # drift residual ACF
    residual_autocorrelation_drift = cal_auto_correlation(residuals_drift, len(drift_predictions))
    plot_acf(residual_autocorrelation_drift, "ACF plot using drift Residuals")

    # add the results to common dataframe
    result_performance = result_performance.append(
        pd.DataFrame(
            {"Model": ["Drift Model"], "MSE": [drift_mse], "RMSE": [drift_rmse],
             "Residual Mean": [drift_mean], "Residual Variance": [drift_variance]}))

    # plot the predicted vs actual data
    drift_df = test.copy(deep=True)
    drift_df["traffic_volume"] = drift_predictions

    plot_multiline_chart_pandas_using_index([train, test, drift_df], "traffic_volume",
                                            ["Train", "Test", "Prediction"], ["Blue", "Orange", "Green"],
                                            "Time", "Traffic Volume",
                                            "Traffic Volume Prediction Using Drift Model",
                                            rotate_xticks=True)

    print("--------------------------- HOLT WINTER ---------------------------")
    # holt winter prediction
    holt_winter_prediction = generic_holt_linear_winter(train["traffic_volume"], test["traffic_volume"], None,
None,
                                                        "mul", None)
    # holt winter mse
    holt_winter_mse = cal_mse(test["traffic_volume"], holt_winter_prediction)

    print()
    print("The MSE for Holt Winter model is:")
    print(holt_winter_mse)

    # holt winter rmse
    holt_winter_rmse = np.sqrt(holt_winter_mse)
    print()
    print("The RMSE for Holt Winter model is:")
    print(holt_winter_rmse)

    # holt winter residual
    residuals_holt_winter = cal_forecast_errors(list(test["traffic_volume"]), holt_winter_prediction)
    residual_autocorrelation_holt_winter = cal_auto_correlation(residuals_holt_winter,
len(holt_winter_prediction))

    # holt winter residual variance
    holt_winter_variance = np.var(residuals_holt_winter)
    print()
    print("The Variance of residual for Holt Winter model is:")
    print(holt_winter_variance)

    # holt winter residual mean
    holt_winter_mean = np.mean(residuals_holt_winter)
```

```
print()
print("The Mean of residual for Holt Winter model is:")
print(holt_winter_mean)

# holt winter residual ACF
plot_acf(residual_autocorrelation_holt_winter, "ACF plot using Holt Winter Residuals")

# add the results to common dataframe
result_performance = result_performance.append(
    pd.DataFrame(
        {"Model": ["Holt Winter Model"], "MSE": [holt_winter_mse], "RMSE": [holt_winter_rmse],
         "Residual Mean": [holt_winter_mean], "Residual Variance": [holt_winter_variance]}))

# plot the predicted vs actual data
holt_winter_df = test.copy(deep=True)
holt_winter_df["traffic_volume"] = holt_winter_prediction

plot_multiline_chart_pandas_using_index([train, test, holt_winter_df], "traffic_volume",
                                        ["Train", "Test", "Prediction"], ["Blue", "Orange", "Green"],
                                        "Time", "Traffic Volume",
                                        "Traffic Volume Prediction Using Holt Winter",
                                        rotate_xticks=True)

print("------------------------------ MULTIPLE LINEAR REGRESSION------------------------")
lm_combined = combined_data.copy(deep=True)

# convert categorical into numerical columns
lm_combined = pd.get_dummies(lm_combined)

# separate train and test data
lm_train = lm_combined[:len(train)]
lm_test = lm_combined[len(train):]

# Scaling the data using MixMax Scaler
mm_scaler = MinMaxScaler()
lm_train_mm_scaled = pd.DataFrame(
    mm_scaler.fit_transform(lm_train[np.setdiff1d(lm_train.columns, ["traffic_volume"])]),
    columns=np.setdiff1d(lm_train.columns, ["traffic_volume"]))
lm_train_mm_scaled.set_index(lm_train.index, inplace=True)
lm_train_mm_scaled["traffic_volume"] = lm_train["traffic_volume"]

lm_test_mm_scaled = pd.DataFrame(mm_scaler.transform(lm_test[np.setdiff1d(lm_test.columns,
["traffic_volume"])]),
                                 columns=np.setdiff1d(lm_test.columns, ["traffic_volume"]))
lm_test_mm_scaled.set_index(lm_test.index, inplace=True)
lm_test_mm_scaled["traffic_volume"] = lm_test["traffic_volume"]

# linear model using all variables
basic_model = normal_equation_using_statsmodels(
    lm_train_mm_scaled[np.setdiff1d(lm_train_mm_scaled.columns, "traffic_volume")],
    lm_train_mm_scaled["traffic_volume"], intercept=False)
```

```python
    print()
    print("The summary of linear model with all variables is:")
    print(basic_model.summary())

    features = np.setdiff1d(lm_train_mm_scaled.columns,
                            ["rain_1h", "holiday_Christmas Day", "holiday_Memorial Day", "holiday_Thanksgiving Day",
                             "holiday_New Years Day", "holiday_Independence Day", "holiday_Labor Day", "clouds_all",
                             "holiday_Washingtons Birthday", "holiday_Martin Luther King Jr Day"])

    # linear model using features which pass the t-test
    pruned_model = normal_equation_using_statsmodels(lm_train_mm_scaled[np.setdiff1d(features,
"traffic_volume")],
                                                     lm_train_mm_scaled["traffic_volume"], intercept=False)

    print()
    print("The summary of linear model after feature selection:")
    print(pruned_model.summary())

    # linear model predictions
    lm_predictions = normal_equation_prediction_using_statsmodels(pruned_model, lm_test_mm_scaled[
        np.setdiff1d(features, "traffic_volume")], intercept=False)

    # linear model mse
    lm_mse = cal_mse(test["traffic_volume"], lm_predictions)

    print()
    print("The MSE for Linear Model model is:")
    print(lm_mse)

    # linear model rmse
    lm_rmse = np.sqrt(lm_mse)
    print()
    print("The RMSE for Linear Model model is:")
    print(lm_rmse)

    # linear model residual
    residuals_lm = cal_forecast_errors(list(test["traffic_volume"]), lm_predictions)
    residual_autocorrelation = cal_auto_correlation(residuals_lm, len(lm_predictions))

    # linear model residual variance
    lm_variance = np.var(residuals_lm)
    print()
    print("The Variance of residual for Linear Model model is:")
    print(lm_variance)

    # linear model residual mean
    lm_mean = np.mean(residuals_lm)
    print()
    print("The Mean of residual for Linear Model model is:")
    print(lm_mean)

