WIND TURBINE ELECTRICITY PREDICTION

Anwesha Tomar
DATS 6450 Dr Reza Jafari

Table of Contents

Abstract:

In this project we make use of a Wind Turbine dataset to predict the Active Power generated by the wind turbine. As of now we make use of fossil fuels to be able to power our cities. If this same consumption can be satisfied by wind energy we will be able to stop using fossil fuels and in turn reduce our carbon footprint. We code various models such as average, naive, drift, seasonal exponential smoothing, Holt Winter, Holt Winter Seasonal, ARMA and ARIMA. The performance of these models is compared and the best model is recommended for forecast.

Introduction:

We are predicting the value of Active Power generated from the wind turbines. If we can predict the amount of energy generated it can be easily estimated how much the energy should be charged or what it should cost.

To be able to get these predictions we code average, naive, drift, seasonal exponential smoothing, Holt winter, ARMA and ARIMA to be able to calculate future values. In the average model future values are based on the average of the model. In the naive model we use the value of the training data and use it to predict the future values. The values of the drift model are calculated by plotting a line from the last point and projecting it to all future values. In Holt winter we use the multiplicative or additive models to fit the data and then make predictions. Next we develop the ARMA model by using the GPAC code to find the value of the order.

Once all the models are created we compare the performance of the models and recommend the best model.

Dataset:

The dataset consists of 5000 points. The dataset contains the following columns:

- Date/Time, LV ActivePower (kW)
- Wind Speed (m/s)
- Theoretical_Power_Curve (KWh)
- Wind Direction (°)

After running the ADF test we can see that the data is stationary.

ADF Statistic: -3.994378
p-value: 0.001440
Critical Values:
    1%: -3.432
    5%: -2.862
    10%: -2.567

We can see that the p value is very small and lower than 0.05, thus the data is definitely stationary.

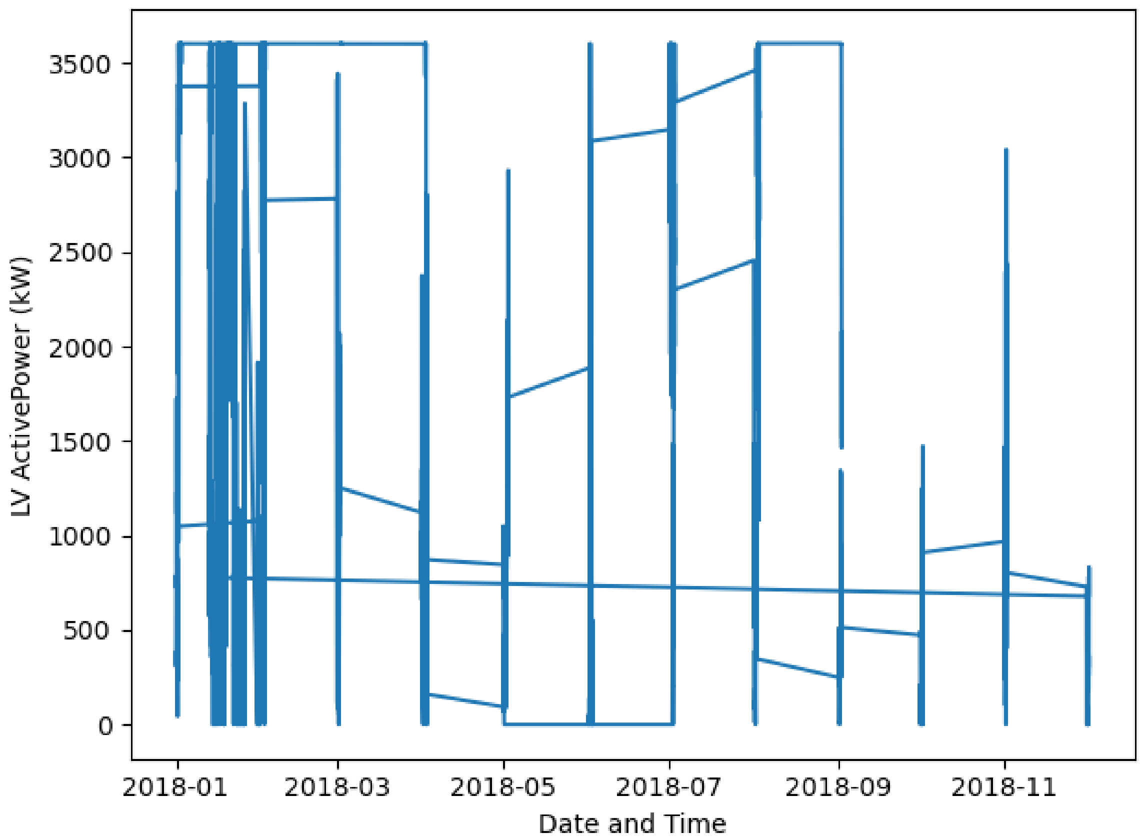


Fig 1. Plot of target value vs time

As you can see it is very difficult to understand the progression of the data as the data is recorded over 10 minute intervals over the span of a year it becomes extremely difficult to plot around 5000 data points and get some inference from it. Therefore, to get a better understanding of the data I subset a couple days to give an example of what the data looks like.

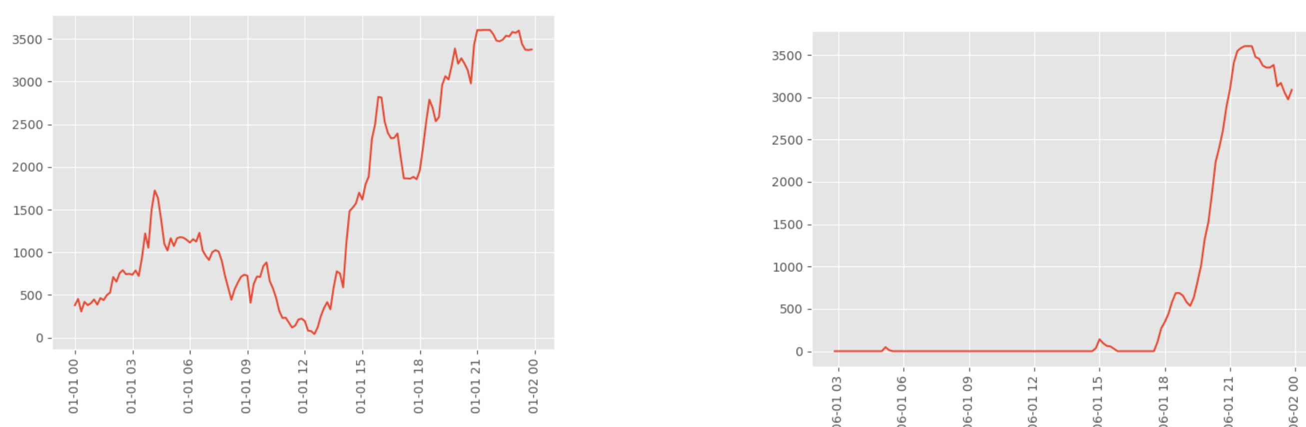
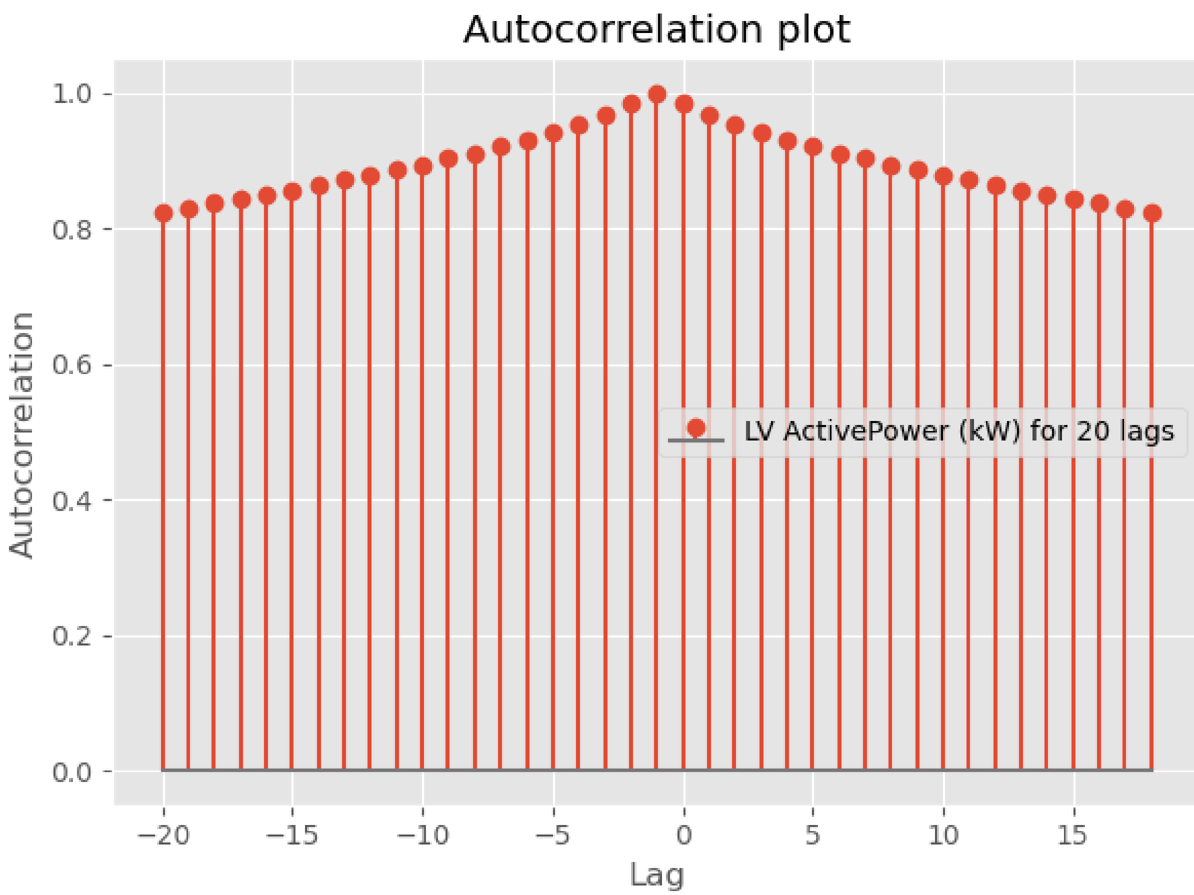


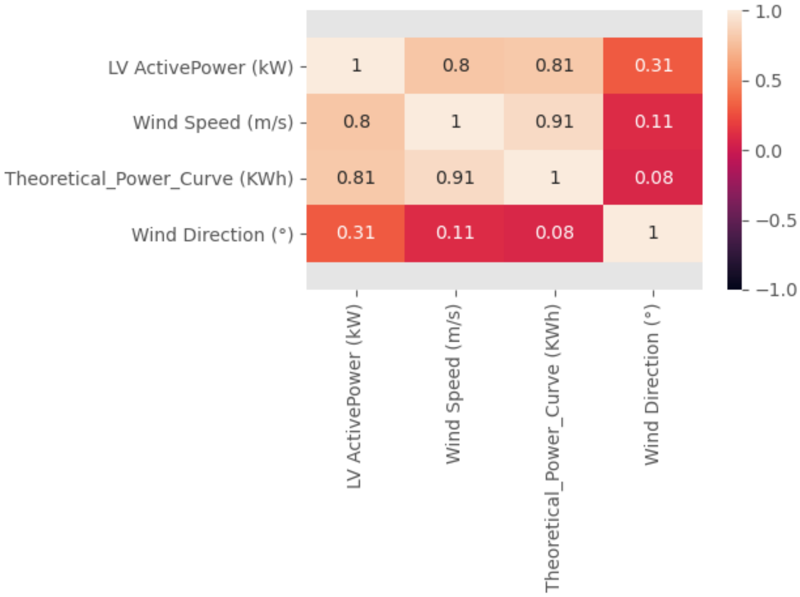
Fig 2. Plot of Daily data to show data distribution

The ACF plot of the data:



We can see that the data does not represent white noise so we can move forward with our prediction and analysis.

Correlation heat map:



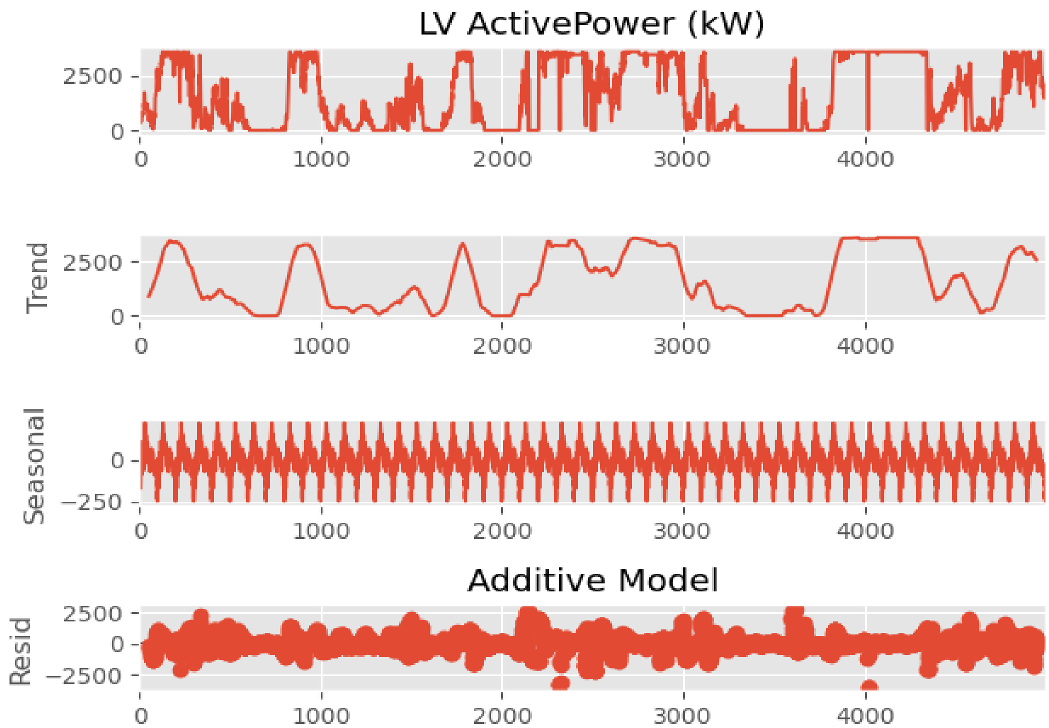
The target value is LV Active Power, as we can see from the Wind Speed and Theoretical Power Curve have a strong positive correlation with the target value while. Wind Direction has a low positive correlation. On exploring the data further it was noticed that Theoretical Power Curve is the amount of power that should be generated while LV Active Power is the amount of power that was actually generated. This means that the target variable has a strong multicollinearity with Theoretical Power Curve, this is confirmed by the correlation coefficient.

Pearsons Ceofficient:

Verifies above mentioned results

Correlation between: LV ActivePower (kW)-Wind Speed (m/s)
0.8010269972589192
Correlation between: LV ActivePower (kW)-Theoretical_Power_Curve (KWh)
0.8144311570359695
Correlation between: LV ActivePower (kW)-Wind Direction (°)
0.3115103700278061

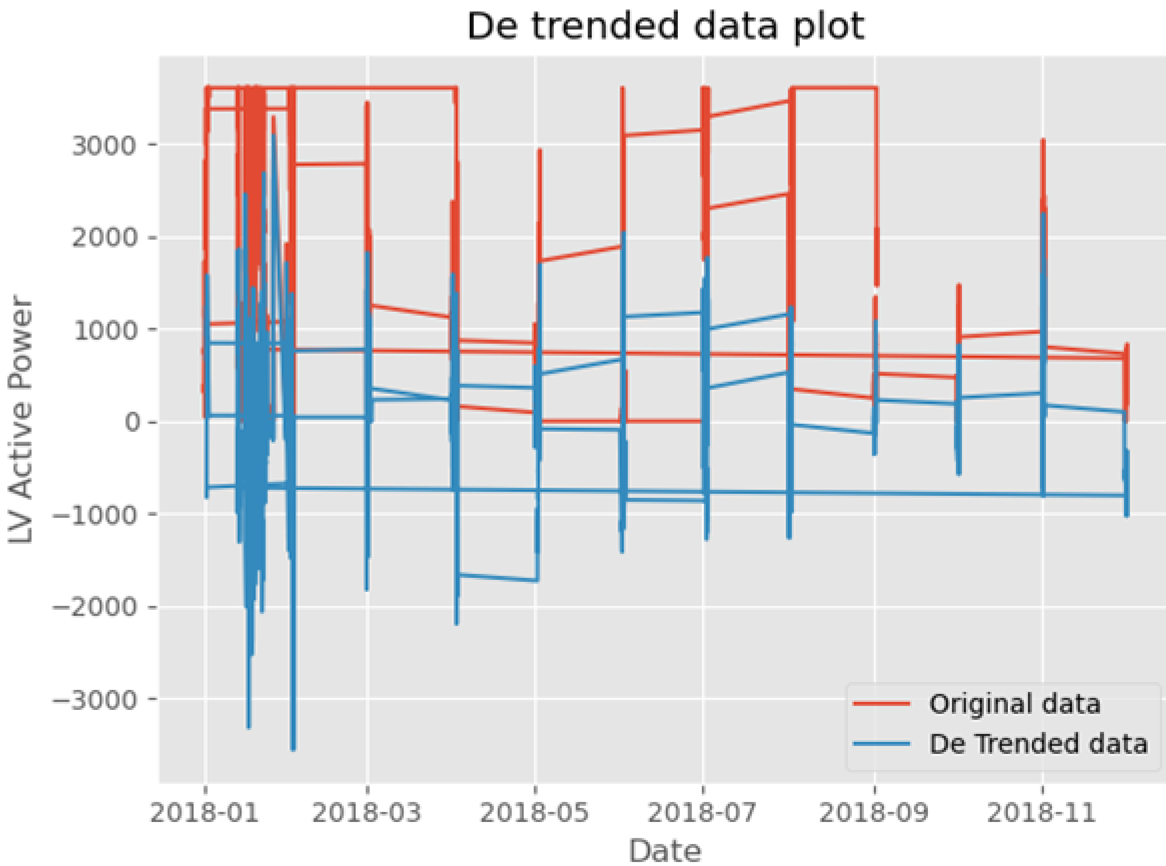
Time Series Decomposition:



The strength of trend for data set is: 0.7042208166148513
The strength of seasonality for data set is: 0.025644565545394293

The data has a very high trend but a very low seasonality, this means that we will not implement the SARIMA model for this dataset.

