Appendix

```
import numpy as np
import pandas as pd
import statsmodels.api as sm
from scipy import signal
import scipy
import math
from scipy.stats import chi2
import numpy.linalg as LA
import seaborn as sns
import matplotlib.pyplot as plt
from matplotlib.colors import LogNorm
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.feature_selection import SelectKBest,f_classif
from sklearn.metrics import mean_squared_error
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
from statsmodels.tsa.seasonal import STL
from statsmodels.stats.outliers_influence import variance_inflation_factor
from statsmodels.tsa.stattools import adfuller
from statsmodels.tsa.stattools import kpss
from sklearn.model_selection import train_test_split
import pmdarima as pm
import statsmodels.tsa.holtwinters as ets
from sklearn.preprocessing import StandardScaler
from xgboost import XGBRegressor
from sklearn.decomposition import PCA
import plotly.express as px
from statsmodels.tsa.arima.model import ARIMA
from sklearn.metrics import mean_squared_error
# import matplotlib
# matplotlib.use('TkAgg')
from sklearn.metrics import r2_score


features = pd.read_csv('features.csv')
power = pd.read_csv('power.csv')
df = pd.merge(features,power, how='inner', on = 'Timestamp')
df['Timestamp'] = pd.to_datetime(df.iloc[:,0])

#checking if timestamps are equidistant
time_diff = df['Timestamp'].diff()
if all(time_diff == 600):
    print("The timestamps are equally separated by 10 minutes")
else:
    print("The timestamps are not equally separated by 10 minutes")

#we will now down sample the data
df.set_index('Timestamp', inplace=True)
df = df.resample('H').mean()
df.fillna(df.mean(),inplace=True) #or use interpolate on particular column or 
try using it on a dataframe
df.to_csv("Downsampled_Dataset.csv")
df = pd.read_csv("Downsampled_Dataset.csv",parse_dates=True)
```

```
df['Timestamp'] = pd.to_datetime(df['Timestamp'])
print("Timestamps have been set to hourly basis and data is now ready to be
used")

# select the index values where the time difference is not 3600 seconds
df['time_diff1'] = (df['Timestamp'] - df['Timestamp'].shift(1)).dt.seconds
mask = df['time_diff1'] != 3600
df['time_diff1'].to_csv("checkDownsampled_Dataset.csv")
result = df.loc[mask, 'Timestamp'].index

print("The non equidistant timestamps after sampling are - ", result)
df = df.drop(columns='time_diff1')


date = pd.date_range(start = '1/1/2019',
                     end = '08/14/2021',
                     periods = len(df))


fig,ax = plt.subplots(figsize = (16,8))
ax.plot(df['Timestamp'], df['Power(kW)'])
ax.set_title('Turbine Power Generation')
ax.set_xlabel('Time')
ax.set_ylabel('Power (kW)')
ax.margins(x=0)
plt.show()

def Cal_rolling_mean_var(sliced_df):
    rollingMean = []
    rollingVariance = []
    for i in range(len(sliced_df)):
        mean = (sliced_df.iloc[0:i]).mean()
        rollingMean.append(mean)
        variance = np.var(sliced_df.iloc[0:i])
        rollingVariance.append(variance)

    rollingVariance = pd.Series(rollingVariance)
    rollingVariancenotna = rollingVariance.notna()
    newrollingVariance = rollingVariance[rollingVariancenotna].tolist()


    fig, axes = plt.subplots(2, 1, figsize=(15,8))

    axes[0].plot(rollingMean)
    axes[0].set(xlabel='Samples',
            ylabel='Magnitude')
    axes[0].set_title('Rolling Mean-'+sliced_df.columns[0])

    axes[1].plot(newrollingVariance)
    axes[1].set(xlabel='Samples',
            ylabel='Magnitude')
    axes[1].set_title('Rollling Variance-'+sliced_df.columns[0])
    plt.legend(['Mean and Variance'],loc="lower right")
    plt.tight_layout()
    plt.show()
def ADF_Cal(x):
    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))
def kpss_test(timeseries):
    print ('Results of KPSS Test:')
    kpsstest = kpss(timeseries, regression='c', nlags="auto")
    kpss_output = pd.Series(kpsstest[0:3], index=['Test Statistic','p-
value','Lags Used'])
    for key,value in kpsstest[3].items():
        kpss_output['Critical Value (%s)'%key] = value
    print (kpss_output)
def ACF_PACF_Plot(y, lags):
    acf = sm.tsa.stattools.acf(y, nlags=lags)
    pacf = sm.tsa.stattools.pacf(y, nlags=lags)
    fig = plt.figure(figsize=(16,8))
    plt.subplot(211)
    plt.suptitle('ACF/PACF of data')
    plot_acf(y, ax=plt.gca(), lags=lags)
    plt.subplot(212)
    plot_pacf(y, ax=plt.gca(), lags=lags)
    fig.tight_layout(pad=3)
    plt.show()
def norderdiff(DF, col_name, order):
    # name = col_name + '_' + str(order) + '_Diff'
    # DF[name] = 0
    # temp1 = DF[col_name][::-1]
    # temp2 = temp1[0:-order] - temp1.values[order:]
    # DF[name] = temp2[::-1]
    DF[f'{col_name}_diff_{order}'] = DF[col_name].diff(order)
    return DF
def calGPAC(ry3,j,k):
    df = pd.DataFrame(0, index=range(j), columns=range(1, k))
    for a in range(1,k):