    # linear model residual ACF
```

```
    plot_acf(residual_autocorrelation, "ACF plot for Linear Model Residuals")

    # linear model Q value
    Q_value_lm = box_pierce_test(len(test), residuals_lm, len(test))
    print()
    print("The Q Value of residuals for Linear Model model is:")
    print(Q_value_lm)

    # add the results to common dataframe
    result_performance = result_performance.append(
        pd.DataFrame(
            {"Model": ["Multiple Linear Regression Model"], "MSE": [lm_mse], "RMSE": [lm_rmse],
             "Residual Mean": [lm_mean], "Residual Variance": [lm_variance]}))

    # plot the actual vs predicted values
    lm_predictions_scaled = lm_test_mm_scaled.copy(deep=True)
    lm_predictions_scaled["traffic_volume"] = lm_predictions

    plot_multiline_chart_pandas_using_index([lm_train_mm_scaled, lm_test_mm_scaled,
lm_predictions_scaled],
                                            "traffic_volume",
                                            ["Train", "Test", "Prediction"], ["Blue", "Orange", "Green"],
                                            "Time", "Traffic Volume",
                                            "Traffic Volume Prediction Using Multiple Linear Model",
                                            rotate_xticks=True)

    # correlation coefficient for linear model after feature selection
    corr = lm_train[features].corr()
    label_ticks = ["Columbus Day", "No Holiday", "State Fair", "Veterans Day", "temp", "traffic_volume",
"Clear",
                   "Clouds", "Fog", "Rain", "Snow"]
    plot_heatmap(corr, "Correlation Coefficient for Linear Model after feature selection", label_ticks,
label_ticks, 45)

    print("------------------------------ ARMA ------------------------------")
    j = 12
    k = 12
    lags = j + k

    y_mean = np.mean(train["traffic_volume"])
    y = np.subtract(y_mean, train["traffic_volume"])

    actual_output = np.subtract(y_mean, test["traffic_volume"])

    # autocorrelation of traffic volume
    ry = cal_auto_correlation(y, lags)

    # create GPAC Table
    gpac_table = create_gpac_table(j, k, ry)
    print()
    print("GPAC Table:")
    print(gpac_table.to_string())
```

```
    print()

    plot_heatmap(gpac_table, "GPAC Table for Traffic Volume")

    # # estimate the order of the process
    # # the possible orders identified from GPAC table don't pass the chi square test
    possible_order2 = [(2, 5), (2, 7), (4, 0), (4, 2), (4, 5), (4, 7), (6, 5), (10, 3)]

    print()
    print("The possible orders identified from GPAC for ARMA process are:")
    print(possible_order2)
    print()
    print("We noticed that none of the identified ARMA order from the GPAC table pass the chi squared
test.")
    print()

    # # checking which orders pass the GPAC test
    # print(gpac_order_chi_square_test(possible_order2, y, '2018-05-07 00:00:00', '2018-09-30
00:00:00',
    #                                  lags,
    #                                  test["traffic_volume"], y_mean))

    print(
        "Thus we try for all possible combinations of orders from the GPAC table in a brute force manner; \n"
        "the ARMA(4,6) passes the Chi Square test, but shows no pattern in GPAC table;\n"
        "this might be possible since we have only 584 samples in the training data.")

    print()
    print("The ARMA(4,6) model summary is:")

    possible_order = [(4, 6)]
    gpac_order_chi_square_test(possible_order, y, '2018-05-07 00:00:00', '2018-09-30 00:00:00',
                               lags, actual_output)

    n_a = 4
    n_b = 6

    model = statsmodels_estimate_parameters(n_a, n_b, y)
    print(model.summary())

    # ARMA predictions
    arma_prediction = statsmodels_predict_ARMA_process(model, "2018-05-07 00:00:00", "2018-09-30
00:00:00")

    # add the subtracted mean back into the predictions
    arma_prediction = np.add(y_mean, arma_prediction)

    # ARMA mse
    arma_mse = cal_mse(test["traffic_volume"], arma_prediction)
    print()
    print(f"The MSE for ARMA({n_a}, {n_b}) model is:")
    print(arma_mse)
```

```python
    # ARMA rmse
    arma_rmse = np.sqrt(arma_mse)
    print()
    print(f"The RMSE for ARMA({n_a}, {n_b}) model is:")
    print(arma_rmse)

    # ARMA residual
    residuals_arma = cal_forecast_errors(list(test["traffic_volume"]), arma_prediction)

    # ARMA residual variance
    arma_variance = np.var(residuals_arma)
    print()
    print("The Variance of residual for ARMA model is:")
    print(arma_variance)

    # ARMA residual mean
    arma_mean = np.mean(residuals_arma)
    print()
    print(f"The Mean of residual for ARMA({n_a}, {n_b}) model is:")
    print(arma_mean)

    # ARMA residual ACF
    residual_autocorrelation_arma = cal_auto_correlation(residuals_arma, len(arma_prediction))
    plot_acf(residual_autocorrelation_arma, f"ACF plot for ARMA({n_a}, {n_b}) Residuals")

    # ARMA covariance matrix
    print()
    statsmodels_print_covariance_matrix(model, n_a, n_b)

    # ARMA estimated variance of error
    statsmodels_print_variance_error(model, n_a, n_b)

    # add the results to common dataframe
    result_performance = result_performance.append(
        pd.DataFrame(
            {"Model": [f"ARMA({n_a}, {n_b}) Model"], "MSE": [arma_mse], "RMSE": [arma_rmse],
             "Residual Mean": [arma_mean], "Residual Variance": [arma_variance]}))

    # plot the predicted vs actual data
    arma_df = test.copy(deep=True)
    arma_df["traffic_volume"] = arma_prediction

    plot_multiline_chart_pandas_using_index([train, test, arma_df], "traffic_volume",
                                            ["Train", "Test", "Prediction"], ["Blue", "Orange", "Green"],
                                            "Time", "Traffic Volume",
                                            f"Traffic Volume Prediction Using ARMA({n_a}, {n_b})",
                                            rotate_xticks=True)

# ---------------------------------------Final Performance Metrics--------------------
print()
```

```python
print("The performance metrics for all the models is shown:")
print(result_performance.sort_values(["RMSE"]).reset_index(drop=True).to_string())
```

ToolBox.py