Plotted below we see the detreded data which further confirms that the data has a very high trend.



Feature Selection:

Using Wind Speed (m/s), Theoretical_Power_Curve (KWh),Wind Direction (°)-

Metrics:
Adj R2 0.7388687674669445
AIC 80454.51336522578
BIC 80480.58213799144
F statistic 4715.877053372894
F p-value 0.0

Using Wind Speed (m/s), Theoretical_Power_Curve (KWh)-

Metrics:
Adj R2 0.6835165521748432

AIC 81414.7533410593
BIC 81434.30492063356
F statistic 5399.227407788051
F p-value 0.0

Using Wind Speed (m/s)-

Metrics:
Adj R2 0.6835165521748432
AIC 81414.7533410593
BIC 81434.30492063356
F statistic 5399.227407788051
F p-value 0.0

We notice that the Adj R2 value drops after we eliminate Wind Direction as we want the value of Adj R2 to be high we see it is highest with all three variables. Since Theoretical Power Curve introduces multicollinearity, we drop that variable. All in all we can only use two features: Wind Speed and Wind Direction.

Multiple Linear Regression:

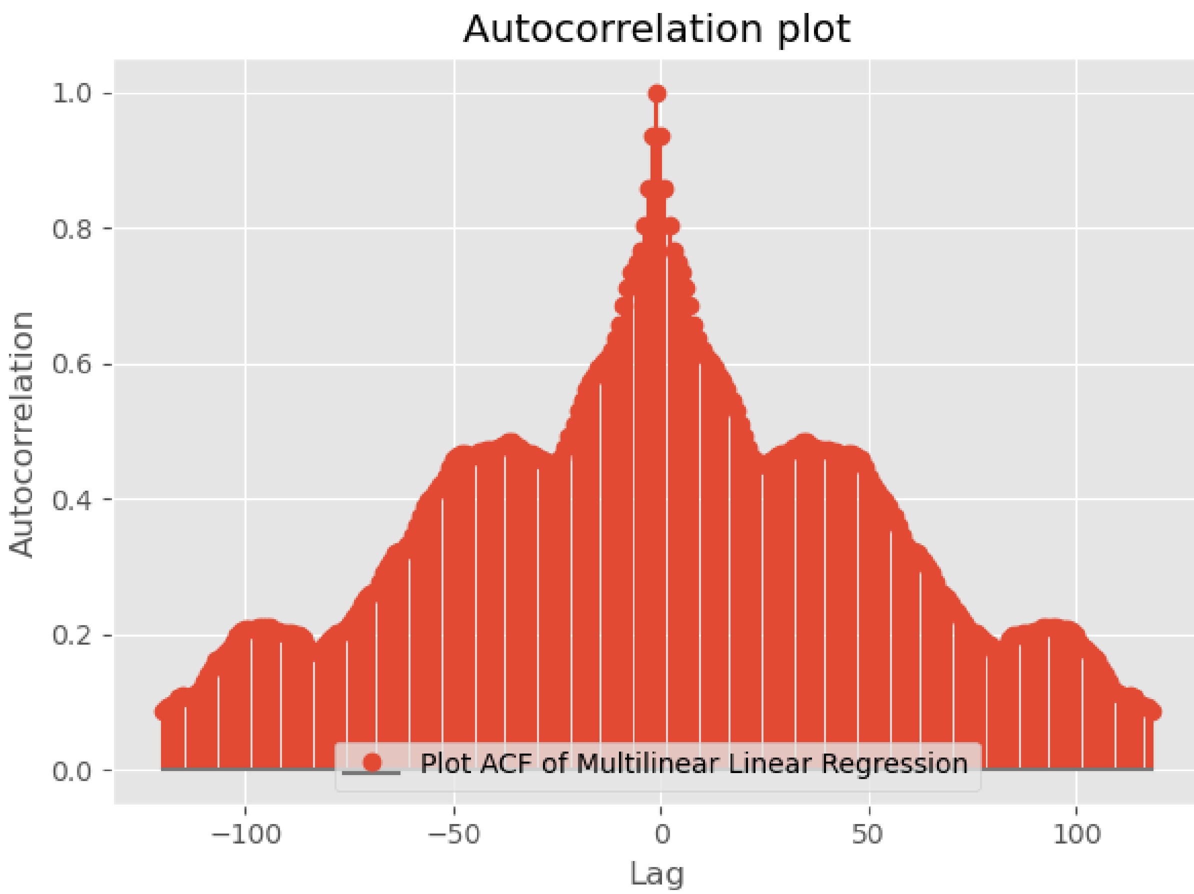
```
                            OLS Regression Results
==============================================================================
Dep. Variable:     LV ActivePower (kW)   R-squared:                       0.659
Model:                             OLS   Adj. R-squared:                  0.659
Method:                  Least Squares   F-statistic:                     3870.
Date:                 Wed, 09 Dec 2020   Prob (F-statistic):               0.00
Time:                         14:25:21   Log-Likelihood:                -32658.
No. Observations:                 4000   AIC:                         6.532e+04
Df Residuals:                     3997   BIC:                         6.534e+04
Df Model:                            2
Covariance Type:             nonrobust
=====================================================================================
                        coef    std err          t      P>|t|      [0.025      0.975]
-------------------------------------------------------------------------------------
const             -1416.4066     37.675    -37.596      0.000   -1490.270   -1342.543
Wind Speed (m/s)    238.5783      2.903     82.194      0.000     232.888     244.269
Wind Direction (°)    4.4585      0.155     28.856      0.000       4.156       4.761
==============================================================================
Omnibus:                      394.040   Durbin-Watson:                   0.112
Prob(Omnibus):                  0.000   Jarque-Bera (JB):              628.862
Skew:                          -0.718   Prob(JB):                    2.78e-137
Kurtosis:                       4.309   Cond. No.                         511.
```

Next we perform multiple linear regression, using Wind Speed and Wind Direction. The Prob of F statistics is lower than 0.05, thus the model performs a lot better than the null model and passes the F test.

Adj R2 0.659266273005014
AIC 65321.85974934304
BIC 65340.741898263346
F statistic 3869.718616437177
F p-value 0.0
Q value 40.16520043050357
Mean of residue 177.6439324653744
Variance of residue 457702.62080746226

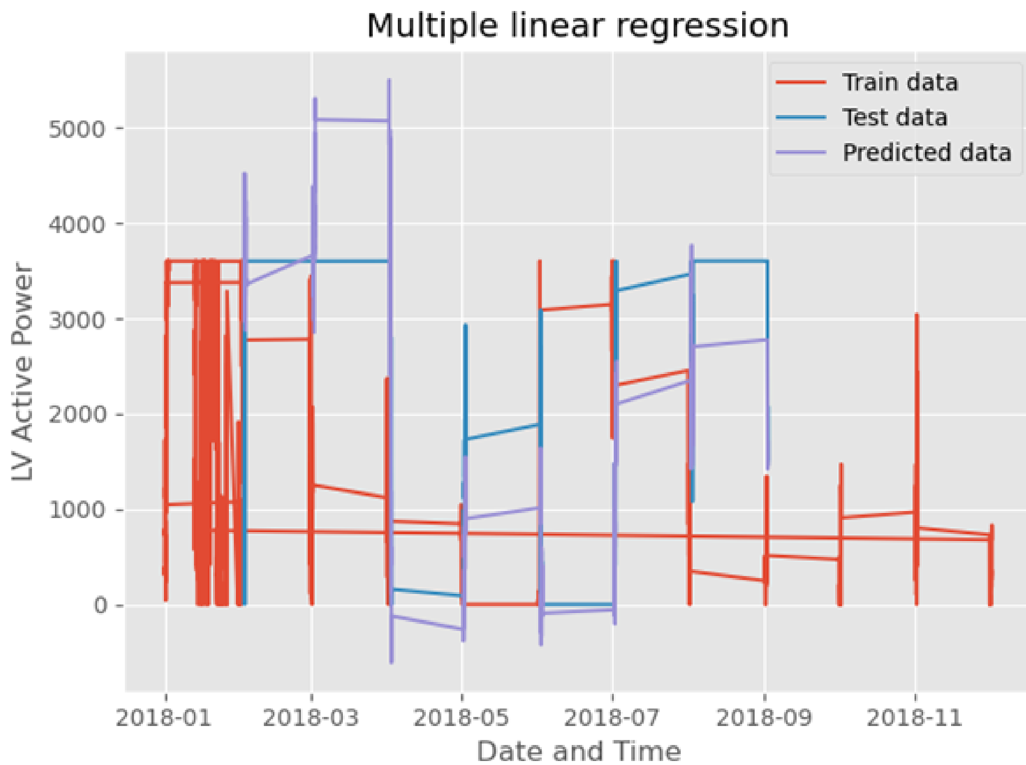
The value of P(t) is less than 0.05 therefore the model passes the T test values.

ACF plot of Multiple Linear Regression model:



The ACF continues to decay but it does not represent white noise.

The h step forecast for Multiple Linear Regression



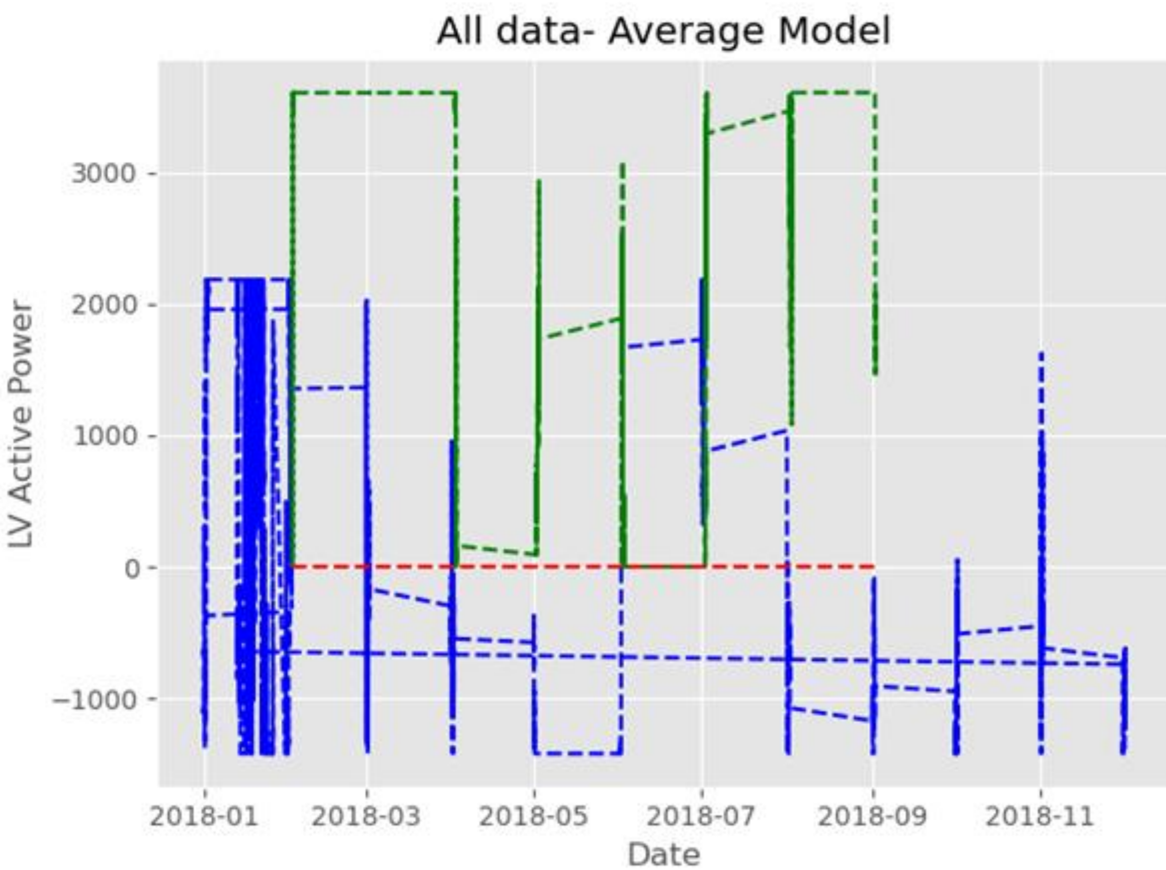
Modelling:

The **train data** is represented by **blue**, the **test data** is represented by **green** and **forecast values** are represented by **red**. For all the models we have **subtracted the mean** of the LV Active Power to code the models we then add the mean values back to the output of the models.

Base Models:

Average Model:

We find the h step prediction of the test data and find statistics based on the output.

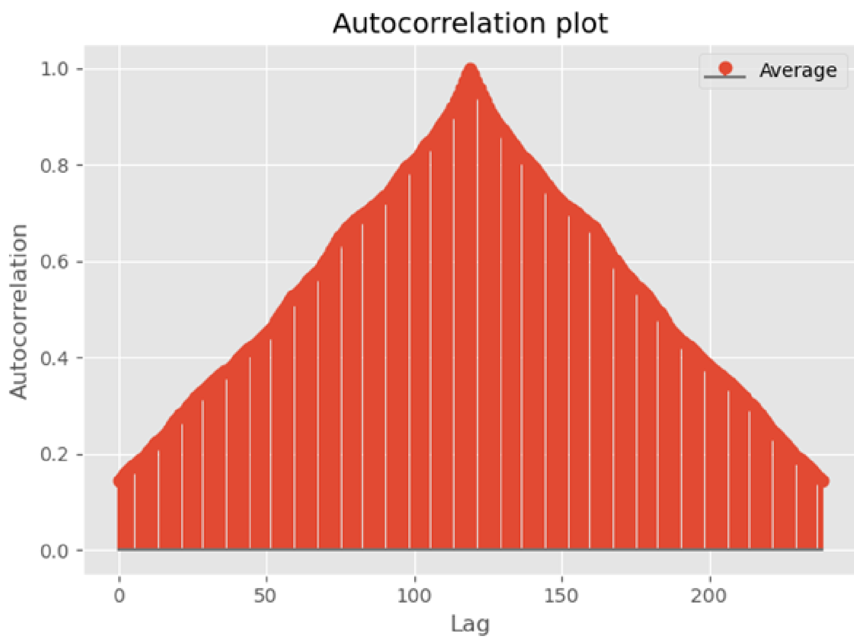


Mean of error of Average Model: 105.49108012674635

Variance of error of Average Model 1941719.830479737

MSE of Average Base Model: 1952848.1984660444

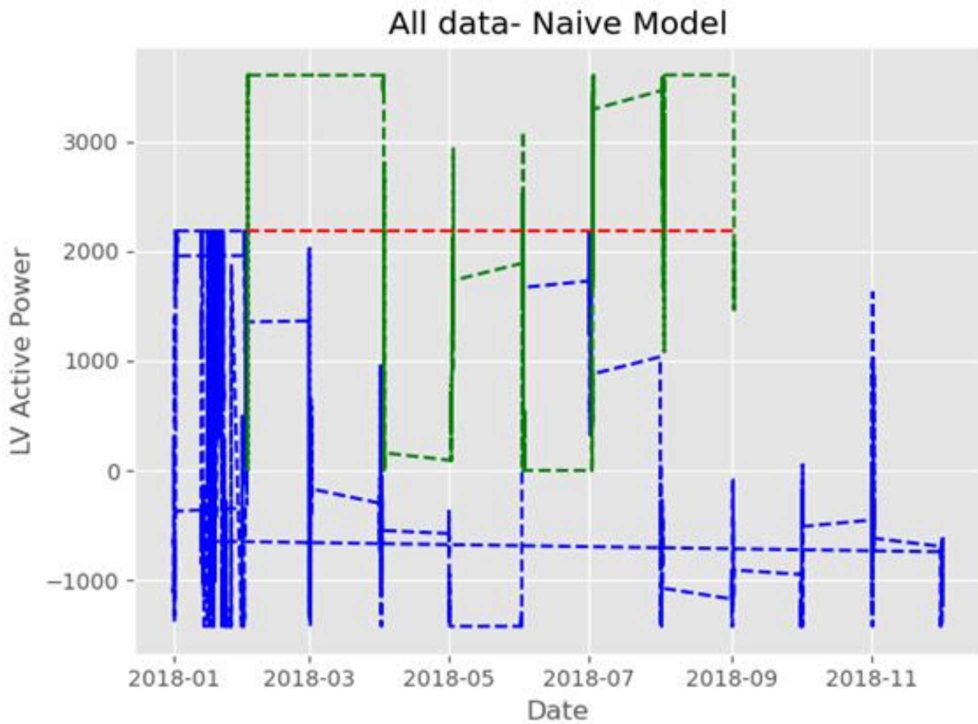
RMSE of Average Base Model: 1397.4434509009816



The ACF of the Average model does not resemble white noise, but it continues to decay with time.