```python
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns
import statsmodels.api as sm
import statsmodels.tsa.holtwinters as ets
from lifelines import KaplanMeierFitter
from scipy.signal import dlsim
from scipy.stats import chi2
from scipy.stats import t
from sklearn.model_selection import train_test_split
from statsmodels.tsa.seasonal import seasonal_decompose
from statsmodels.tsa.stattools import adfuller


def plot_line_using_pandas_index(df, y_axis_data, titleOfPlot, color, x_axis_label, legend=False,
                                 y_axis_label=None, legendList=None, x_tick_data=False):
    """
    Plots line chart based on index as x axis and y axis label passed
    """
    line_chart = df[y_axis_data].plot(kind="line", rot=90, legend=legend, title=titleOfPlot, color=color)
    line_chart.set_xlabel(x_axis_label)

    line_chart.set_ylabel(y_axis_data if y_axis_label is None else y_axis_label)
    if legend:
        line_chart.legend(legendList)
    if x_tick_data:
        plt.xticks(df.index.values)
    plt.show()


def plot_line(df, x_axis_data, y_axis_data, titleOfPlot, color, legend=False, x_axis_label=None,
y_axis_label=None,
              legendList=None, x_tick_data=False):
    """
    Plots line chart based on x axis label and y axis label passed
    """
    line_chart = df.plot.line(x=x_axis_data, y=y_axis_data, rot=90, legend=legend, title=titleOfPlot,
color=color)
    line_chart.set_xlabel(x_axis_data if x_axis_label is None else x_axis_label)
    line_chart.set_ylabel(y_axis_data if y_axis_label is None else y_axis_label)
    if legend:
        line_chart.legend(legendList)
    if x_tick_data:
        plt.xticks(df[x_axis_data])
    plt.show()
```

```
def plot_line_subplot(df, x_axis_data, y_axis_data, titleOfPlot, color, axes, legend=False,
x_axis_label=None,
                      y_axis_label=None,
                      legend_list=None):
    """
    Creates line chart based on x axis label and y axis label passed and the axes object, but does not plot it
    """

    line_chart = df.plot.line(x=x_axis_data, y=y_axis_data, rot=45, legend=legend, title=titleOfPlot,
color=color,
                              ax=axes)
    line_chart.set_xlabel(x_axis_data if x_axis_label is None else x_axis_label)
    line_chart.set_ylabel(y_axis_data if y_axis_label is None else y_axis_label)
    if legend:
        line_chart.legend(legend_list)


def get_descriptive_stats(df, attribute):
    """"
    Computes mean, standard deviation and variance for a Dataframe attribute
    """
    mean_data = df[attribute].mean().round(3)
    variance_data = df[attribute].var().round(3)
    std_data = df[attribute].std().round(3)

    return mean_data, variance_data, std_data


def compute_stepwise_stats(df, time_attribute, data_attribute):
    """
    Computes stepwise mean and variance based on data frame and attribute specified
    """

    # initialize empty data frame
    stepwise_df = pd.DataFrame(columns=["Time", "Stepwise Mean", "Stepwise Variance"])

    for index in range(0, len(df)):
        # compute mean and variance and append it to empty data frame
        stepwise_df = stepwise_df.append({"Time": df.iloc[index][time_attribute],
                                          # using index + 1 since head(0) is NaN and hence start from next index
                                          "Stepwise Mean": df.head(index + 1)[data_attribute].mean(),
                                          "Stepwise Variance": df.head(index + 1)[data_attribute].var()},
                                         ignore_index=True)
    return stepwise_df


def adf_cal(df, attribute):
    """
    Computes and prints ADF Statistics using statsmodels.tsa.stattools.adfuller()
    """
    print("ADF Test for", attribute)
```

```
    result = adfuller(df[attribute])
    print("ADF Statistic: %f" % result[0])
    print("p-value: %f" % result[1])
    print("Critical Values: ")
    for key, value in result[4].items():
        print("\t%s: %.3f" % (key, value))
    print()

    if result[1] < 0.05:
        print("Since p-value is less than 0.05, reject null hypothesis thus time series data is Stationary")
    else:
        print(
            "Since p-value is not less than 0.05, we failed to reject null hypothesis thus time series data is "
            "Non-Stationary")
    print()
    return result


def plot_seasonal_decomposition(input_list, frequency_of_data: int, title,
type_of_decomposition="additive"):
    decomposed_data = seasonal_decompose(input_list, type_of_decomposition,
period=frequency_of_data)
    decomposed_data.plot()
    plt.title(title)
    plt.show()


def transform_using_differencing(df, time_attribute, data_attribute):
    """
    Transforming using first order differencing corrects trend in non stationary data. Differencing has a
caveat that
    we lose the first data point.
    """

    # initialize empty data frame
    difference_df = pd.DataFrame(columns=[time_attribute, data_attribute])

    for index in range(0, len(df) - 1):
        difference_df = difference_df.append({
            time_attribute: df.iloc[index][time_attribute],
            # Difference Value = Next - Current
            data_attribute: (df.iloc[index + 1][data_attribute] - df.iloc[index][data_attribute]),
        }, ignore_index=True)

    return difference_df


def reverse_transform_for_differencing(original_input_list, differenced_df_list_with_predicted_values):
    """ returns transformed values for predicted values only"""
    last_index = len(original_input_list) - 1
    prediction_range = len(differenced_df_list_with_predicted_values) - len(original_input_list) + 1
```

```python
    back_transformed = []
    predicted_sum = 0
    for i in range(prediction_range):
        predicted_sum += differenced_df_list_with_predicted_values[last_index + i]
        predicted_value = original_input_list[last_index] + predicted_sum
        back_transformed.append(predicted_value)

    return back_transformed


def transform_using_logarithms(df, data_attribute):
    """
    Transforming using logarithm corrects variance in non stationary data. \nNote: We are using log to the
base 10.
    """
    log_df = df.copy(deep=True)
    log_df[data_attribute] = np.log10(log_df[data_attribute])

    return log_df


def reverse_transform_using_logarithms(original_input_list, log_transformed_list):
    reversed_log = np.power(10, log_transformed_list[len(original_input_list):])
    return list(reversed_log)


def correlation_coefficient_cal(x, y):
    """
    Python function that returns correlation coefficient based on formula of,
    r = (cross correlation of x and y) / ((std dev of x) * (std dev of y))
    Takes 2 dataset [series data] as input and returns the correlation coefficient
    """

    # find the mean of x
    x_mean = np.mean(x)

    # find the mean of y
    y_mean = np.mean(y)

    # multiply the difference between x mean and x with y mean and y
    numerator = np.sum(np.multiply(np.subtract(x, x_mean), np.subtract(y, y_mean)))