Naive Model:

We find the h step prediction of the test data and find statistics based on the output.

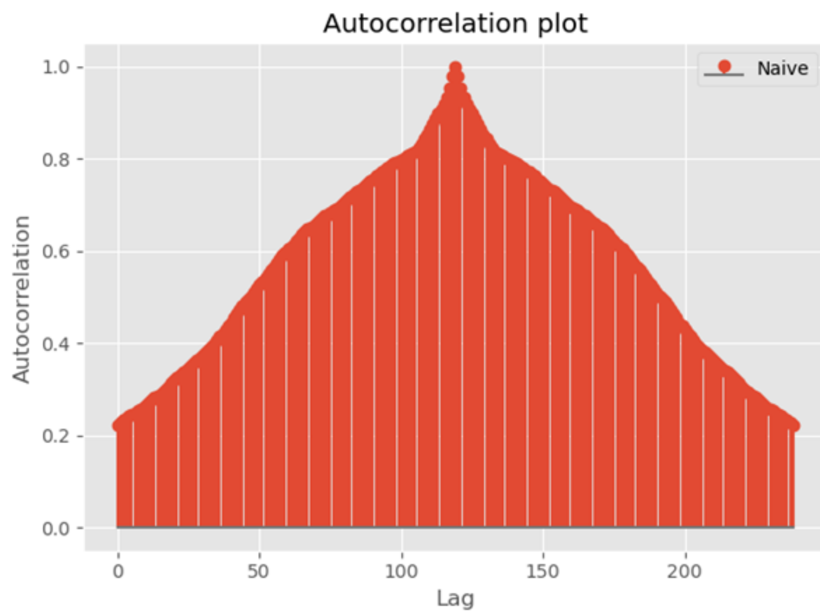


Mean of error of Naive Model: 94.30954345504651

Variance of error of Naive Model 1822037.7141250235

MSE of Naive Model: 1830932.0041117226

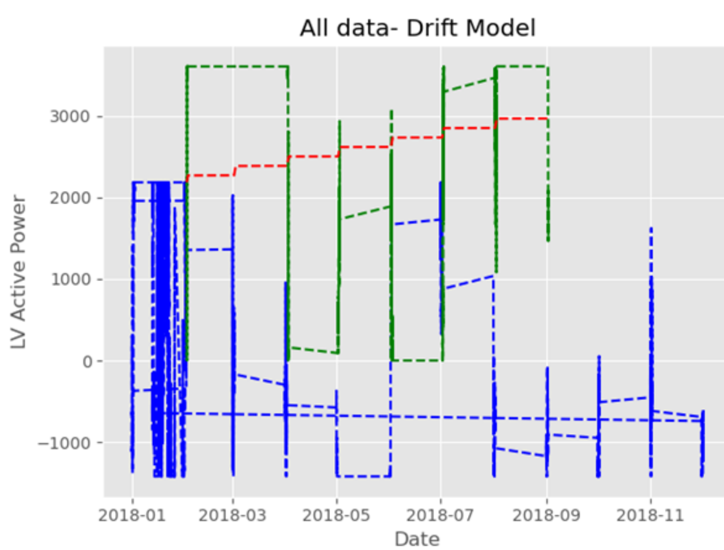
RMSE of Naive Model: 1353.1193606299935



The ACF of Naive model does not represent white noise but it decays with time.

Drift Model:

We find the h step prediction of the test data and find statistics based on the output.

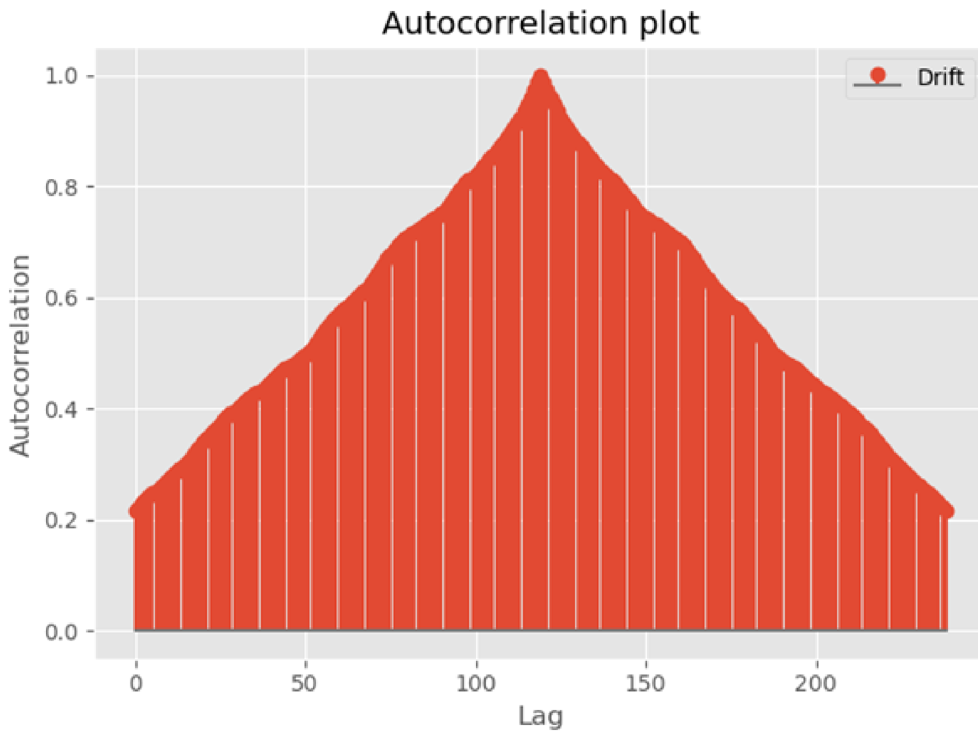


Mean of error of Drift Model: -2478.7502767393594

Variance of error of Drift Model 2010314.4396619329

MSE of Drift: 8154517.3740973845

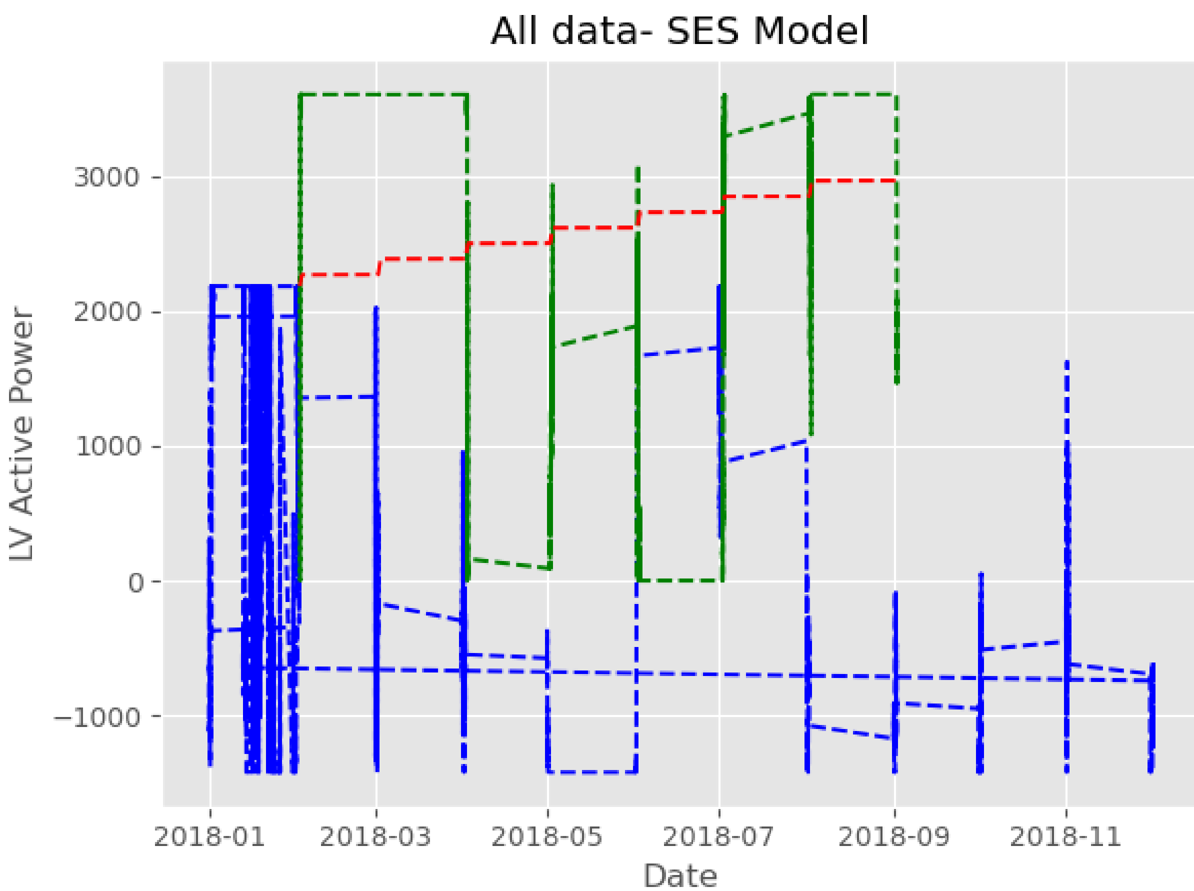
RMSE of Drift: 2855.6115586853516



The ACF of the Drift model does not represent white noise but continues to decay.

Simple Exponential Smoothing:

We find the h step prediction of the test data and find statistics based on the output.

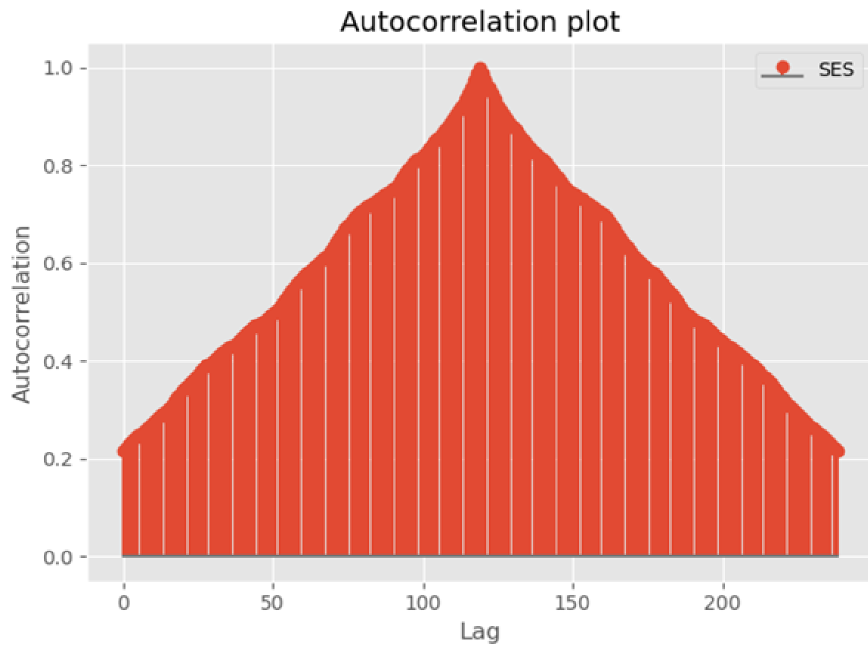


Mean of error of SES Model -2478.7502767393594

Variance of error of SES Model 2010314.4396619329

MSE of SES: 8154517.3740973845

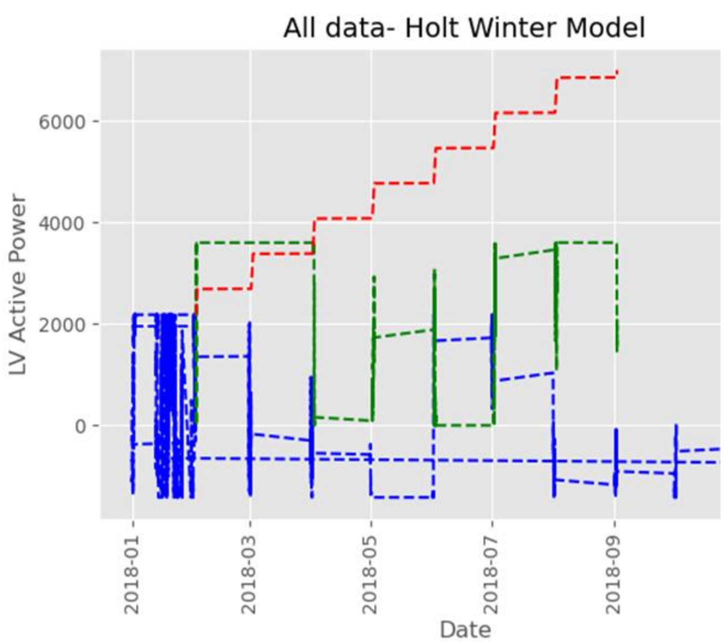
RMSE of SES: 2855.6115586853516



As we can see the ACF of the SES model does not represent white noise but it decays with time.

Holt Winter Model:

We find the h step prediction of the test data and find statistics based on the output.



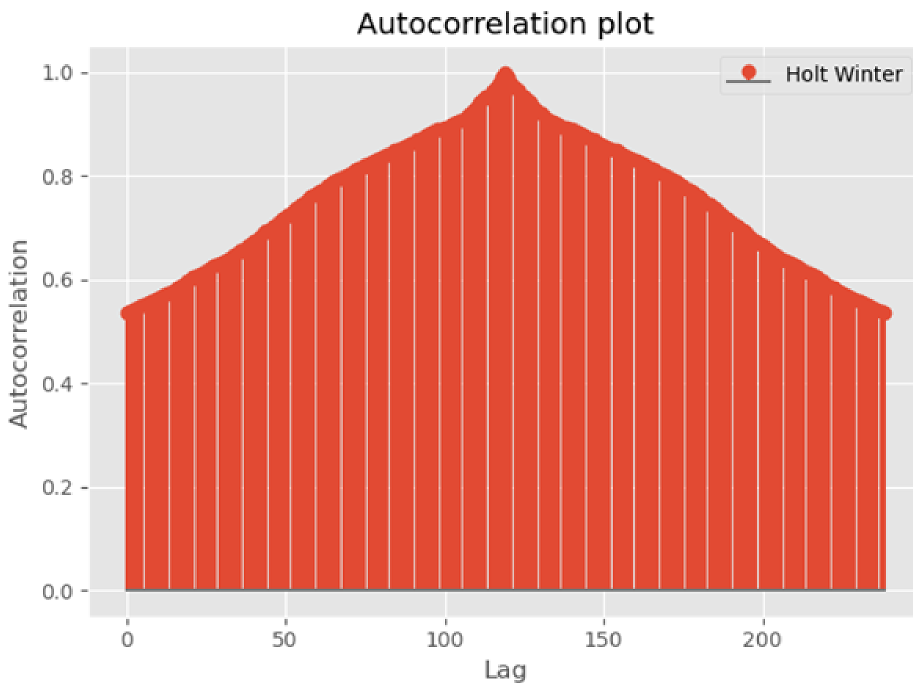
Mean of error of Holt Model   -2319.5252833418326

Variance of error of Holt Model 5145833.725079206

MSE of Holt Winter 10526031.265141152

RMSE of Holt Winter process 3244.384574174454

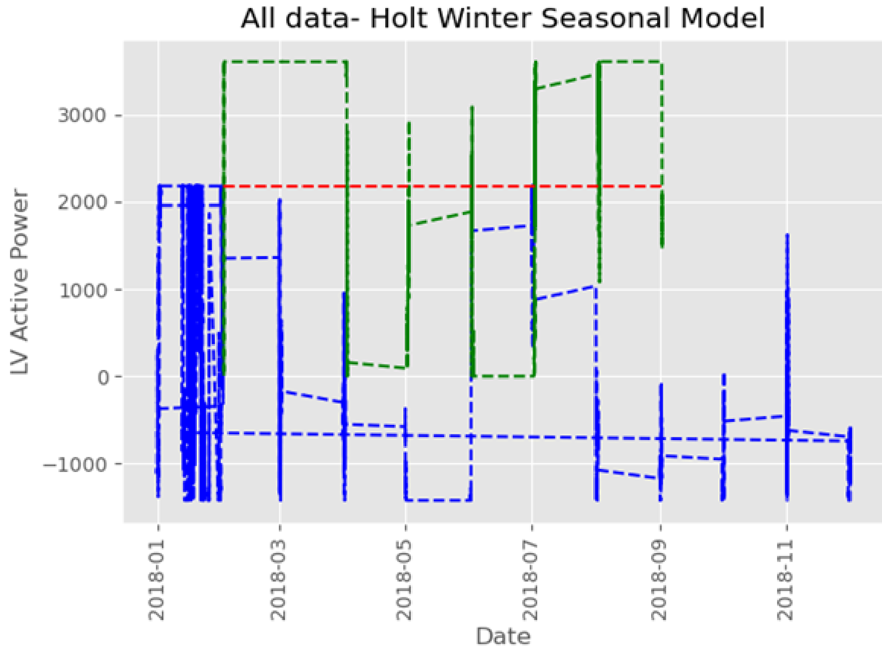
ACF of Holt Winter model:



The ACF decays with time but it does not represent white noise.

Holt Winter Seasonal Model:

We find the h step prediction of the test data and find statistics based on the output.



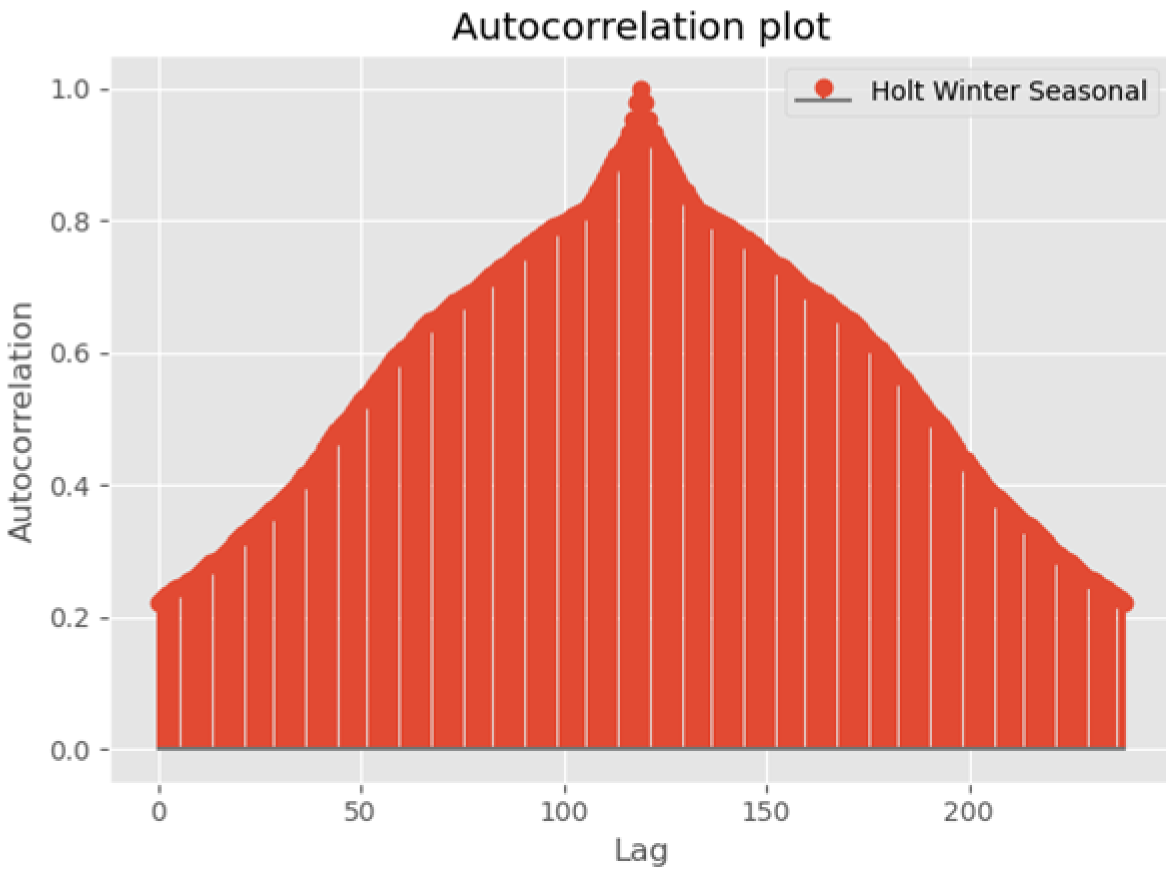
Mean of error of Holt Model 94.3108334020787

Variance of error of Holt Model 1822037.71412502

MSE of Holt Winter 1830932.2474220176

RMSE of Holt Winter process 1353.1194505371718

ACF of the Holt Winter Seasonal Model



The ACF decays with time but it does not represent white noise.

ARMA Model:

GPAC-

	1	2	3	4	5	6	7	8
0	1.00795	-0.04326	0.07896	0.00932	-0.04427	-0.05398	0.05627	0.00304
1	1.00864	1.79300	0.08403	0.38435	-0.05561	-0.09999	0.05917	1.14916
2	1.00743	-0.21353	0.61101	-0.10493	0.12147	-0.07194	-0.04107	-0.07924
3	1.00718	3.01255	0.84097	0.79546	-0.07643	-0.09139	0.19404	0.00304
4	1.00793	-1.23534	1.91341	1.48881	-1.62612	-0.04751	0.20223	17.47562
5	1.00885	0.96602	0.68764	-0.33283	-0.74415	-3.37094	0.02886	-0.13797
6	1.00798	-0.31366	0.98717	-2.21397	0.07359	-2.92524	-13.78495	-0.14967
7	1.00771	3.88874	1.29360	-2.39943	-96.63817	-2.97109	1.41549	-0.39858

We find four combinations for the GPAC-(1,2),(1,7),(3,2),(3,7). All four combinations pass the chi-sq test therefore we move forward with testing the RMSE values and as shown in the presentation the RMSE of the combination (1,7) is the lowest.

The ARMA process was done in three different ways. The first step we used data without differencing or transforming and used the inbuilt function to find the outputs.

Method 1:

```
ARMA model summary
                              ARMA Model Results
==============================================================================
Dep. Variable:     LV ActivePower (kW)   No. Observations:                 4000
Model:                      ARMA(1, 7)   Log Likelihood              -27818.311
Method:                        css-mle   S.D. of innovations            253.455
Date:                 Wed, 16 Dec 2020   AIC                          55656.622
Time:                         07:41:59   BIC                          55719.562
Sample:                              0   HQIC                         55678.933

==============================================================================================
                                 coef    std err          z      P>|z|      [0.025      0.975]
----------------------------------------------------------------------------------------------
const                          0.0001    336.904   3.22e-07      1.000    -660.320     660.320
ar.L1.LV ActivePower (kW)      0.9914      0.002    440.013      0.000       0.987       0.996
ma.L1.LV ActivePower (kW)      0.0070      0.016      0.432      0.666      -0.025       0.039
ma.L2.LV ActivePower (kW)     -0.0522      0.016     -3.189      0.001      -0.084      -0.020
ma.L3.LV ActivePower (kW)     -0.0533      0.016     -3.311      0.001      -0.085      -0.022
ma.L4.LV ActivePower (kW)     -0.0759      0.015     -5.029      0.000      -0.106      -0.046
ma.L5.LV ActivePower (kW)      0.0556      0.016      3.396      0.001       0.023       0.088
ma.L6.LV ActivePower (kW)     -0.0976      0.017     -5.913      0.000      -0.130      -0.065
ma.L7.LV ActivePower (kW)     -0.0378      0.016     -2.302      0.021      -0.070      -0.006
```

ARMA confidence interval

	0	1
const	-660.319837	660.320054
ar.L1.LV ActivePower (kW)	0.986954	0.995786
ma.L1.LV ActivePower (kW)	-0.024779	0.038773
ma.L2.LV ActivePower (kW)	-0.084348	-0.020128
ma.L3.LV ActivePower (kW)	-0.084853	-0.021754
ma.L4.LV ActivePower (kW)	-0.105543	-0.046347
ma.L5.LV ActivePower (kW)	0.023491	0.087610
ma.L6.LV ActivePower (kW)	-0.130012	-0.065280
ma.L7.LV ActivePower (kW)	-0.070009	-0.005619

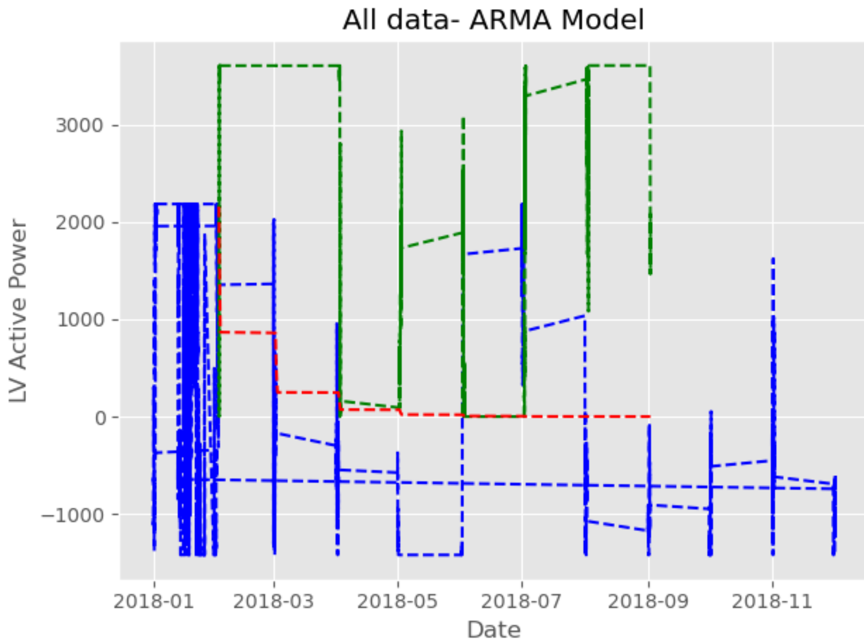
As you can see the confidence intervals do not include a zero, therefore there no simplification.

Roots

	Real	Imaginary	Modulus	Frequency
AR.1	1.0087	+0.0000j	1.0087	0.0000
MA.1	1.3218	-0.0000j	1.3218	-0.0000
MA.2	0.8666	-1.2250j	1.5006	-0.1520
MA.3	0.8666	+1.2250j	1.5006	0.1520
MA.4	-0.5079	-1.2981j	1.3939	-0.3094
MA.5	-0.5079	+1.2981j	1.3939	0.3094
MA.6	-1.4353	-0.0000j	1.4353	-0.5000
MA.7	-3.1862	-0.0000j	3.1862	-0.5000

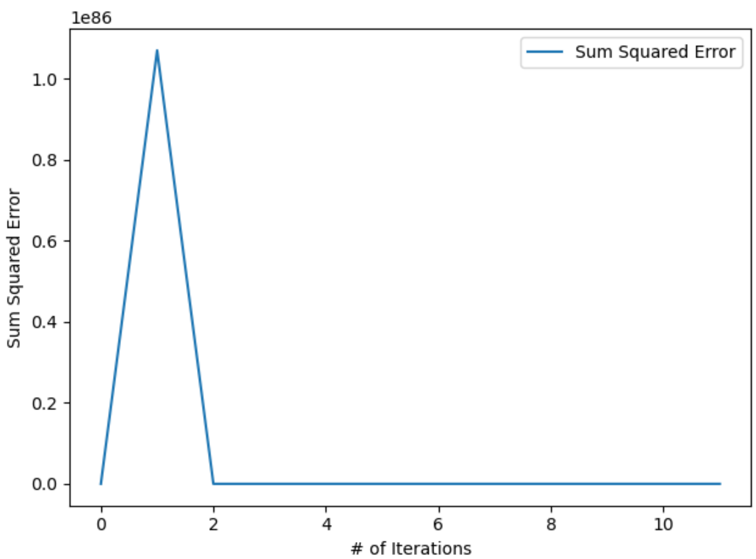
On looking at the roots we see that none of the roots are the same, therefore no simplification is needed.

Forecast:



Method 2:

We use LM algorithm to find parameters

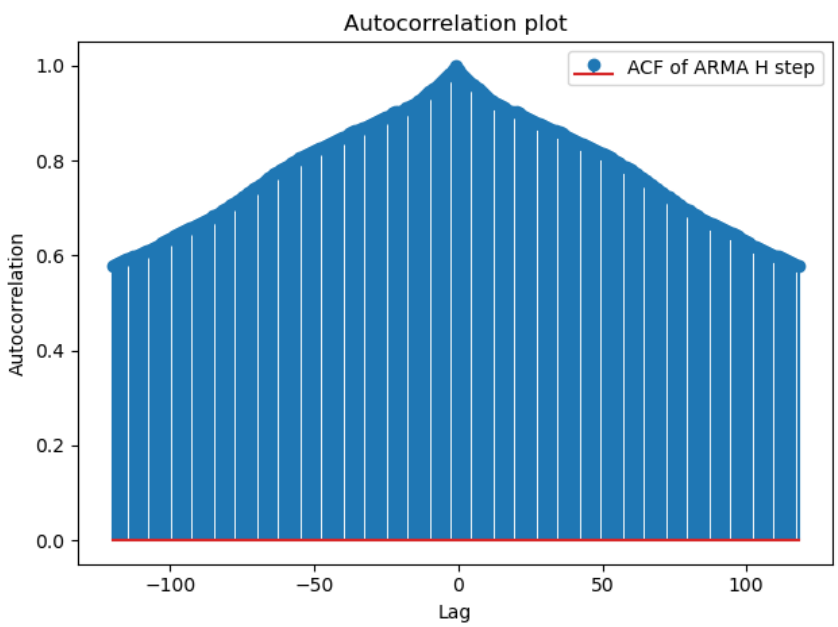


We see that the code converges at the 10th iteration.

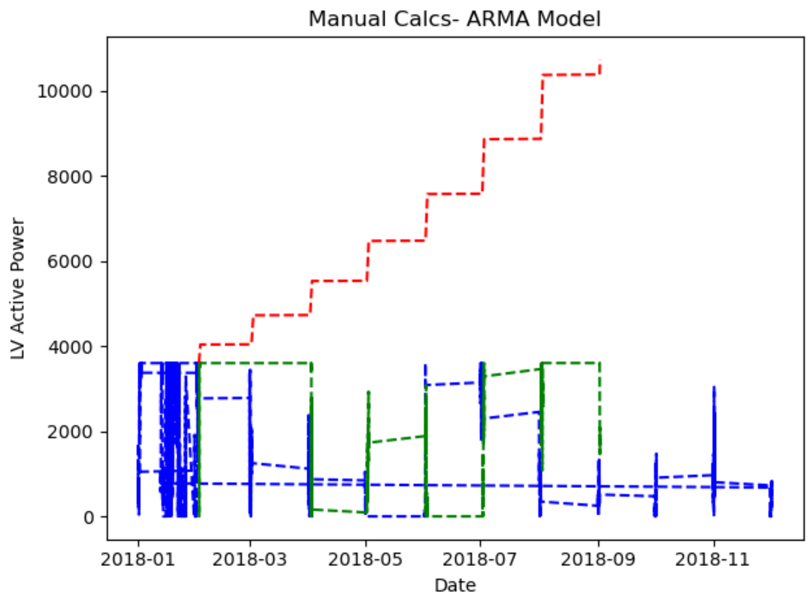
Below we see the estimated parameter list and the covariance matrix of the of the LM algorithm:

```
Estimated parameters :  [-0.99618301  0.00491499 -0.05504857 -0.05561321 -0.07856018  0.05308865
 -0.10017572 -0.04026626]
Estimated Covariance matrix :  [[ 2.30120360e-06  2.38812658e-06  2.61097062e-06  2.35737179e-06
   2.39170988e-06  2.32456142e-06  2.55664648e-06  2.30484725e-06]
 [ 2.38812658e-06  2.52584429e-04  2.84475775e-06 -1.06384079e-05
  -1.23189337e-05 -1.77649608e-05  1.54025034e-05 -2.27956525e-05]
 [ 2.61097062e-06  2.84475775e-06  2.50532520e-04  4.08697808e-06
  -1.24103122e-05 -1.36643645e-05 -1.85866837e-05  1.53651072e-05]
 [ 2.35737179e-06 -1.06384079e-05  4.08697808e-06  2.50019984e-04
   5.66568351e-06 -1.09324438e-05 -1.36822187e-05 -1.78152730e-05]
 [ 2.39170988e-06 -1.23189337e-05 -1.24103122e-05  5.66568351e-06
   2.49339598e-04  5.63226946e-06 -1.24659674e-05 -1.24038627e-05]
 [ 2.32456142e-06 -1.77649608e-05 -1.36643645e-05 -1.09324438e-05
   5.63226946e-06  2.49954485e-04  3.99664222e-06 -1.07543383e-05]
 [ 2.55664648e-06  1.54025034e-05 -1.85866837e-05 -1.36822187e-05
  -1.24659674e-05  3.99664222e-06  2.50412830e-04  2.69839135e-06]
 [ 2.30484725e-06 -2.27956525e-05  1.53651072e-05 -1.78152730e-05
  -1.24038627e-05 -1.07543383e-05  2.69839135e-06  2.52417826e-04]]
Estimated variance of error :  64543.984584077916
```

Then we hard code one step and h step predictions.