    # find standard deviation of x
    x_std_dev = np.sqrt(np.sum(np.square(np.subtract(x, x_mean))))

    # find standard deviation of y
    y_std_dev = np.sqrt(np.sum(np.square(np.subtract(y, y_mean))))

    # multiply x_std_dev and y_std_dev
    denominator = x_std_dev * y_std_dev

    # perform division
```

```python
    if denominator != 0:
        # round the division to 3 decimal places
        return round(numerator / denominator, 3)
    else:
        return 0


def create_scatter_plot(x, y, x_label, y_label, title_of_plot, color):
    """Title should contain correlation coefficient"""
    plt.scatter(x, y, c=color)
    plt.xlabel(x_label)
    plt.ylabel(y_label)
    plt.title(title_of_plot)
    plt.show()


def plot_hist(input_data, title, color, x_axis_label=None, y_axis_label=None):
    """ Plots histogram based on list of input_data"""
    plt.hist(input_data, color=color)
    plt.title(title)
    plt.xlabel(x_axis_label)
    plt.ylabel(y_axis_label)
    plt.show()


def cal_auto_correlation(input_array, number_of_lags, precision=3):
    """
    :param precision: tells the precision of rounding
    :param input_array: a vector or array which contains the values
    :param number_of_lags: how many time shifts are
    desired when number_of_lags = 0, it means no time shift
    :return: a list containing the values of auto correlation for the number of lags specified
    """

    # find the mean
    mean_of_input = np.mean(input_array)

    # create empty result array
    result = []

    # compute denominator for autocorrelation equation
    denominator = np.sum(np.square(np.subtract(input_array, mean_of_input)))

    # iterate for the number of lags mentioned
    for k in range(0, number_of_lags):

        # initialize numerator
        numerator = 0

        # iterate from k to the size of input array
        for i in range(k, len(input_array)):
            numerator += (input_array[i] - mean_of_input) * (input_array[i - k] - mean_of_input)
```

```
        if denominator != 0:
            # perform division and append output to list
            result.append(np.round(numerator / denominator, precision))

    return result


def compute_autocorrelation_single_lag(input_array, lag, precision=3):
    # find the mean
    mean_of_input = np.mean(input_array)

    # compute denominator for autocorrelation equation
    denominator = np.sum(np.square(np.subtract(input_array, mean_of_input)))

    # initialize numerator
    numerator = 0

    # iterate from k to the size of input array
    for i in range(lag, len(input_array)):
        numerator += (input_array[i] - mean_of_input) * (input_array[i - lag] - mean_of_input)

    if denominator != 0:
        # perform division and append output to list
        return round(numerator / denominator, precision)


def plot_acf(autocorrelation, title_of_plot, x_axis_label="Lags", y_axis_label="Magnitude"):
    # make a symmetric version of autocorrelation using slicing
    symmetric_autocorrelation = autocorrelation[:0:-1] + autocorrelation
    x_positional_values = [i * -1 for i in range(0, len(autocorrelation))][:0:-1] + [i for i in
                                                        range(0, len(autocorrelation))]
    # plot the symmetric version using stem
    plt.stem(x_positional_values, symmetric_autocorrelation, use_line_collection=True)
    plt.xlabel(x_axis_label)
    plt.ylabel(y_axis_label)
    plt.title(title_of_plot)
    plt.show()


def cal_mse(actual_values, predicted_values):
    # mean square error
    return np.round(np.mean(np.square(np.subtract(predicted_values, actual_values))), 3)


def cal_sse(actual_values, predicted_values):
    # sum square errors
    return np.round(np.sum(np.square(np.subtract(predicted_values, actual_values))), 3)


def cal_forecast_errors(actual_values, predicted_values):
    # forecast errors is difference between  observed values and predicted values
```

```python
    return np.subtract(actual_values, predicted_values)


def plot_multi_scatter_plot(list_of_y_data, list_of_x_data, list_of_legends, title_of_chart, x_axis_label,
y_axis_label,
                            list_of_colors, size_of_marker=50):
    """Plots multiple scatter plots on same chart"""
    for i in range(0, len(list_of_x_data)):
        plt.scatter(list_of_y_data[i], list_of_x_data[i], s=size_of_marker, c=list_of_colors[i])
    plt.xlabel(x_axis_label)
    plt.legend(list_of_legends)
    plt.ylabel(y_axis_label)
    plt.title(title_of_chart)
    plt.show()


def plot_multi_line_chart(list_of_y_data, x_common_data, list_of_colors, list_of_legends, title_of_chart,
                          x_axis_label, y_axis_label):
    """ Plots multiple lines on same chart, using common x-axis data"""
    for i in range(0, len(list_of_y_data)):
        # create line charts
        plt.plot(list_of_y_data[i], color=list_of_colors[i], label=list_of_legends[i], marker="o", linestyle="--")

        # add the x axis data
        plt.xticks(x_common_data)

    # set the x axis label
    plt.xlabel(x_axis_label)

    # set the y axis label
    plt.ylabel(y_axis_label)

    # create the legend
    plt.legend()

    # set the title of chart
    plt.title(title_of_chart)

    plt.show()


def box_pierce_test(number_of_samples, residuals, lags):
    """

    :param number_of_samples: Total number of samples in the data :param residuals: residuals are
difference between
    predicted and observed values :param lags: To perform autocorrelation we specify the lags (if h = 2,
    it means ignore zeroth and find first and second, to do this we add 1 to the lag and then ignore the 0th
ACF)
    :return: Q statistic rounded to 3 decimals
    """
    return round(number_of_samples * np.sum(np.square(cal_auto_correlation(residuals, lags + 1)[1:])), 3)
```

```python
def Q_value(y, autocorrelation_of_residuals):
    """ Computes Q value for comparing with chi_critical for Chi Square Test. Same as box_pierce_test(..)"""
    Q = len(y) * np.sum(np.square(autocorrelation_of_residuals[1:]))
    return Q


def generic_average_method(input_data, step_ahead):
    """Predicts the average value for the specified steps"""
    # returns a flat prediction
    return [np.round(np.mean(input_data), 3) for i in range(0, step_ahead)]


def generic_naive_method(input_data, step_ahead):
    """Predicts using naive method for specified steps"""
    return [input_data[-1] for i in range(0, step_ahead)]


def generic_drift_method(input_data, step_ahead):
    """Predicts using drift method for specified steps"""
    predicted_values = []

    for i in range(0, step_ahead):
        predicted_value = input_data[-1] + (i + 1) * ((input_data[-1] - input_data[0]) / (len(input_data) - 1))

        predicted_values.append(round(predicted_value, 3))

    return predicted_values


def generic_ses_method(input_data, step_ahead, alpha, initial_condition):
    """Predicts using SES method for specified steps. SES has a flat prediction curve and works best for
data with no
    trend and no seasonality """

    summation_part = 0
    for h in range(0, len(input_data)):
        summation_part += (alpha * ((1 - alpha) ** h)) * input_data[len(input_data) - h - 1]

    predicted_value = summation_part + ((1 - alpha) ** len(input_data)) * initial_condition

    return [round(predicted_value, 3) for i in range(0, step_ahead)]


def generic_holt_linear_trend(train_data, test_data):
    """ Works best for data with trend only"""
    holt_linear = ets.Holt(train_data).fit()
    predictions = list(holt_linear.forecast(len(test_data)))
    return predictions
```

```
def generic_holt_linear_winter(train_data, test_data, seasonal_period: int, trend="mul", seasonal="mul",
                              trend_damped=False):
    """ Works best for data with trend and seasonality"""
    holt_winter = ets.ExponentialSmoothing(train_data, trend=trend, seasonal=seasonal,
                                           seasonal_periods=seasonal_period, damped=trend_damped).fit()
    holt_winter_forecast = list(holt_winter.forecast(len(test_data)))
    return holt_winter_forecast


def split_df_train_test(df, test_size, random_seed=42):
    """ Test set size should be equal to the size of the prediction we want."""
    train, test = train_test_split(df, shuffle=False, test_size=test_size, random_state=random_seed)
    return train, test


def plot_multiline_chart_pandas_using_index(list_of_dataframes, y_axis_common_data, list_of_label,
list_of_color,
                                            x_label, y_label, title_of_plot, rotate_xticks=False):
    """Plots multiple line charts into single chart. This API uses list of pandas data having same x_axis label
and
    same y_axis label """
    for i, df in enumerate(list_of_dataframes):
        df[y_axis_common_data].plot(label=list_of_label[i], color=list_of_color[i])
    plt.legend(loc='best')
    plt.xlabel(x_label)
    plt.ylabel(y_label)
    plt.title(title_of_plot)
    if rotate_xticks:
        plt.xticks(rotation=90)
    plt.show()


def plot_multiline_chart_pandas(list_of_dataframes, x_axis_common_data, y_axis_common_data,
list_of_label,
                                list_of_color,
                                x_label, y_label, title_of_plot, rotate_xticks=False):
    """Plots multiple line charts into single chart. This API uses list of pandas data having same x_axis label
and
    same y_axis label """
    for i, df in enumerate(list_of_dataframes):
        plt.plot(df[x_axis_common_data], df[y_axis_common_data], label=list_of_label[i],
color=list_of_color[i])
    plt.legend(loc='best')
    plt.xlabel(x_label)
    plt.ylabel(y_label)
    plt.title(title_of_plot)
    if rotate_xticks:
        plt.xticks(rotation=90)
    plt.show()


def cal_standard_error(number_of_features, forecast_errors):
```

```python
    """Calculate standard deviation of error using the forecast residuals"""
    denominator = len(forecast_errors) - number_of_features - 1
    return np.sqrt(np.sum(np.square(forecast_errors)) / denominator)


def cal_variance_of_error(number_of_features, forecast_errors):
    """Calculate variance of error using the forecast residuals"""
    return np.square(cal_standard_error(number_of_features, forecast_errors))


def cal_r_squared(predicted_values, actual_values):
    """Calculates R-square value"""
    return np.square(correlation_coefficient_cal(predicted_values, actual_values))


def cal_adjusted_r_squared(predicted_values, actual_values, number_of_features):
    """Calculates adjusted r squared value"""
    r_squared = cal_r_squared(predicted_values, actual_values)
    return 1 - ((1 - r_squared) * ((len(predicted_values) - 1) / (len(predicted_values) - number_of_features -
1)))


def cal_95_confidence(predicted_values, std_error, x_matrix, intercept=True):
    """predicted values computed using predict_using_normal_equation_parameters,
       std_error computed using cal_standard_error(..)
       x_matrix is the test feature matrix
    """
    if intercept:
        # append ones if intercept present
        x_matrix = np.column_stack((np.ones(shape=x_matrix.shape[0]), x_matrix))

    confidence_tuples = []
    for i in range(0, len(predicted_values)):
        interval = 1.96 * std_error * (np.sqrt(
            1 + np.dot(np.dot(x_matrix[i], np.linalg.inv(np.dot(x_matrix.transpose(), x_matrix))),
                       np.vstack(x_matrix[i]))))

        confidence_tuples.append([predicted_values[i] - interval, predicted_values[i] + interval])

    return confidence_tuples


def normal_equation_regression(train_df, target_label: object, intercept=True):
    """Performs linear regression using the normal equation"""
    if intercept:
        x_train = np.column_stack(
            (np.ones(shape=len(train_df)), train_df[np.setdiff1d(train_df.columns, target_label)]))
    else:
        x_train = train_df[np.setdiff1d(train_df.columns, target_label)]

    y_train = np.vstack(train_df[target_label])
```

```
    normal_equation_coefficient = np.round(np.dot(
        np.dot(np.linalg.inv(np.dot(x_train.transpose(), x_train)), x_train.transpose()), y_train), 3)

    return normal_equation_coefficient.flatten()


def predict_using_normal_equation_parameters(test_data_nested_list, normal_equation_coefficient_list,
intercept=True):
    """Predict the test data inputs based on normal equation containing the intercepts and parameters of
input
    variable"""

    if intercept:
        # add the intercept
        predicted_values = normal_equation_coefficient_list[0]
        start_index = 1
    else:
        predicted_values = 0
        start_index = 0

    for i in range(len(test_data_nested_list)):
        predicted_values += test_data_nested_list[i] * normal_equation_coefficient_list[i + start_index]

    return list(predicted_values)


def normal_equation_using_statsmodels(train_feature_list, train_target_list, intercept=True):
    if intercept:
        train_feature_list = sm.add_constant(train_feature_list)

    model = sm.OLS(train_target_list, train_feature_list)
    results = model.fit()
    return results


def normal_equation_prediction_using_statsmodels(OLS_model, test_feature_list, intercept=True):
    if intercept:
        test_feature_list = sm.add_constant(test_feature_list, has_constant='add')

    predicted_values_OLS = OLS_model.predict(test_feature_list)
    return predicted_values_OLS


def cyclic_shift(input_list, shift_by, clip_by):
    """Shifts the array to the left by amount specified in shift_by variables. The input array shrinks by clip_by
due
    to loss in data point; which is a caveat of autoregression """
    return np.roll(input_list, -shift_by)[:-clip_by]


def autoregression_data_prepper(y_input_list, order_of_auto_regressor):
    """Prepares the input array for autoregression by creating data with order_of_auto_regressor shift
```