We can see that the ACF of the hard coded ARMA completely decays in 120 lags.



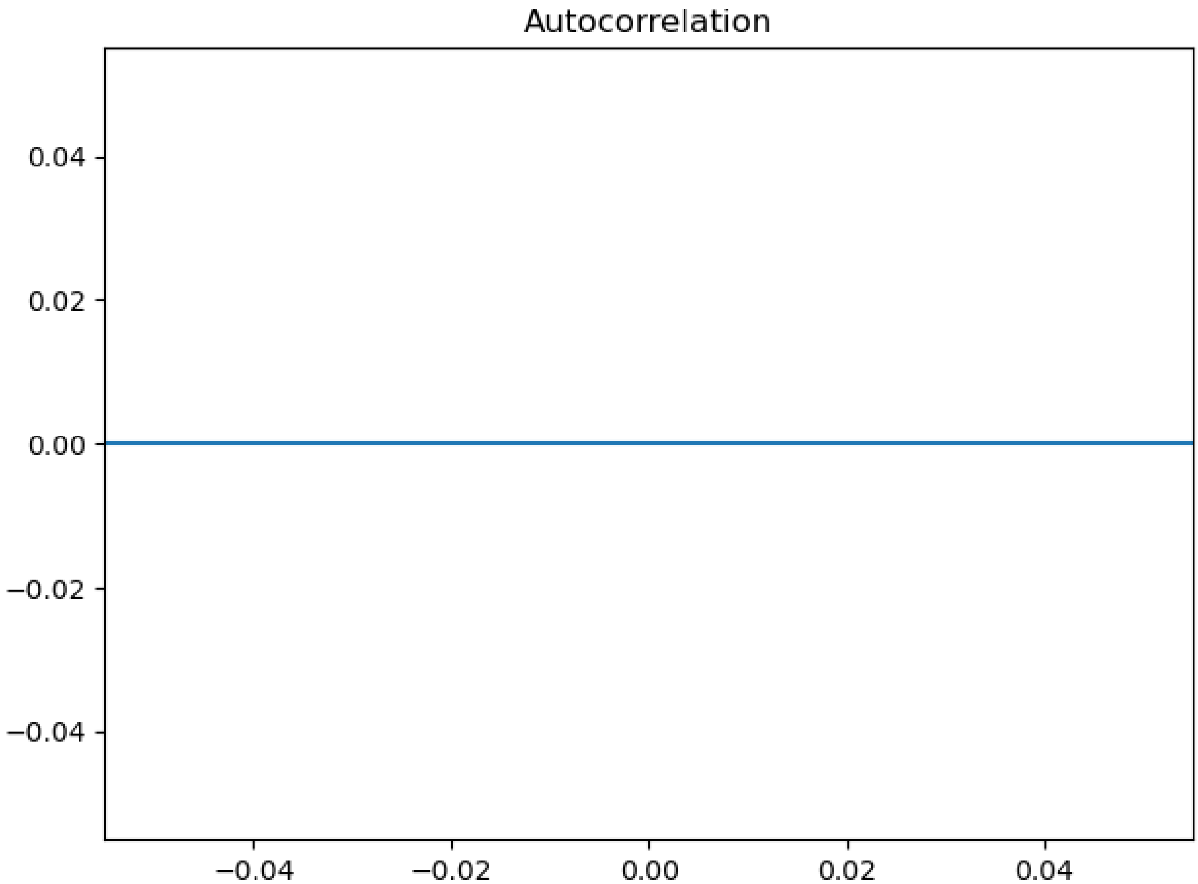
MSE: 25699009.624391627
RMSE 5069.4190618247
Q value 146.4074314568248

Method 3 - Lastly we use differenced data:

After differencing the data, the ACF and GPAC break. I believed that there may have been a fault in my code so I attempted to use the in built function for ACF and I still got Nan outputs.

ACF values:

data_acf
[nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan]



GPAC Values:

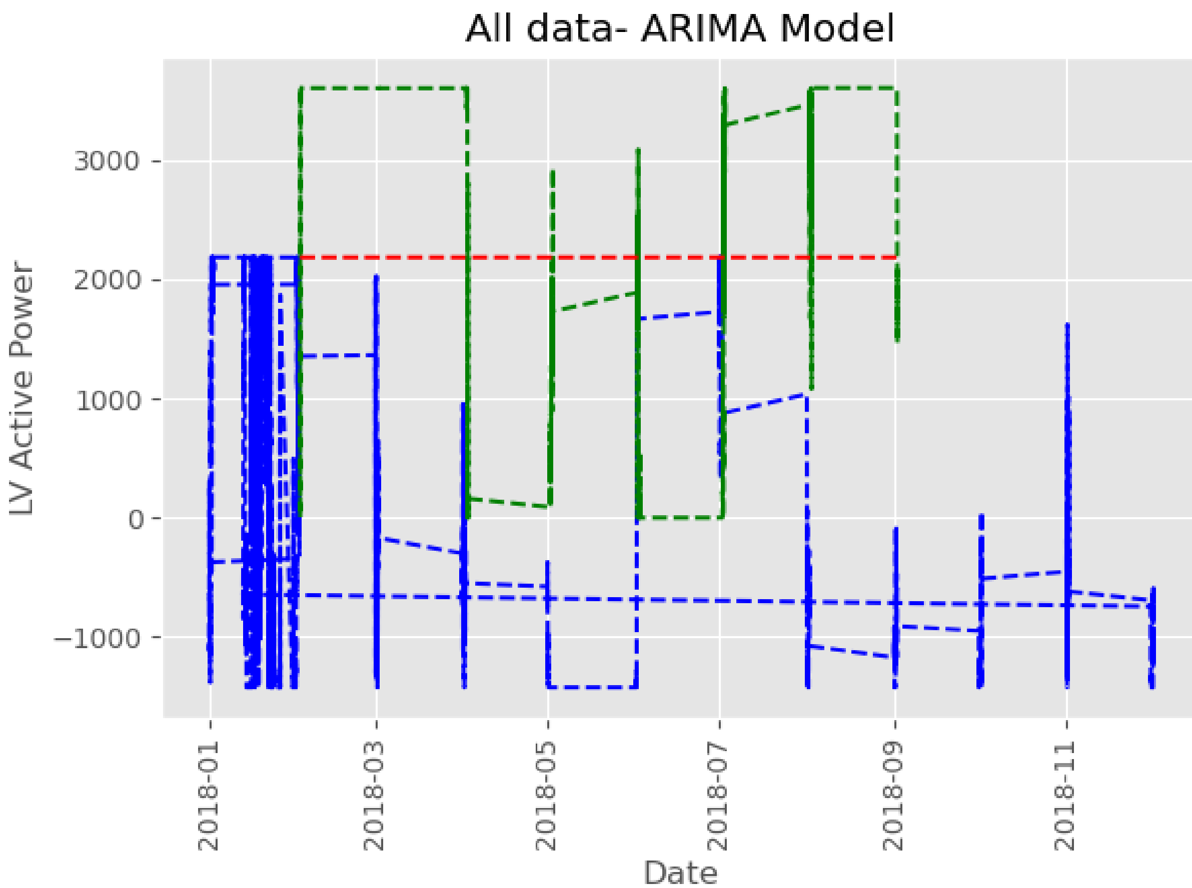
	1	2	3	4	5	6	7	8
0	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
1	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
3	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
4	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
5	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
6	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
7	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

I tried to find out how to fix this but I was unable to do so. I believe that since the data is recorded over a period of ten minute intervals after we do first difference the data distribution changes.

ARIMA:

We find the h step prediction of the test data and find statistics based on the output.

For ARIMA model I used 1 as the order for the AR, 7 as the order for MA and 1 for the value of D to perform differencing.



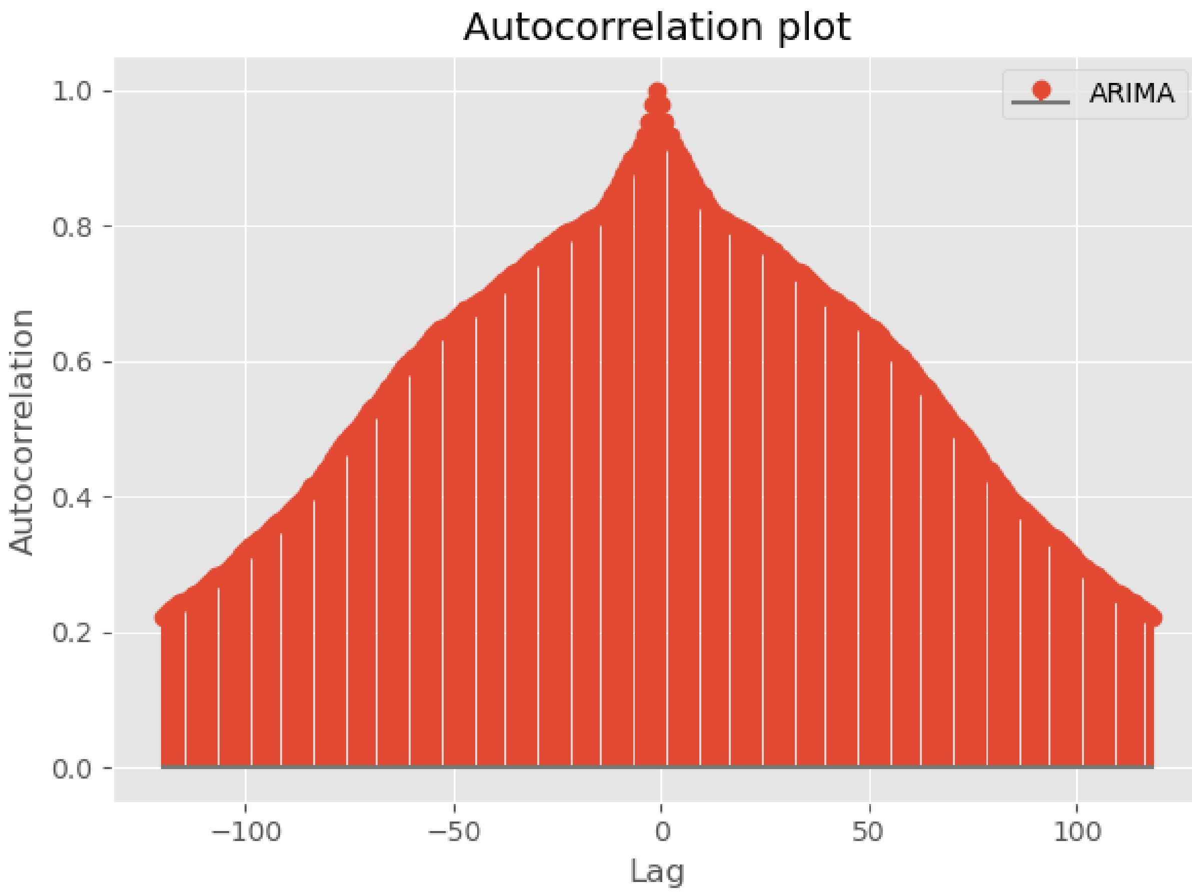
Mean of error of ARIMA Model -409.3197895698537

Variance of error of ARIMA Model 2192109.596496847

MSE of ARIMA process 2359652.2866303558

RMSE of ARIMA process 1536.115974342548

I was hoping that introducing a differencing variable would reduce the error or improve the accuracy of the model.



Model selected:

Q value Multiple Linear Regression 40.16520043050357

Q value Average Model 81.99712787277191

Q value Naive Model 88.92135966850353

Q value Drift Model 89.9517026952922

Q value SES Model 89.9517026952922

Q value Holt Model 138.8496438805654

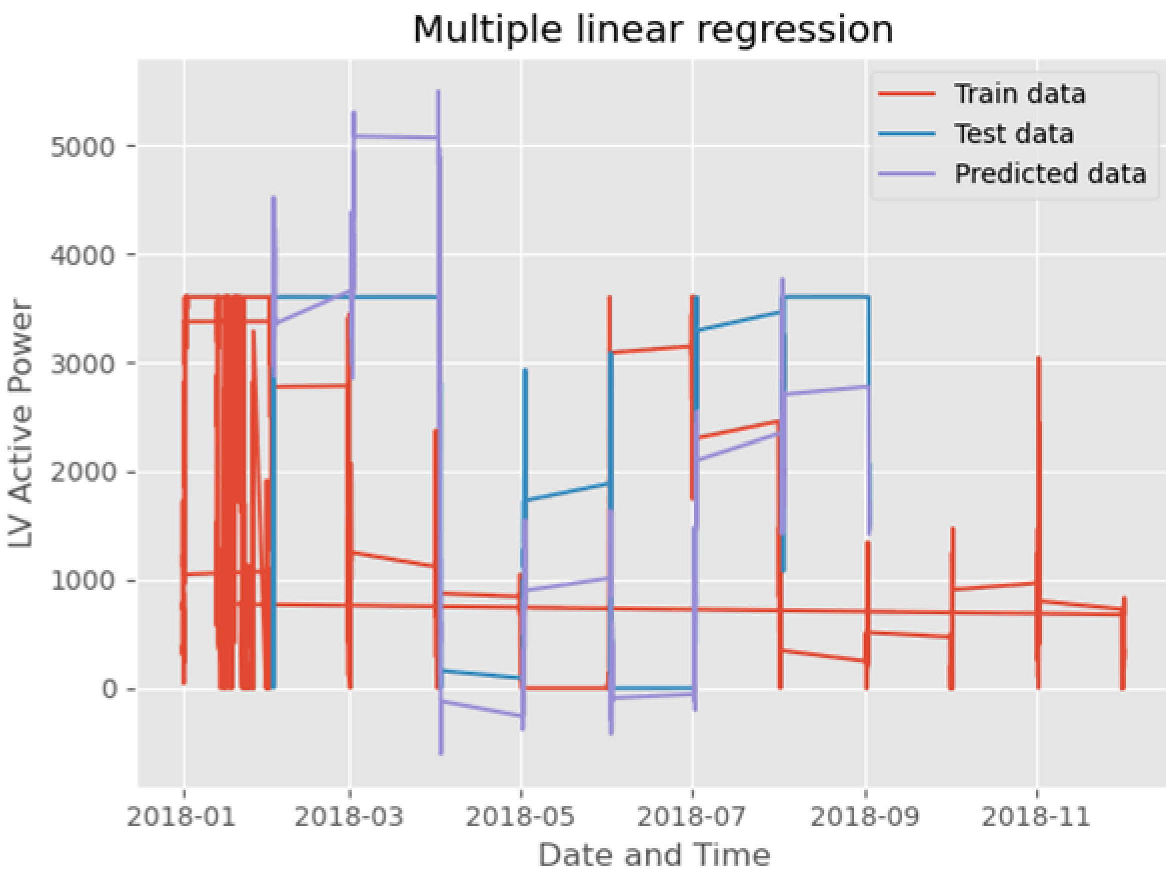
Q value Holt Seasonal Model 88.92135966850353

Q value ARMA Model 75.80093080365906

Q value ARIMA Model 88.92136072962846

On looking at the Q values I selected Multiple Linear to be the final model to perform forecasting. This means that data can only accurately be predicted when the data has other features to support it, if the model is trained with additional features then the model performs better.

Multiple linear regression



Conclusion:

Thus, we coded different time series models and we found that Multiple Linear Regression model was the best, this decision was made based on Q values. In order to improve the model, I believe we are required to perform a little more pre-processing. I would attempt to reduce the data by taking the mean value of the day so that we have one value for each day instead of every ten minutes.