```
    order_of_auto_regressor = n_a
    """

    prepared_data = pd.DataFrame()

    for i in range(1, order_of_auto_regressor + 1):
        shifted_data = cyclic_shift(y_input_list, order_of_auto_regressor - i, order_of_auto_regressor)
        prepared_data["x(" + str(i) + ")"] = np.multiply(shifted_data, -1)

    prepared_data["y(t)"] = y_input_list[order_of_auto_regressor:]

    return prepared_data


def generate_auto_regressor_data(number_of_samples, initial_condition, parameter_list,
mean_of_white_noise=0,
                                 std_dev_of_white_noise=1, seed=0):
    """Generates white noise and then creates AR data based on the order which is inferred from size of
    parameter_list """
    np.random.seed(seed)
    white_noise = np.random.normal(mean_of_white_noise, std_dev_of_white_noise, number_of_samples)
    y = np.zeros(shape=len(white_noise))

    # multiply by -1 since we take AR coefficients to the RHS
    parameter_list = np.multiply(parameter_list, -1)

    for t in range(len(white_noise)):

        temp = 0

        # iterate over each coefficient
        for i in range(len(parameter_list)):
            # if the index goes below zero then use initial condition
            if (t - (i + 1)) < 0:
                temp += parameter_list[i] * initial_condition
            else:
                temp += parameter_list[i] * y[t - (i + 1)]

        # add white noise
        y[t] = temp + white_noise[t]

    return y


def generate_moving_averages_data(number_of_samples, initial_condition, parameter_list,
mean_of_white_noise=0,
                                  std_dev_of_white_noise=1, seed=0):
    """Generates white noise and then creates MA data based on the order which is inferred from size of
    parameter_list """
    np.random.seed(seed)
    white_noise = np.random.normal(mean_of_white_noise, std_dev_of_white_noise, number_of_samples)
    y = np.zeros(shape=len(white_noise))
```

```
    for t in range(len(white_noise)):

        # add white noise
        temp_sum = white_noise[t]

        # iterate over each coefficient
        for i in range(len(parameter_list)):
            # if the index goes below zero then use initial condition
            if (t - (i + 1)) < 0:
                temp_sum += parameter_list[i] * initial_condition
            else:
                temp_sum += parameter_list[i] * white_noise[t - (i + 1)]

        # store value in list
        y[t] = temp_sum

    return y


def generate_arma_data(number_of_samples, initial_condition, parameter_list_ar, parameter_list_ma,
                       mean_of_white_noise=0, std_dev_of_white_noise=1, seed=0):
    """Generates white noise and then creates ARMA data based on the order of AR which is inferred from
size of
    parameter_list_ar and order of MA which is inferred from size of parameter_list_ma"""

    np.random.seed(seed)

    white_noise = np.random.normal(mean_of_white_noise, std_dev_of_white_noise, number_of_samples)
    y = np.zeros(shape=len(white_noise))

    # multiply by -1 since we take AR coefficients to the RHS
    parameter_list_ar = np.multiply(parameter_list_ar, -1)

    for t in range(len(white_noise)):

        # add white noise
        temp_sum = white_noise[t]

        # iterate over each coefficient of AR process [denominator]
        for i in range(len(parameter_list_ar)):
            # if the index goes below zero then use initial condition
            if (t - (i + 1)) < 0:
                temp_sum += parameter_list_ar[i] * initial_condition
            else:
                temp_sum += parameter_list_ar[i] * y[t - (i + 1)]

        # iterate over each coefficient of MA process [numerator]
        for i in range(len(parameter_list_ma)):
            # if the index goes below zero then use initial condition
            if (t - (i + 1)) < 0:
                temp_sum += parameter_list_ma[i] * initial_condition
```

```python
            else:
                temp_sum += parameter_list_ma[i] * white_noise[t - (i + 1)]

        # store value in list
        y[t] = temp_sum

    return y


def generate_arma_data_user_input():
    # Generates ARMA(na,nb) process using inputs from the user
    print()
    number_of_samples = int(input("Enter the number of data samples:"))

    ar_order = int(input("Enter the order of AR portion:"))
    ar_coefficients = []
    for order in range(ar_order):
        ar_coefficients.append(float(input("Enter coefficient excluding coefficient for y(t) and the sign of "
                                           "coefficients \n "
                                           "should be entered as though the coefficients are on LHS of ARMA equation")))

    ma_order = int(input("Enter the order of MA portion:"))
    ma_coefficients = []
    for order in range(ma_order):
        ma_coefficients.append(float(input("Enter coefficients excluding coefficient for e(t) and press enter")))

    # set seed
    seed = int(input("Enter the seed for random data:"))

    return generate_arma_data(number_of_samples, 0, ar_coefficients, ma_coefficients, 0, 1, seed)


def perform_auto_regression():
    """Performs auto regression using input from user"""
    print()
    number_of_samples = int(input("Enter number of samples:\n"))
    order_of_auto_regressor = int(input("Enter the order # of the AR process:\n"))

    parameter_list = []
    print("Enter coefficients excluding coefficient for y(t) and the sign of coefficients "
          "should be entered as though the coefficients are on LHS of AR equation")
    for i in range(order_of_auto_regressor):
        parameter_list.append(float(input()))

    intercept_str = input("Do you want intercept? (Y/N)")
    intercept = True if intercept_str.lower() == "y" else False

    y = generate_auto_regressor_data(number_of_samples, 0, parameter_list)
    train_df = autoregression_data_prepper(y, order_of_auto_regressor)
    coefficients = normal_equation_regression(train_df, "y(t)", intercept)

    return parameter_list, coefficients
```

```python
def get_max_denominator_indices(j, k_scope):
    # create denominator indexes based on formula for GPAC
    denominator_indices = np.zeros(shape=(k_scope, k_scope), dtype=np.int64)

    for k in range(k_scope):
        denominator_indices[:, k] = np.arange(j - k, j + k_scope - k)

    return denominator_indices


def get_apt_denominator_indices(max_denominator_indices, k):
    apt_denominator_indices = max_denominator_indices[-k:, -k:]
    return apt_denominator_indices


def get_numerator_indices(apt_denominator_indices, k):
    numerator_indices = np.copy(apt_denominator_indices)
    # take the 0,0 indexed value and then create a range of values from (indexed_value+1,
indexed_value+k)
    indexed_value = numerator_indices[0, 0]
    y_matrix = np.arange(indexed_value + 1, indexed_value + k + 1)

    # replace the last column with this new value
    numerator_indices[:, -1] = y_matrix

    return numerator_indices


def get_ACF_by_index(numpy_indices, acf):
    # select values from an array based on index specified
    result = np.take(acf, numpy_indices)
    return result


def get_phi_value(denominator_indices, numerator_indices, ry, precision=5):
    # take the absolute values since when computing phi value, we use ACF and ACF is symmetric in nature
    denominator_indices = np.abs(denominator_indices)
    numerator_indices = np.abs(numerator_indices)

    # replace the indices with the values of ACF
    denominator = get_ACF_by_index(denominator_indices, ry)
    numerator = get_ACF_by_index(numerator_indices, ry)

    # take the determinant
    denominator_det = np.round(np.linalg.det(denominator), precision)
    numerator_det = np.round(np.linalg.det(numerator), precision)

    # divide it and return the value of phi
    return np.round(np.divide(numerator_det, denominator_det), precision)
```

```
def create_gpac_table(j_scope, k_scope, ry, precision=5):
    # initialize gpac table
    gpac_table = np.zeros(shape=(j_scope, k_scope), dtype=np.float64)

    for j in range(j_scope):
        # create the largest denominator
        max_denominator_indices = get_max_denominator_indices(j, k_scope)

        for k in range(1, k_scope + 1):
            #  slicing largest denominator as required
            apt_denominator_indices = get_apt_denominator_indices(max_denominator_indices, k)

            # for numerator replace denominator's last columnn with index starting from j+1 upto k times
            numerator_indices = get_numerator_indices(apt_denominator_indices, k)

            # compute phi value
            phi_value = get_phi_value(apt_denominator_indices, numerator_indices, ry, precision)
            gpac_table[j, k - 1] = phi_value

    gpac_table_pd = pd.DataFrame(data=gpac_table, columns=[k for k in range(1, k_scope + 1)])

    return gpac_table_pd


def cal_t_test_correlation_coefficient(correlation_coefficient, number_of_observations,
number_of_confounding_variables,
                                       alpha_level, two_tail=False):
    degree_of_freedom = number_of_observations - 2 - number_of_confounding_variables
    t_value = np.abs(
        correlation_coefficient * np.sqrt(np.divide(degree_of_freedom, 1 -
np.square(correlation_coefficient))))
    # t value from t table
    if two_tail:
        alpha_level = alpha_level / 2
    critical_t_test = t.ppf(1 - alpha_level, degree_of_freedom)

    print()
    if t_value > critical_t_test:
        print(
            f"The absolute value of test statistic {t_value} exceeded the critical t-value {critical_t_test} from the
table; "
            f"hence the correlation coefficient (partial correlation) is statistically significant.")
    else:
        print(
            f"The absolute value of test statistic {t_value} did not exceed the critical t-value {critical_t_test} from
the table;\n "
            f"hence the correlation coefficient (partial correlation) is not statistically significant.")

    return t_value, critical_t_test
```

```python
def cal_partial_correlation(a, b, c):
    r_ab = correlation_coefficient_cal(a, b)
    r_ac = correlation_coefficient_cal(a, c)
    r_bc = correlation_coefficient_cal(b, c)

    return np.divide(r_ab - r_ac * r_bc, np.multiply(np.sqrt(1 - np.square(r_ac)), np.sqrt(1 - np.square(r_bc))))


# LMA steps
def LMA_step_0(n_a, n_b):
    # initially theta are all zeroes
    theta = np.zeros(shape=(n_a + n_b, 1))
    return theta.flatten()


def LMA_step_1(n_a, n_b, theta, delta, y):
    X_list = []

    # compute e using theta
    e = extract_white_noise_from_y(n_a, theta, y)

    # sse_old
    sse_old = np.matmul(np.transpose(e), e)

    # add delta to each theta values
    for i in range((n_a + n_b)):
        # theta_second = theta.deepcopy()
        theta_second = theta.copy()
        theta_second[i] = theta[i] + delta

        # compute e with the modified theta value
        e_second = extract_white_noise_from_y(n_a, theta_second, y)

        x_i = (e - e_second) / delta
        X_list.append(x_i)

    X = np.column_stack(X_list)

    A = np.matmul(np.transpose(X), X)
    g = np.matmul(np.transpose(X), e)

    return A, g, sse_old


def extract_num_den_from_theta(theta, n_a):
    theta_ar = theta.copy()
    theta_ma = theta.copy()
    # ar
    ar = np.insert(theta_ar[:n_a], 0, 1)
    # ma
    ma = np.insert(theta_ma[n_a:], 0, 1)
```

```
    # pad zeroes
    if len(ar) < len(ma):
        ar = np.append(ar, [0 for i in range(len(ma) - len(ar))])
    elif len(ma) < len(ar):
        ma = np.append(ma, [0 for i in range(len(ar) - len(ma))])

    return ar, ma


def extract_white_noise_from_y(n_a, theta, y):
    den_ar, num_ma = extract_num_den_from_theta(theta, n_a)

    # extract white noise from the given data of y(t)
    system = (den_ar, num_ma, 1)
    extracted_white_noise = dlsim(system, y)[1].flatten()

    return extracted_white_noise


def LMA_step_2(mu, A, g, n_a, n_b, theta, y):
    # we are updating theta in Step 2

    # compute delta_theta and return it
    mu_identity = np.multiply(mu, np.identity(n_a + n_b))
    delta_theta = np.round(np.matmul(np.linalg.inv(A + mu_identity), g), 8)

    # compute theta_new
    theta_new = np.add(theta.flatten(), delta_theta.flatten())

    extracted_white_noise = extract_white_noise_from_y(n_a, theta_new, y)

    # compute SSE_new using theta_new
    sse_new = np.matmul(extracted_white_noise.transpose(), extracted_white_noise)

    # # check if SSE_new is NaN
    if np.isnan(sse_new):
        sse_new = 10 ** 10

    return delta_theta, sse_new, theta_new


def LMA_step_3(y, sse_old, sse_new, n_a, n_b, mu, delta, delta_theta, A, g, theta_new, mu_max=10 **
27,
               MAX_ITERATIONS=70):
    iterations = 0
    iteration_list = []
    sse_list = []

    while iterations < MAX_ITERATIONS:
        # keeping track of iterations and sse new
        iteration_list.append(iterations)
        sse_list.append(sse_new)
```

```
    if sse_new < sse_old:
        if np.linalg.norm(delta_theta) < 10 ** -3:

            theta = theta_new
            var_error = sse_new / (len(y) - (n_a + n_b))
            covariance_theta = np.multiply(var_error, np.linalg.inv(A))
            return theta, var_error, covariance_theta, pd.DataFrame({"SSE": sse_list, "Iteration":
iteration_list})

        else:
            theta = theta_new
            mu /= 10

    # theta is theta_old
    theta = theta_new

    # return to step 1
    A, g, sse_old = LMA_step_1(n_a, n_b, theta, delta, y)

    # return to step 2
    delta_theta, sse_new, theta_new = LMA_step_2(mu, A, g, n_a, n_b, theta, y)

    while sse_new >= sse_old:
        mu *= 10
        if mu > mu_max:
            print("MU Error")
            return None, None, None, None