Appendix:

code_collection.py- Toolbox for codes.
FINAL_PROJ.py- Codes for the term project
LM_tester.py- Implementation of Levenberg Marquardt Algorithm
term_proj_arma.py- ARMA code using LM parameters
term_proj_ARMA_diff.py- ARMA code after performing first differencing

code_collection.py-
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from numpy import linalg
from scipy import signal
#mean of values for lab1
def calc_mean(data):
    sum_sales = 0
    m_sales = []
    v_sales = []
    temp = pd.Series()
    n = 1
    for s in data['Sales']:
        sum_sales = sum_sales + s
        mean = sum_sales / n
        temp = data['Sales'][:n]
        variance = temp.var()
        m_sales.append(mean)
        v_sales.append(variance)
        n += 1
    sum_gdp = 0
    m_gdp = []
    v_gdp = []
    temp = pd.Series()
    n = 1
    for g in data['GDP']:
        sum_gdp = sum_gdp + g
        mean = sum_gdp / n
        temp = data['GDP'][:n]
        variance = temp.var()
        m_gdp.append(mean)
        v_gdp.append(variance)
        n += 1
    sum_adb = 0
    m_adb = []
```

```python
    v_adb = []
    temp = pd.Series()
    n = 1
    for a in data['AdBudget']:
        sum_adb = sum_adb + a
        mean = sum_adb / n
        temp = data['AdBudget'][:n]
        variance = temp.var()
        m_adb.append(mean)
        v_adb.append(variance)
        n += 1
    return m_sales,v_sales, m_gdp, v_gdp, m_adb, v_adb
#correlation coefficient
def correlation_coefficent_cal(x,y):
    mean1=np.mean(x)
    mean2=np.mean(y)
    sum1=np.sum((x-mean1)*(y-mean2))
    sum2=np.sqrt(np.sum((x-mean1)**2))
    sum3=np.sqrt(np.sum((y-mean2)**2))
    val2=sum2*sum3
    r =sum1/val2
    return r
# ACF calculations
def calc_tow(Y,tow):
    l=len(Y)
    num=0
    den=0
    y_mean=np.mean(Y)
    for s in Y:
        den=den+((s-y_mean)**2)
    for j in range(tow,l):
        v1=Y[j] - y_mean
        v2=Y[j-tow] - y_mean
        num=num+(v1*v2)
    val=num/den
    return val
def calc_acf(data,lags):
    val = []
    for i in range(0, lags):
        val.append(calc_tow(data, i))
    val_w = val[1:]
    r_val = val_w[::-1]
    acf_val= r_val + val
    return acf_val
def plt_acf(acf_set,lb,lags):
    plt.stem(np.arange(-lags,lags-1),acf_set, label=lb)
    plt.title("Autocorrelation plot")
    plt.xlabel('Lag')
    plt.ylabel('Autocorrelation')
    plt.legend()
    plt.show()


# --------------------------------
#  Generate y - auxiliary functions
# --------------------------------
# Input coefficients based on order
```

```
def process_coef(order, process):
    coefficients = []
    for i in range(1, order + 1):
        coef = float(input("Enter coefficient #{} of {} process (ex: -.5 or
.8) = ".format(i, process)))
        coefficients.append(coef)
    return coefficients


# den and num should have the same size
# function returns [1, a1, a2,...]
def prepare_dlsim_coef(a, b):
    # a = den = ar_coef = y
    # b = num = ma_coef = e
    # Evaluate size
    if type(a) != list or type(b) != list:
        a = a.tolist()
        b = b.tolist()
    if len(a) > len(b):
        while len(a) > len(b):
            b.append(0.0)
    elif len(b) > len(a):
        while len(b) > len(a):
            a.append(0.0)
    # add 1
    den_dlsim = [1]
    den_dlsim.extend(a)
    num_dlsim = [1]
    num_dlsim.extend(b)
    return den_dlsim, num_dlsim


# generate y with dlsim
def dlsim_generate_y(e, den, num):
    den = den  # na
    num = num  # nb
    sys = (num, den, 1)
    tout, y = signal.dlsim(sys, e)
    return y.flatten()


# convert y (in case mean!=0 and var!=1)
def convert_y(y, mean, var, ar_coef, ma_coef):
    if mean != 0 and var != 1:
        mean_y = (mean * (1 + np.sum(ma_coef))) / (1 + np.sum(ar_coef))
        y_final = y - mean_y
        print("Mean is different from 0 and Var is different from 1. Data was
converted.")
    else:
        y_final = y
    return y_final


# ---------------------------------
#   Generate y - MAIN fuction
# ---------------------------------
# input variables
```

```
def generate_y():
    # ---------- input variables -------------
    T = int(input("Enter number of data samples = "))
    mean = int(input("Enter the mean of white noise = "))
    var = int(input("Enter the variance of white noise = "))
    std = np.sqrt(var)
    na = int(input("Enter the AR order = "))
    process = "AR"
    ar_coef = process_coef(na, process)
    nb = 1
    nb = int(input("Enter the MA order = "))
    process = "MA"
    ma_coef = process_coef(nb, process)
    print("AR coefficients = ", ar_coef)
    print("Ma coefficients = ", ma_coef)

    # ---------- Generate y -------------
    # prepare dlsim coefficients
    ar_coef_dlsim, ma_coef_dlsim = prepare_dlsim_coef(a=ar_coef, b=ma_coef)
    # Define e
    e_orig = np.random.normal(mean, std, size=T)
    # Generate y
    y_orig = dlsim_generate_y(e=e_orig, den=ar_coef_dlsim, num=ma_coef_dlsim)
    # Convert y
    y = convert_y(y=y_orig, mean=mean, var=var, ar_coef=ar_coef,
ma_coef=ma_coef)

    return na, nb, ar_coef, ma_coef, y


# def data_generation():
# input variables and generate y
#    na,nb,ar_coef,ma_coef,y = generate_y()
#    return na,nb,ar_coef,ma_coef,y

# --------------------------------
#     LM - auxiliary functions
# --------------------------------
# hyperparameters
iterations = 10
delta = pow(10, -6)
miu = 0.01
miu_max = pow(10, 10)
max_iterations = 50


# Calculate e
def cal_e(na, nb, y, theta):
    den_orig = theta[:na]  # ar
    num_orig = theta[na:]  # ma
    # prepare dlsim coefficients
    den_dlsim, num_dlsim = prepare_dlsim_coef(a=den_orig, b=num_orig)
    # generate e with dlsim
    den = den_dlsim  # na
    num = num_dlsim  # nb
    sys = (den, num, 1)
    tout, e = signal.dlsim(sys, y)
```

```python
    return e.flatten()


# Calculate SSE
def cal_SSE(e):
    # e transposed
    e_transposed = e.T
    # calculate SSE
    SSE = e.dot(e_transposed)
    return SSE


# calculate negative gradient
def cal_gradient(na, nb, y, e, delta, theta):
    num_parameters = na + nb
    x_matrix_values = []
    x = np.zeros(len(e))
    for i in range(0, num_parameters):
        theta[i] += delta
        e_new = cal_e(na, nb, y, theta)
        x = (e - e_new) / delta
        x_matrix_values.append(x)
        # subtract delta
        theta[i] -= delta
    X = np.stack(x_matrix_values, axis=1)
    # Calculate A
    A = X.T.dot(X)
    # Calculate g
    g = X.T.dot(e)
    return A, g


# Calculate theta change
def cal_delta_theta(na, nb, miu, A, g):
    num_parameters = na + nb
    theta_change = g.dot(np.linalg.inv((np.identity(num_parameters) * miu) +
A))
    return theta_change


# ------------ step 1 ------------
def step1(na, nb, y, delta, theta):
    # calculate e
    e = cal_e(na, nb, y, theta)
    # calculate SSE
    SSE_old = cal_SSE(e=e)
    # calculate negative gradient
    A, g = cal_gradient(na, nb, y, e, delta, theta)
    return SSE_old, A, g


# ------------ step 2 ------------
def step2(na, nb, y, miu, A, g, theta):
    # calcultae delya change
    delta_theta = cal_delta_theta(na, nb, miu, A, g)
    # add cange to theta
    theta_new = theta + delta_theta
```

```
    # calculate new e based on change
    e_new = cal_e(na, nb, y, theta=theta_new)
    # calcultae new SSE based on change
    SSE_new = cal_SSE(e=e_new)
    return SSE_new, theta_new, delta_theta


# calculate confidence intervals
def cal_conf_interval(theta_hat, cov_theta_hat):
    coef = theta_hat
    a = coef - (2 * (np.sqrt(cov_theta_hat)))
    b = coef + (2 * (np.sqrt(cov_theta_hat)))
    conf = np.concatenate((a, b), axis=None)
    return conf


# ---------------------------------
#       LM - MAIN function
# ---------------------------------
def LM_algorithm():
    na, nb, ar_coef, ma_coef, y = generate_y()
    y = np.array(y)

    # hyperparameters
    # iterations = 50
    delta = pow(10, -6)
    miu = 0.01
    miu_max = pow(10, 10)
    max_iterations = 50

    SSE_list = []

    # initializse thetas to zero
    theta_init = [0.0 for a in range(1, na + 1)] + [0.0 for b in range(1, nb
+ 1)]
    SSE_old, A, g = step1(na, nb, y, delta, theta=theta_init)
    SSE_new, theta_new, delta_theta = step2(na, nb, y, miu, A, g,
theta=theta_init)
    SSE_list.append(SSE_new)

    iterations = 1
    # if iterations < max_iterations:
    for i in range(1, max_iterations):

        if np.isnan(SSE_new):
            SSE_new = np.exp(10)
        else:
            if SSE_new < SSE_old:
                if linalg.norm(np.array(delta_theta), ord=2) < pow(10, -3):
                    # print("***************Algorithm converges
******************")
                    # algorithm converges because there is no significant
contribution of delta theta
                    theta_hat = theta_new
                    # variance of error
                    var_error = SSE_new / (len(y) - (na + nb))
                    # cov_theta
```

```
                    cov_theta_hat = var_error * linalg.inv(A)   # for conf
intervals
                    break
                else:
                    theta_old = theta_new
                    miu = miu / 10  # decrease miu

            while SSE_new > SSE_old:
                miu = miu * 10  # increase miu
                # print("increase miu", miu)
                if miu > miu_max:
                    print("ERROR: miu exceeds maximum.")
                    break
                # change miu
                theta_old = theta_new
                SSE_new, theta_new, delta_theta = step2(na, nb, y, miu, A, g,
theta=theta_old)

            # theta = theta_new
            SSE_old, A, g = step1(na, nb, y, delta, theta=theta_old)
            SSE_new, theta_new, delta_theta = step2(na, nb, y, miu, A, g,
theta=theta_old)
            SSE_list.append(SSE_new)
            iterations += 1

    plt.figure()
    plt.plot(range(0, iterations), np.array(SSE_list))
    plt.title("SSE vs number of iterations")
    plt.show()

    # elif iterations > max_iterations:
    #    print("ERROR: iterations exceed maximum.")
    print("True parameters are = ", ar_coef, ma_coef)
    print("Estimated parameters are = ", theta_hat)
    print("Variance is = ", var_error)
    print("Covariance Matrix is = ", cov_theta_hat)
    # print("iterations = " ,iterations)

    # calculate confidence intervals
    conf_intervals = []
    for i in range(len(theta_hat)):
        intervals = cal_conf_interval(theta_hat[i], cov_theta_hat[i][i])
        conf_intervals.append(intervals)
    # print confidence intervals of a coefficients
    for i in range(len(conf_intervals[:na])):
        print(conf_intervals[:na][i][0], " < a{} < ".format(i + 1),
conf_intervals[:na][i][1])
    # print confidence intervals of b coefficients
    for i in range(len(conf_intervals[na:])):
        print(conf_intervals[na:][i][0], " < b{} < ".format(i + 1),
conf_intervals[na:][i][1])

    return theta_hat, var_error, cov_theta_hat, conf_intervals, y, na, nb,
ar_coef, ma_coef


# ////////////////////////////////////////////// #
```

```python
# ----- Autocorrelation Function ----------- #
# ////////////////////////////////////////// #
def ACF_cal(y, lags, title):
    # convert to array
    array = np.array(y)
    # get mean
    mean = sum(y) / len(y)
    # length
    samples = len(y)
    # subtract mean to each value
    x = array - mean
    # Calculate Denominator
    denominator = x.dot(x)
    # Calculate Taus
    tau = np.zeros(lags + 1)
    tau[0] = 1  # The first tau is 1
    # Create a loop that iterates all lags (except lag 0)
    for i in range(lags):
        tau[i + 1] = x[i + 1:].dot(x[:-(i + 1)])
    # divide by denominator
    tau[1:lags + 1] = tau[1:lags + 1] / denominator

    # plot
    coordinates = []
    for i in range(len(tau)):
        coordinates.append((i, tau[i]))
        coordinates.append((i * -1, tau[i]))

    axis_x = []
    axis_y = []
    for pair in coordinates:
        axis_x.append(pair[0])
        axis_y.append(pair[1])

    # plot
    plt.figure(figsize=(8, 5))
    plt.title("Autocorrelation of {} (lags={} and samples={})".format(title,
lags, samples),
              fontsize=14)
    plt.xlabel('Lags', fontsize=12)
    plt.ylabel('Magnitude', fontsize=12)
    plt.stem(axis_x, axis_y)
    plt.show()

    r = tau.copy()

    return r

def GPAC_matrix(R,j,k):
    num=np.zeros((k,k))
    den=np.zeros((k,k))
    for a1 in range(0,k):
        for a2 in range(0,k):
            #diagonal elements
            if a1==a2:
                num[a1][a2]=R[j]
                den[a1][a2] = R[j]
```

```
            #last diagonal element
            if a1==k-1 and a2==k-1:
                num[a1][a2] = R[j+k]
                den[a1][a2] = R[j]
            #a1=0 a2=1
            if a1>a2:
                num[a1][a2] = R[j + a1]
                den[a1][a2] = R[j + a1]
            # a2=0 a1=1
            if a2 > a1:
                num[a1][a2] = R[j - a2]
                den[a1][a2] = R[j - a2]
            #last row
            if a1==k-1:
                num[a1][a2] = R[j +k - a2-1]
                den[a1][a2] = R[j + k - a2 - 1]
            #last column
            if a2==k-1:
                num[a1][a2] = R[j+ a1 +1]
                den[a1][a2] = R[j -k+a1+1]
    final=np.linalg.det(num)/np.linalg.det(den)
    return final
def GPAC(r,j,k):
    arr = np.zeros((j, k))
    for a1 in range(0,j):
        for a2 in range(0,k):
            arr[a1][a2]=GPAC_matrix(r, a1, a2)
    return arr
def adj_seasonality(data,y):
    y_hat=[]
    for i, j in zip(data,y):
        y_hat.append(i-j)
    return y_hat


# for adding zeros to get the strength of seasonality addition
def modify(mva,fold):
    l=[0]*fold
    b=[0]*fold
    final=l+mva+b
    return final



def test_input(m,k):
    if (m>=3):
        if (m%2!=0):
            if (k>0 and k%2==0):
                print("Wrong input, both should be odd values.")
                return False
        elif(m%2==0):
            if (k%2!=0):
                print("Wrong Input, both should be even values.")
                return False
    else:
        print("Input moving average value greater than 2.")
        return False
```

```
def adf_test(data):
    X = data.values
    result = adfuller(X)
    print(result[0])
#function for folding
def folding(data,fold):
    k=int((fold-1)/2)
    mva = []
    test = len(data) - 2 * (k+1)
    for i in range(0, len(data)):
        if (k <= test):
            temp = i + fold
            val = sum(data[i:temp]) / fold
            mva.append(val)
            k = k + 1
    return mva


#intitial function
def mv_avg(m,fold,data):
    k=int((m-1)/2)
    mva = []
    test=len(data)-2*k
    for i in range(0,len(data)):
        if (k <=test):
            temp=i+m
            val=sum(data[i:temp])/m
            mva.append(val)
            k=k+1
    if (m%2==0):
        final=folding(mva,fold)
        return final
    else:
        return mva
def q_val(train_set, acf):
    T = len(train_set)
    sum= 0
    for val in acf[1:]:
        sum += (val**2)
    Q = T*sum
    return sum
#calculation avg
def calc_avg(train,test):
    final_avg=[]
    error=[]
    error2=[]
    avg=np.mean(train)
    for i in range(0,len(test)):
        final_avg.append(avg)
    for t,f in zip(train,final_avg):
        error.append(t-f)
    for e in error:
        error2.append(e**2)
    mse=np.mean(error2)
    return final_avg,error,mse
#calculating naive
def calc_naive(train,test):
```

```
    train=list(train)
    test=list(test)
    j=0
    naive = []
    error = []
    error2 = []
    for t in range(0,len(test)):
        naive.append(train[-1])
        j+=1
    for t1,t2 in zip(test,naive):
        error.append(t1-t2)
    for e in error:
        error2.append(e**2)
    mse=np.mean(error2)
    return naive,error,mse
#calculations for drift
def calc_drift(train,test):
    train=list(train)
    test=list(test)
    yt=train[-1]
    y1=train[0]
    T=len(train)
    error=[]
    error2=[]
    final_val=[]
    for j in range(1,len(test)):
        num=j*((yt-y1)/(T-1))
        final=yt+num
        final_val.append(final)
    for t,f in zip(train,final_val):
        error.append(t-f)
        error2.append((t-f)**2)
    mse = np.mean(error2)
    return final_val,error,mse
#caculations for ses
def calc_ses(alfa,train,test):
    ses=[]
    error=[]
    error2=[]
    ses.append(train[-1])
    for j in range(1,len(test)):
        ses.append(train[-1])
    for i, j in zip(test,ses):
        error.append(i-j)
        error2.append((i-j)**2)
    mse = np.mean(error2)
    return ses,error,mse
```

FINAL_PROJ.py-