        # return to step 2
        delta_theta, sse_new, theta_new = LMA_step_2(mu, A, g, n_a, n_b, theta, y)

    iterations += 1

    if iterations > MAX_ITERATIONS:
        print("Iterations Error")
        return None, None, None, None


def perform_LMA_parameter_estimation(n_a, n_b, y, seed=42):
    np.random.seed(seed)
    # step 0
    # theta are unknown parameters
    theta = LMA_step_0(n_a, n_b)

    # step 1
    delta = 10 ** -6
    A, g, sse_old = LMA_step_1(n_a, n_b, theta, delta, y)

    # step 2
    mu = 0.01
    delta_theta, sse_new, theta_new = LMA_step_2(mu, A, g, n_a, n_b, theta, y)
```

```python
    # step 3
    return LMA_step_3(y, sse_old, sse_new, n_a, n_b, mu, delta, delta_theta, A, g, theta_new)


def compute_confidence_interval(estimated_parameters, covariance_matrix, n):
    confidence_interval = []

    estimated_parameters = list(estimated_parameters)
    covariance_matrix = covariance_matrix.reset_index(drop=True)
    covariance_matrix.columns = [i for i in range(covariance_matrix.shape[0])]

    for i in range(n):
        upper = estimated_parameters[i] + 2 * np.sqrt(covariance_matrix[i][i])
        lower = estimated_parameters[i] - 2 * np.sqrt(covariance_matrix[i][i])
        confidence_interval.append([upper, lower])

        # For printing the results
        print(f"{lower} < {estimated_parameters[i]} < {upper}")

    return confidence_interval


def plot_survival_curve(duration_list: list, event_list: list, label_list: list, title_of_chart=object):
    if len(duration_list) == len(event_list):
        kmf = KaplanMeierFitter()
        for i in range(len(duration_list)):
            kmf.fit(durations=duration_list[i], event_observed=event_list[i], label=label_list[i])
            kmf.plot()
        plt.title(title_of_chart)
        plt.show()
    else:
        print("Duration and event list size are not same, thus cannot create survival plot.")


def plot_heatmap(corr_df, title, xticks=None, yticks=None, x_axis_rotation=0, annotation=True):
    sns.heatmap(corr_df, annot=annotation)
    plt.title(title)
    if xticks is not None:
        plt.xticks([i for i in range(len(xticks))], xticks, rotation=x_axis_rotation)
    if yticks is not None:
        plt.yticks([i for i in range(len(yticks))], yticks)
    plt.show()


def chi_square_test(Q, lags, n_a, n_b, alpha=0.01):
    dof = lags - n_a - n_b
    chi_critical = chi2.isf(alpha, df=dof)

    if Q < chi_critical:
        print(f"The residual is white and the estimated order is n_a= {n_a} and n_b = {n_b}")
    else:
```

```
        print(f"The residual is not white with n_a={n_a} and n_b={n_b}")

    return Q < chi_critical


# -------------------------- statsmodels related wrapper classes-----------------------------
def statsmodels_estimate_parameters(n_a, n_b, y, trend="nc"):
    model = sm.tsa.ARMA(y, (n_a, n_b)).fit(trend=trend, disp=0)
    return model


def statsmodels_print_parameters(model, n_a, n_b):
    # print the parameters which are estimated
    for i in range(n_a):
        print("The AR coefficients a {}".format(i), "is:", model.params[i])
    print()
    for i in range(n_b):
        print("The MA coefficients b {}".format(i), "is:", model.params[i + n_a])
    print()


def statsmodels_print_covariance_matrix(model, n_a, n_b):
    print(f"Estimated covariance matrix for n_a = {n_a} and n_b = {n_b}: \n{model.cov_params()}")
    print()
    return model.cov_params()


def statsmodels_print_variance_error(model, n_a, n_b):
    print(f"Estimated variance of error for n_a = {n_a} and n_b = {n_b}: \n{model.sigma2}")
    print()
    return model.sigma2


def statsmodels_print_confidence_interval(model, n_a, n_b):
    # confidence interval
    print(
        f"The confidence interval for estimated parameters for n_a = {n_a} and n_b = {n_b}: \n
{model.conf_int()}")
    print()
    return model.conf_int()


def statsmodels_predict_ARMA_process(model, start, stop):
    model_hat = model.predict(start=start, end=stop)
    return model_hat


def statsmodels_plot_predicted_true(y, model_hat, n_a, n_b):
    true_data = pd.DataFrame({"Magnitude": y, "Samples": [i for i in range(len(y))]})
    fitted_data = pd.DataFrame({"Magnitude": model_hat, "Samples": [i for i in range(len(model_hat))]})

    plot_multiline_chart_pandas([true_data, fitted_data], "Samples", "Magnitude", ["True data", "Fitted
```

```
data"],
                          ["red", "blue"], "Samples", "Magnitude",
                          f"ARMA process with n_a={n_a} and n_b={n_b}")


def statsmodels_print_roots_AR(model):
    print("Real part:")
    for root in model.arroots:
        print(root.real)
    print("Imaginary part:")
    for root in model.arroots:
        print(root.imag)


def statsmodels_print_roots_MA(model):
    print("Real part:")
    for root in model.maroots:
        print(root.real)
    print("Imaginary part:")
    for root in model.maroots:
        print(root.imag)


# check whether order passes chi square test
def gpac_order_chi_square_test(possible_order_ARMA, train_data, start, stop, lags, actual_outputs):
    results = []

    for n_a, n_b in possible_order_ARMA:
        try:
            # estimate the model parameters
            model = statsmodels_estimate_parameters(n_a, n_b, train_data)

            # predict the traffic_volume on test data
            # performing h step predictions
            predictions = statsmodels_predict_ARMA_process(model, start=start, stop=stop)

            # # add mean back to the forecast values
            # predictions = np.add(mean_to_add, predictions)

            # calculate forecast errors
            residuals = cal_forecast_errors(actual_outputs, predictions)

            # autocorrelation of residuals
            re = cal_auto_correlation(residuals, lags)

            # compute Q value for chi square test
            Q = Q_value(actual_outputs, re)

            # checking the chi square test
            if chi_square_test(Q, lags, n_a, n_b):
                results.append((n_a, n_b))
```

```python
    except Exception as e:
        # print(e)
        pass

return results
```