```
import numpy as np
import pandas as pd
```

```
import warnings
import scipy
import code_collection
import statsmodels.api as sm
from matplotlib import style
import seaborn as sns
from scipy.stats import chisquare
import statsmodels.tsa.holtwinters as ets
from statsmodels.tsa.seasonal import seasonal_decompose
import matplotlib.pyplot as plt
from statsmodels.tsa.stattools import adfuller
from sklearn.model_selection import train_test_split
warnings.filterwarnings("ignore")
style.use('ggplot')
np.set_printoptions(suppress=True)
data=pd.read_csv("T1.csv",header=0)
#selecting only the first 5000 data points
data=data[:5000]
date_check=data['Date/Time']
print("Start date:",date_check[0])
print("End date:",date_check.iloc[-1])
#data['LV ActivePower (kW)']=data['LV ActivePower (kW)']-np.mean('LV
ActivePower (kW)')
#data=data-np.mean(data)
data1=data[:144]
#EDA-data conversion
data['Date/Time']=pd.to_datetime(data['Date/Time'])
print(data.head(0))
print(data.dtypes)
print(len(data))
plt.plot(data['Date/Time'],data['LV ActivePower (kW)'])
plt.xlabel('Date and Time')
plt.ylabel('LV ActivePower (kW)')
plt.show()
#ADF test on dependent variable
X = data['LV ActivePower (kW)']
result = adfuller(X)
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))
data_acf=code_collection.calc_acf(data['LV ActivePower (kW)'],20)
code_collection.plt_acf(data_acf,"LV ActivePower (kW) for 20 lags",20)
data_acf_80=code_collection.calc_acf(data['LV ActivePower (kW)'],80)
code_collection.plt_acf(data_acf_80,"LV ActivePower (kW) for 80 lags",80)
data_acf_120=code_collection.calc_acf(data['LV ActivePower (kW)'],120)
code_collection.plt_acf(data_acf_120,"LV ActivePower (kW) for 120 lags",120)

gpac_data=code_collection.GPAC(data_acf,8,8)
print("Correlation between: LV ActivePower (kW)-Wind Speed (m/s)")
print(scipy.stats.pearsonr(data['LV ActivePower (kW)'],data['Wind Speed
(m/s)'])[0])
print("Correlation between: LV ActivePower (kW)-Theoretical_Power_Curve
(KWh)")
print(scipy.stats.pearsonr(data['LV ActivePower
(kW)'],data['Theoretical_Power_Curve (KWh)'])[0])
```

```python
print("Correlation between: LV ActivePower (kW)-Wind Direction (°)")
print(scipy.stats.pearsonr(data['LV ActivePower (kW)'],data['Wind Direction
(°)'])[0])
#plot correlation matrix
corr = data.corr()
#plt.figure(figsize=(10, 8))
ax = sns.heatmap(corr, vmin = -1, vmax = 1, annot = True)
bottom, top = ax.get_ylim()
ax.set_ylim(bottom + 0.5, top - 0.5)
plt.show()
#de trended data
#decpose seasonality
lv_act_pow1 = data['LV ActivePower (kW)']
#additive
result1 = seasonal_decompose(lv_act_pow1, model="additive",period=240)
result1.plot()
plt.show()
de_trended=data['LV ActivePower (kW)']-result1.seasonal
#seasonally adjusted data
plt.title("Removing Seosonality data plot")
plt.xlabel("Date")
plt.ylabel("LV Active Power")
plt.plot(data['Date/Time'],data['LV ActivePower (kW)'],label="Original data")
plt.plot(data['Date/Time'],de_trended,label="After removing sesonality data")
plt.legend()
plt.show()
#stength
st=result1.seasonal
tt=result1.trend
rt=result1.resid
#calculating strength trend
vart1=np.var(rt)
vart2=np.var( tt+ rt)
ft_1=1-(vart1/vart2)
ft_0=0-(vart1/vart2)
ft=max(ft_1,ft_0)
print("The strength of trend for data set is:",ft)
# calculating strength seasonality
vars1 = np.var(rt)
vars2 = np.var(rt + st)
fs_1 = 1 - (vars1 / vars2)
fs_0 = 0 - (vars1 / vars2)
fs = max(fs_1, fs_0)
print("The strength of seasonality for data set is:", fs)
#backward regression
print("Using Wind Speed (m/s), Theoretical_Power_Curve (KWh),Wind Direction
(°)")
elim1=['Wind Speed (m/s)', 'Theoretical_Power_Curve (KWh)','Wind Direction
(°)']
train1=data[elim1]
test1=data['LV ActivePower (kW)']
train1 = sm.add_constant(train1)
model1 = sm.OLS(test1, train1).fit()
print("Metrics:")
print("Adj R2", model1.rsquared_adj)
print("AIC", model1.aic)
print("BIC", model1.bic)
```

```
print("F statistic", model1.fvalue)
print("F p-value", model1.f_pvalue)
print("Using Wind Speed (m/s), Theoretical_Power_Curve (KWh)")
elim2=['Wind Speed (m/s)', 'Theoretical_Power_Curve (KWh)']
train2=data[elim2]
test2=data['LV ActivePower (kW)']
train2 = sm.add_constant(train2)
model2 = sm.OLS(test2, train2).fit()
print("Metrics:")
print("Adj R2", model2.rsquared_adj)
print("AIC", model2.aic)
print("BIC", model2.bic)
print("F statistic", model2.fvalue)
print("F p-value", model2.f_pvalue)
print("Using Wind Speed (m/s)")
elim3=['Wind Speed (m/s)']
train3=data[elim2]
test3=data['LV ActivePower (kW)']
train3 = sm.add_constant(train3)
model3 = sm.OLS(test2, train3).fit()
print("Metrics:")
print("Adj R2", model3.rsquared_adj)
print("AIC", model3.aic)
print("BIC", model3.bic)
print("F statistic", model3.fvalue)
print("F p-value", model3.f_pvalue)
#GPAC
gpac_matrix=np.asmatrix(gpac_data)
print("GPAC Values:")
column_labels=[]
row_labels=[]
for i in range(1,len(gpac_matrix)+1):
    column_labels.append(i)
    row_labels.append(i-1)
df = pd.DataFrame(gpac_data, columns=column_labels, index=row_labels)
print(df)
#split data
y_train=data[:4000]
y_test=data[4000:]
d_ml=data['Date/Time']
d_ml_train=d_ml[:4000]
d_ml_test=d_ml[4000:]
x=data.drop(columns=['LV ActivePower
(kW)','Date/Time','Theoretical_Power_Curve (KWh)'])
y=data['LV ActivePower (kW)']
#multiple linear regression
x_train1, x_test1,y_train1, y_test1 = train_test_split(x,y, test_size=0.20,
random_state=42, shuffle=False)
X = sm.add_constant(x_train1)
model = sm.OLS(y_train1,X).fit()
print("Multiple linear regression\n", model.summary())
#perform one step prediction
added_values = sm.add_constant(x_test1)
pred_ml=model.predict(added_values)
#plot data
plt.title("Multiple linear regression")
plt.xlabel("Date and Time")
```

```
plt.ylabel("LV Active Power")
plt.plot(d_ml_train,y_train1,label='Train data')
plt.plot(d_ml_test,y_test1, label='Test data')
plt.plot(d_ml_test,pred_ml,label='Predicted data')
plt.legend()
plt.show()
res_l=y_test1-pred_ml
#test for multiple linear regression
print("Metrics:")
print("Adj R2", model.rsquared_adj)
print("AIC", model.aic)
print("BIC", model.bic)
print("F statistic", model.fvalue)
print("F p-value", model.f_pvalue)
res_l=list(res_l)
acf_model_l=code_collection.calc_acf(res_l,lags=120)
code_collection.plt_acf(acf_model_l,"Plot ACF of Multilinear Linear
Regression",120)
q_vv=code_collection.q_val(y_test1,acf_model_l)
print("Q value",q_vv)
print("Mean of residue",np.mean(res_l))
print("Variance of residue",np.var(res_l))
#y_train,y_test=train_test_split(data,shuffle=False, test_size=0.20)
y_test_drift=y_test[:-1]
len(y_test_drift)
#prediction_avg=pd.DataFrame({"Month": test["Month"], "#Passengers":avg})
#base models
y_train['LV ActivePower (kW)']=y_train['LV ActivePower (kW)']-
np.mean(y_train['LV ActivePower (kW)'])
avg,av_er,av_mse=code_collection.calc_avg(y_train['LV ActivePower
(kW)'],y_test['LV ActivePower (kW)'])
avg=avg+np.mean(y_train['LV ActivePower (kW)'])
plt.title("All data- Average Model")
plt.xlabel("Date")
plt.ylabel("LV Active Power")
plt.plot(y_train["Date/Time"],y_train['LV ActivePower (kW)'],'b--')
plt.plot(y_test["Date/Time"],y_test['LV ActivePower (kW)'],'g--')
plt.plot(y_test["Date/Time"],avg,'r--')
plt.show()
naive,nav_er,nav_mse=code_collection.calc_naive(y_train['LV ActivePower
(kW)'],y_test['LV ActivePower (kW)'])
naive=naive+np.mean(y_train['LV ActivePower (kW)'])
plt.title("All data- Naive Model")
plt.xlabel("Date")
plt.ylabel("LV Active Power")
plt.plot(y_train["Date/Time"],y_train['LV ActivePower (kW)'],'b--')
plt.plot(y_test["Date/Time"],y_test['LV ActivePower (kW)'],'g--')
plt.plot(y_test["Date/Time"],naive,'r--')
plt.show()
drift,d_er,d_mse=code_collection.calc_drift(y_train['LV ActivePower
(kW)'],y_test['LV ActivePower (kW)'])
drift=drift+np.mean(y_train['LV ActivePower (kW)'])
plt.title("All data- Drift Model")
plt.xlabel("Date")
plt.ylabel("LV Active Power")
plt.plot(y_train["Date/Time"],y_train['LV ActivePower (kW)'],'b--')
plt.plot(y_test["Date/Time"],y_test['LV ActivePower (kW)'],'g--')
```

```
plt.plot(y_test_drift["Date/Time"],drift,'r--')
plt.show()
ses,s_er,s_mse=code_collection.calc_drift(y_train['LV ActivePower
(kW)'],y_test['LV ActivePower (kW)'])
ses=ses+np.mean(y_train['LV ActivePower (kW)'])
plt.title("All data- SES Model")
plt.xlabel("Date")
plt.ylabel("LV Active Power")
plt.plot(y_train["Date/Time"],y_train['LV ActivePower (kW)'],'b--')
plt.plot(y_test["Date/Time"],y_test['LV ActivePower (kW)'],'g--')
plt.plot(y_test_drift["Date/Time"],ses,'r--')
plt.show()
#printing base models only with test set
split_day=y_test["Date/Time"]
split_lv=y_test['LV ActivePower (kW)']
avg_day=avg
naive_day=naive
drift_day=drift
ses_day=ses
plt.title("Daily data- Naive Model")
plt.xlabel("Date")
plt.ylabel("LV Active Power")
plt.plot(split_day[:105],split_lv[:105],'b-.')
plt.plot(split_day[:105],naive_day[:105],'g-.')
plt.xticks(rotation=90)
plt.show()
plt.title("Daily data- Drift Model")
plt.xlabel("Date")
plt.ylabel("LV Active Power")
plt.plot(split_day[:105],split_lv[:105],'b-.')
plt.plot(split_day[:105],drift_day[:105],'g-.')
plt.xticks(rotation=90)
plt.show()
plt.title("Daily data- SES Model")
plt.xlabel("Date")
plt.ylabel("LV Active Power")
plt.plot(split_day[:105],split_lv[:105],'b-.')
plt.plot(split_day[:105],ses_day[:105],'g-.')
plt.xticks(rotation=90)
plt.show()
#arma process
arma_date=y_test['LV ActivePower (kW)']
#start_date=arma_date.loc[0]
#end_date=arma_date.loc[-1]
arma_error_1,arma_error_2, arma_error_3, arma_error_4=[],[],[],[]
arma_1=sm.tsa.ARMA(y_train['LV ActivePower (kW)'],
order=(1,1)).fit(disp=False)
arma_pred_1=arma_1.forecast(len(y_test['LV ActivePower (kW)']))[0]
arma_pred_1=arma_pred_1+np.mean(y_train['LV ActivePower (kW)'])
for i,j in zip(y_test['LV ActivePower (kW)'],arma_pred_1):
   arma_error_1.append(i-j)
arma_2=sm.tsa.ARMA(y_train['LV ActivePower (kW)'],
order=(1,3)).fit(disp=False)
arma_pred_2=arma_2.forecast(len(y_test['LV ActivePower (kW)']))[0]
arma_pred_2=arma_pred_2+np.mean(y_train['LV ActivePower (kW)'])
for i,j in zip(y_test['LV ActivePower (kW)'],arma_pred_2):
   arma_error_2.append(i-j)
```

```
arma_3=sm.tsa.ARMA(y_train['LV ActivePower (kW)'],
order=(2,1)).fit(disp=False)
arma_pred_3=arma_3.forecast(len(y_test['LV ActivePower (kW)']))[0]
arma_pred_3=arma_pred_3+np.mean(y_train['LV ActivePower (kW)'])
for i,j in zip(y_test['LV ActivePower (kW)'],arma_pred_3):
   arma_error_3.append(i-j)
arma_4=sm.tsa.ARMA(y_train['LV ActivePower (kW)'],
order=(2,3)).fit(disp=False)
arma_pred_4=arma_4.forecast(len(y_test['LV ActivePower (kW)']))[0]
arma_pred_4=arma_pred_4+np.mean(y_train['LV ActivePower (kW)'])
for i,j in zip(y_test['LV ActivePower (kW)'],arma_pred_4):
   arma_error_4.append(i-j)
ch1=chisquare(arma_error_1)[1]
ch2=chisquare(arma_error_2)[1]
ch3=chisquare(arma_error_3)[1]
ch4=chisquare(arma_error_4)[1]
r_a1=np.sqrt(np.mean(arma_error_1))
r_a2=np.sqrt(np.mean(arma_error_2))
r_a3=np.sqrt(np.mean(arma_error_3))
r_a4=np.sqrt(np.mean(arma_error_4))
print("Chi Sq test for ARMA (1,1)", ch1)
print("Chi Sq test for ARMA (1,3)", ch2)
print("Chi Sq test for ARMA (2,1)", ch3)
print("Chi Sq test for ARMA (2,3)", ch4)
#RMSE
print("RMSE for ARMA (1,1)", r_a1)
print("RMSE for ARMA (1,3)", r_a2)
print("RMSE for ARMA (2,1)", r_a3)
print("RMSE for ARMA (2,3)", r_a4)
print("FINAL ARMA ORDER:(1,3)")
arma=sm.tsa.ARMA(y_train['LV ActivePower (kW)'], order=(1,3)).fit(disp=False)
arma_pred=arma.forecast(len(y_test['LV ActivePower (kW)']))[0]
arma_pred=arma_pred+np.mean(y_train['LV ActivePower (kW)'])
#arma model details
print("ARMA model summary")
print(arma.summary())
print("ARMA confidence interval")
print(arma.conf_int())
plt.title("Daily data- ARMA Model")
plt.xlabel("Date")
plt.ylabel("LV Active Power")
plt.plot(split_day[:105],split_lv[:105],'b-.')
plt.plot(split_day[:105],arma_pred[:105],'g-.')
plt.xticks(rotation=90)
plt.show()
arma_error,mse_arma=[],[]
for i,j in zip(y_test['LV ActivePower (kW)'],arma_pred):
   arma_error.append(i-j)
   mse_arma.append((i-j)**2)
#Holt winter model
model2=ets.Holt(y_train['LV ActivePower (kW)'],
initialization_method="estimated").fit()
holt_pred=model2.forecast(len(y_test['LV ActivePower (kW)']))
holt_pred=holt_pred+np.mean(y_train['LV ActivePower (kW)'])
plt.title("All data- Holt Winter Model")
plt.xlabel("Date")
plt.ylabel("LV Active Power")
```

```
plt.plot(y_train['Date/Time'],y_train['LV ActivePower (kW)'],'b--')
plt.plot(y_test['Date/Time'],y_test['LV ActivePower (kW)'],'g--')
plt.plot(y_test['Date/Time'],holt_pred,'r--')
plt.xticks(rotation=90)
plt.show()
plt.title("Daily data- Holt Winter Model")
plt.xlabel("Date")
plt.ylabel("LV Active Power")
plt.plot(split_day[:105],split_lv[:105],'b-.')
plt.plot(split_day[:105],holt_pred[:105],'g-.')
plt.xticks(rotation=90)
plt.show()
holt_error,mse_holt=[],[]
for i,j in zip(y_test['LV ActivePower (kW)'],holt_pred):
   holt_error.append(i-j)
   mse_holt.append((i-j)**2)
#holt seasonal model
model_holt_seasonal=ets.ExponentialSmoothing(y_train['LV ActivePower (kW)'],
initialization_method="estimated").fit()
holt_pred_s=model_holt_seasonal.forecast(len(y_test['LV ActivePower (kW)']))
holt_pred_s=holt_pred_s+np.mean(y_train['LV ActivePower (kW)'])
plt.title("All data- Holt Winter Seasonal Model")
plt.xlabel("Date")
plt.ylabel("LV Active Power")
plt.plot(y_train['Date/Time'],y_train['LV ActivePower (kW)'],'b--')
plt.plot(y_test['Date/Time'],y_test['LV ActivePower (kW)'],'g--')
plt.plot(y_test['Date/Time'],holt_pred_s,'r--')
plt.xticks(rotation=90)
plt.show()
plt.title("Daily data- Holt Winter Seasonal Model")
plt.xlabel("Date")
plt.ylabel("LV Active Power")
plt.plot(split_day[:105],split_lv[:105],'b-.')
plt.plot(split_day[:105],holt_pred_s[:105],'g-.')
plt.xticks(rotation=90)
plt.show()
holt_error_s,mse_holt_s=[],[]
for i,j in zip(y_test['LV ActivePower (kW)'],holt_pred_s):
   holt_error_s.append(i-j)
   mse_holt_s.append((i-j)**2)
#ARIMA model
arima=sm.tsa.arima.ARIMA(y_train['LV ActivePower (kW)'],order=(1,1,3)).fit()
arima_pred=arima.forecast(len(y_test['LV ActivePower (kW)']))
arima_pred=arima_pred+np.mean(y_train['LV ActivePower (kW)'])
plt.title("All data- ARIMA Model")
plt.xlabel("Date")
plt.ylabel("LV Active Power")
plt.plot(y_train['Date/Time'],y_train['LV ActivePower (kW)'],'b--')
plt.plot(y_test['Date/Time'],y_test['LV ActivePower (kW)'],'g--')
plt.plot(y_test['Date/Time'],arima_pred,'r--')
plt.xticks(rotation=90)
plt.show()
plt.title("Daily data-ARIMA Model")
plt.plot(split_day[:105],split_lv[:105],'b-.')
plt.plot(split_day[:105],arima_pred[:105],'g-.')
plt.xticks(rotation=90)
plt.show()
```

```
arima_error,mse_arima=[],[]
for i,j in zip(y_test['LV ActivePower (kW)'],arima_pred):
   arima_error.append(i-j)
   mse_arima.append((i-j)**2)
#mean errors
print("Mean of error of Average Model",np.mean(av_er))
print("Mean of error of Naive Model",np.mean(nav_er))
print("Mean of error of Drift Model",np.mean(d_er))
print("Mean of error of SES Model",np.mean(s_er))
print("Mean of error of ARMA Model",np.mean(arma_error))
print("Mean of error of Holt Model",np.mean(holt_error))
print("Mean of error of Holt Seasonal Model",np.mean(holt_error_s))
print("Mean of error of ARIMA Model",np.mean(arima_error))
#variance errors
print("Variance of error of Average Model",np.var(av_er))
print("Variance of error of Naive Model",np.var(nav_er))
print("Variance of error of Drift Model",np.var(d_er))
print("Variance of error of SES Model",np.var(s_er))
print("Variance of error of ARMA Model",np.var(arma_error))
print("Variance of error of Holt Model",np.var(holt_error))
print("Variance of error of Holt Seasonal Model",np.var(holt_error_s))
print("Variance of error of ARIMA Model",np.var(arima_error))
# calc acf of errors:
acf_avg=code_collection.calc_acf(av_er,120)
acf_naive=code_collection.calc_acf(nav_er,120)
acf_drift=code_collection.calc_acf(d_er,120)
acf_ses=code_collection.calc_acf(s_er,120)
acf_arma=code_collection.calc_acf(arma_error,120)
acf_holt=code_collection.calc_acf(holt_error,120)
acf_holt_s=code_collection.calc_acf(holt_error_s,120)
acf_arima=code_collection.calc_acf(arima_error,120)
#plot acf of errors:
code_collection.plt_acf(acf_avg,"Average",120)
code_collection.plt_acf(acf_naive,"Naive",120)
code_collection.plt_acf(acf_drift,"Drift",120)
code_collection.plt_acf(acf_ses,"SES",120)
code_collection.plt_acf(acf_arma,"ARMA",120)
code_collection.plt_acf(acf_arima,"ARIMA",120)
code_collection.plt_acf(acf_holt,"Holt Winter",120)
code_collection.plt_acf(acf_holt_s,"Holt Winter Seasonal",120)
#printing MSE:
print("MSE of Average Base Model:",av_mse)
print("MSE of Drift Base Model:",d_mse)
print("MSE of Naive Base Model:",nav_mse)
print("MSE of SES:",s_mse)
print("MSE of ARMA process", np.mean(mse_arma))
print("MSE of Holt Winter",np.mean(mse_holt))
print("MSE of Holt Winter Seasonal",np.mean(mse_holt_s))
print("MSE of ARIMA process", np.mean(mse_arima))
#printing RMSE
print("RMSE of Average Base Model:",np.sqrt(av_mse))
print("RMSE of Drift Base Model:",np.sqrt(d_mse))
print("RMSE of Naive Base Model:",np.sqrt(nav_mse))
print("RMSE of SES:",np.sqrt(s_mse))
print("RMSE of ARMA process", np.sqrt(np.mean(mse_arma)))
print("RMSE of Holt Winter",np.sqrt(np.mean(mse_holt)))
print("RMSE of Holt Winter Seasonal",np.sqrt(np.mean(mse_holt_s)))
```

```
print("RMSE of ARIMA process", np.sqrt(np.mean(mse_arima)))
#q values
q_avg=code_collection.q_val(y_test['LV ActivePower (kW)'],acf_avg)
print("Q value Average Model",q_avg)
q_naive=code_collection.q_val(y_test['LV ActivePower (kW)'],acf_naive)
print("Q value Naive Model",q_naive)
q_drift=code_collection.q_val(y_test['LV ActivePower (kW)'],acf_drift)
print("Q value Drift Model",q_drift)
q_ses=code_collection.q_val(y_test['LV ActivePower (kW)'],acf_ses)
print("Q value SES Model",q_ses)
q_holt=code_collection.q_val(y_test['LV ActivePower (kW)'],acf_holt)
print("Q value Holt Model",q_holt)
q_holt_s=code_collection.q_val(y_test['LV ActivePower (kW)'],acf_holt_s)
print("Q value Holt Seasonal Model",q_holt_s)
q_arma=code_collection.q_val(y_test['LV ActivePower (kW)'],acf_arma)
print("Q value ARMA Model",q_arma)
q_arima=code_collection.q_val(y_test['LV ActivePower (kW)'],acf_arima)
print("Q value ARIMA Model",q_arima)
```

LM_tester.py-

```
from FinalProjectFunctions import *
from sklearn.model_selection import train_test_split
import statsmodels.api as sm
from scipy import signal
import warnings
warnings.filterwarnings("ignore")

# IMPORT DATA
df = pd.read_csv('T1.csv', header=0)
data = df.copy()
data['Date/Time']=pd.to_datetime(data['Date/Time'])

dep_var = data['LV ActivePower (kW)']
dep_var=dep_var[:5000]
train, test = train_test_split(dep_var, shuffle=False, test_size=0.2)
# print(train.shape, test.shape)   # (4146,) (1037,)
# print(train.head())
# print(test.head())
def step_0(na,nb):
    theta = np.zeros(shape=(na+nb,1))
    return theta.flatten()

def white_noise_simulation(theta,na,y):
    num = [1] + list(theta[na:])
    den = [1] + list(theta[:na])
    while len(num) < len(den):
        num.append(0)
    while len(num) > len(den):
        den.append(0)
    system = (den, num, 1)
    tout, e = signal.dlsim(system, y)
    e = [a[0] for a in e]
    return np.array(e)
```

```python
def step_1(theta,na,nb,delta,y):
    e = white_noise_simulation(theta,na,y)
    SSE = np.matmul(e.T, e)
    X_all = []
    for i in range(na+nb):
        theta_dummy = theta.copy()
        theta_dummy[i] = theta[i] + delta
        e_n = white_noise_simulation(theta_dummy,na,y)
        X_i = (e - e_n)/delta
        X_all.append(X_i)

    X = np.column_stack(X_all)
    A = np.matmul(X.T,X)
    g = np.matmul(X.T,e)
    return A,g,SSE

def step_2(A,mu,g,theta,na,y):
    I = np.identity(g.shape[0])
    theta_d = np.matmul(np.linalg.inv(A+(mu*I)),g)
    theta_new = theta + theta_d
    e_new = white_noise_simulation(theta_new,na,y)
    SSE_new = np.matmul(e_new.T,e_new)
    if np.isnan(SSE_new):
        SSE_new = 10 ** 10
    return SSE_new, theta_d, theta_new


with np.errstate(divide='ignore'):
    np.float64(1.0) / 0.0

def step_3(max_iterations, mu_max, na, nb, y, mu, delta):
    iteration_num = 0
    SSE = []
    theta = step_0(na, nb)
    while iteration_num < max_iterations:
        print('Iteration ', iteration_num)

        A, g, SSE_old = step_1(theta, na, nb, delta, y)
        print('old SSE : ', SSE_old)
        if iteration_num == 0:
            SSE.append(SSE_old)
        SSE_new, theta_d, theta_new = step_2(A, mu, g, theta, na, y)
        print('new SSE : ', SSE_new)
        SSE.append(SSE_new)

        if SSE_new < SSE_old:
            print('Norm of delta_theta :', np.linalg.norm(theta_d))
            if np.linalg.norm(theta_d) < 1e-3:
                theta_hat = theta_new
                e_var = SSE_new / (len(y) - A.shape[0])
                cov = e_var * np.linalg.inv(A)
                print('\n **** Algorithm Converged **** \n')
                return SSE, theta_hat, cov, e_var
            else:
                theta = theta_new
                mu = mu / 10
```

```
        while SSE_new >= SSE_old:
            mu = mu * 10
            if mu > mu_max:
                print('mu exceeded the max limit')
                return None, None, None, None
            SSE_new, theta_d, theta_new = step_2(A, mu, g, theta, na, y)

        theta = theta_new

        iteration_num+=1
        if iteration_num > max_iterations:
            print('Max iterations reached')
            return None, None, None, None


np.random.seed(10)
mu_factor = 10
delta = 1e-6
epsilon = 0.001
mu = 0.01
max_iterations = 100
mu_max = 1e10

na = 1
nb = 3

SSE, est_params, cov, e_var = step_3(max_iterations, mu_max, na, nb, train,
mu, delta)
print('Estimated parameters : ', est_params)
print('Estimated Covariance matrix : ', cov)
print('Estimated variance of error : ', e_var)

def SSEplot(SSE):
    plt.figure()
    plt.plot(SSE, label = 'Sum Squared Error')
    plt.xlabel('# of Iterations')
    plt.ylabel('Sum Squared Error')
    plt.legend()
    plt.show()

SSEplot(SSE)
```

term_proj_ARMA.py-

```
import numpy as np
import pandas as pd
import warnings
import scipy
import code_collection
import statsmodels.api as sm
from matplotlib import style
import seaborn as sns
from scipy.stats import chisquare
import statsmodels.tsa.holtwinters as ets
from statsmodels.tsa.seasonal import seasonal_decompose
import matplotlib.pyplot as plt
```

```
from statsmodels.tsa.stattools import adfuller
from sklearn.model_selection import train_test_split
#style.use('ggplot')
np.set_printoptions(suppress=True)
data=pd.read_csv("T1.csv",header=0)
#selecting only the first 5000 data points
data=data[:5000]
data['Date/Time']=pd.to_datetime(data['Date/Time'])
date_check=data['Date/Time']
print("Start date:",date_check[0])
print("End date:",date_check.iloc[-1])
#split data
y_train=data[:4000]
y_test=data[4000:]
teta=[0.99378114,0.00519156,0.05946741,0.02693868]
#one step
y_t=y_train['LV ActivePower (kW)']
y_tt=y_test['LV ActivePower (kW)']
y_hat_t_1 = []
for i in range(0,len(y_t)):
    if i==0:
        y_hat_t_1.append(y_t[i]*teta[0] +teta[2]* y_t[i])
    elif i==1:
        y_hat_t_1.append(y_t[i]*teta[0] +teta[1]* y_t[i-1] + teta[2]*(y_t[i]
- y_hat_t_1[i - 1] ) + teta[3]*(y_t[i - 1]))
    else:
        y_hat_t_1.append( y_t[i]*teta[0] +teta[1]* y_t[i-1] + teta[2]*(y_t[i]
- y_hat_t_1[i - 1] ) + teta[3]*(y_t[i - 1] - y_hat_t_1[i-2]))
#forecast function
y_hat_t_h = []
for h in range(0,len(y_tt)):
    if h==0:
        y_hat_t_h.append(y_t.iloc[-1]*teta[0] +teta[1]*y_t.iloc[-2]+
teta[2]*(y_t.iloc[-1] - y_hat_t_1[-2]) + teta[3]*(y_t.iloc[-2]-y_hat_t_1[-
3]))
    elif h==1:
         y_hat_t_h.append(y_hat_t_h[h-1]*teta[0] + teta[1]*y_t.iloc[-1] +
teta[3]*(y_t.iloc[-1] - y_hat_t_1[-2]))

    else:
        y_hat_t_h.append(y_hat_t_h[h-1]*teta[0] -teta[1]*y_hat_t_h[h-2] )
#errors,mse,rmse,acf,q value
error,mse=[],[]
for i,j in zip(y_test['LV ActivePower (kW)'],y_hat_t_h):
   error.append(i-j)
   mse.append((i-j)**2)
MSE=np.mean(mse)
rmse=np.sqrt(MSE)
acf=code_collection.calc_acf(error,120)
code_collection.plt_acf(acf,"ACF of ARMA H step",120)
q_val=code_collection.q_val(y_test['LV ActivePower (kW)'],acf)
plt.title("Manual Calcs- ARMA Model")
plt.xlabel("Date")
plt.ylabel("LV Active Power")
plt.plot(y_train['Date/Time'],y_train['LV ActivePower (kW)'],'b--')
plt.plot(y_test['Date/Time'],y_test['LV ActivePower (kW)'],'g--')
plt.plot(y_test['Date/Time'],y_hat_t_h,'r--')
```

```
plt.show()
print("MSE:",MSE)
print("RMSE",rmse)
print("Q value", q_val)
```

term_proj_ARMA_diff.py-

```
import numpy as np
import pandas as pd
import warnings
import scipy
import code_collection
import statsmodels.api as sm
from matplotlib import style
import seaborn as sns
from scipy.stats import chisquare
import statsmodels.tsa.holtwinters as ets
from statsmodels.tsa.seasonal import seasonal_decompose
import matplotlib.pyplot as plt
from statsmodels.tsa.stattools import adfuller
from sklearn.model_selection import train_test_split
#style.use('ggplot')
np.set_printoptions(suppress=True)
data=pd.read_csv("T1.csv",header=0)
#selecting only the first 5000 data points
data=data[:5000]
data['Date/Time']=pd.to_datetime(data['Date/Time'])
date_check=data['Date/Time']
print("Start date:",date_check[0])
print("End date:",date_check.iloc[-1])
#split data
target=data['LV ActivePower (kW)']
data['LV ActivePower (kW)']= target.diff().dropna(axis=0)
y_train=target[:4000]
y_test=target[4000:]
#x_train, x_test,y_train, y_test= train_test_split(x,y, test_size=0.20,
random_state=42, shuffle=False)
#EDA-data conversion
data['LV ActivePower (kW)']=data['LV ActivePower (kW)'][1:]
sm.graphics.tsa.plot_acf(data['LV ActivePower (kW)'], lags=40)
plt.show()
data_acf=code_collection.calc_acf(data['LV ActivePower (kW)'],10)
code_collection.plt_acf(data_acf,"LV ActivePower (kW) for 20 lags",10)

gpac_data=code_collection.GPAC(data_acf,8,8)
#GPAC
gpac_matrix=np.asmatrix(gpac_data)
print("GPAC Values:")
column_labels=[]
row_labels=[]
for i in range(1,len(gpac_matrix)+1):
    column_labels.append(i)
    row_labels.append(i-1)
df = pd.DataFrame(gpac_data, columns=column_labels, index=row_labels)
print(df)
```

Refrences:

1. Dr. Reza Jafari Class slides
2. Lab codes