        for b in range(0,j):
            ws = (len(ry3) // 2) + a
            we = (len(ry3) // 2) + b
            num = np.zeros((a,a))
            den = np.zeros((a,a))
            for c in range(0,a+1): #loop for stacking vertical slices one
step ahead to make the a*a matrix
                if c>0 and c!=a: #if a = 6 for ex this will run for c=1 2 3 4
5

                    column = np.array(ry3[ws-a+b-c:we+a-c].values)
                    den[:,c] = column
                    #b=b-1
                elif c==0: #only for 0th column
                    column=0
                    column = np.array(ry3[ws-a+b:we+a].values)
                    den[:,c] = column
                    #b=b-1
                elif c == a:  #we allow for c in range(0,a+1) for this elif
c==a to run so that we can do num den thing
                    num = den.copy()
                    finalcol = np.array(ry3[ws-a+b+1:we+a+1].values) #add one
```

```
to those indexes to make your last column
                    num[:,-1] = finalcol
                    determnum = np.linalg.det(num)
                    determden = np.linalg.det(den)

                    if abs(determden) < 1e-5:
                        phi = np.inf
                        df.loc[b, a] = phi
                    else:
                        phi = determnum / determden
                        df.loc[b, a] = phi
                else:
                    print("Code has conceptual issues")
    #arrfinal = df.to_numpy()
    sns.heatmap(df,annot=True,fmt='.3f',linewidths=0.5)
    plt.title("Generalized Partial Autocorrelation function (GPAC) Table")
    plt.tight_layout()
    plt.show()
    return df
def paramEst(y, theta, N,na,nb):
    theta = theta.reshape(len(theta),1)
    mu = 0.01
    mu_factor = 10
    delta = 0.000001
    epsilon = 0.001
    mu_max = 1e20
    MAX = 100
    #Step 1
    n=na+nb
    numIter = 0
    SSEcompiled = []
    while numIter< MAX:
        MAPARAMLOOPNUM = [0] * max(na, nb)
        ARPARAMLOOPDEN = [0] * max(na, nb)

        MAPARAMLOOPNUM[:nb] = theta[na:]
        ARPARAMLOOPDEN[:na] = theta[:na]

        num = np.insert(MAPARAMLOOPNUM, 0, 1).tolist()
        den1 = np.insert(ARPARAMLOOPDEN, 0, 1).tolist()
        den = [valchk if not isinstance(valchk,np.ndarray) else
valchk.flatten()[0] for valchk in den1]

        system = (den, num, 1)  # den,num,1 because we are generating
synthetic error
        _, eNew = signal.dlsim(system, y)
        #eNew = eNew[:, 0]####
        SSE = np.dot(eNew.T , eNew)  # SSE = e^T.e
        SSEcompiled.append(SSE.ravel().flatten())

        X = np.empty((N, n)).reshape(N, n)
        for i in range(0, n):  # X construction loop
            thetatemp = theta.copy()
            thetatemp[i] = thetatemp[i] + delta

            MAPARAMLOOPNUM = [0] * max(na, nb)
            ARPARAMLOOPDEN = [0] * max(na, nb)
```

```
            MAPARAMLOOPNUM[:nb] = thetatemp[na:]
            ARPARAMLOOPDEN[:na] = thetatemp[:na]

            num = np.insert(MAPARAMLOOPNUM, 0, 1).tolist()
            den1 = np.insert(ARPARAMLOOPDEN, 0, 1).tolist()
            den = [valchk if not isinstance(valchk, np.ndarray) else
valchk.flatten()[0] for valchk in den1]

            system = (den, num, 1)  # den,num,1 because we are generating
synthetic error
            _, eNewi = signal.dlsim(system, y)

            X[:, i] = ((eNew.ravel() - eNewi.ravel()) / delta)  # array shape
is flat

        A = np.dot(X.T, X)
        g = np.dot(X.T, eNew)

        # Step2
        delta_theta = np.dot(LA.inv(A + (mu * np.identity(n))) , g)
        thetaNew = theta + delta_theta
        MAPARAMNUMthetaNew = [0] * max(na, nb)
        ARPARAMDENthetaNew = [0] * max(na, nb)

        ARPARAMDENthetaNew[:na] = thetaNew[:na]
        MAPARAMNUMthetaNew[:nb] = thetaNew[na:]

        num = np.insert(MAPARAMNUMthetaNew, 0, 1).tolist()
        den1 = np.insert(ARPARAMDENthetaNew, 0, 1).tolist()
        den = [valchk if not isinstance(valchk, np.ndarray) else
valchk.flatten()[0] for valchk in den1]

        system = (den, num, 1)  # den,num,1 because we are generating
synthetic error
        _, eNewthetaNew = signal.dlsim(system, y)
        SSENew = np.dot(eNewthetaNew.T,eNewthetaNew)

        if SSENew<SSE:
            if LA.norm(delta_theta)<epsilon:
                thetacap = thetaNew.copy()
                variancecapofError = (SSENew/(N-n))
                sdval = np.sqrt(variancecapofError)
                covariancethetacap = variancecapofError*(LA.inv(A))
                covDiag = np.diag(covariancethetacap)
                print(f"Estimated params are:\n {thetacap}")
                # print(f"True params are: {AN[1:na+1] + BN[1:nb+1]}")
                print(f"Standard deviation of the Error is : {sdval}")
                print(f"Estimated variance of Error is :
{variancecapofError}")
                numMARoots = np.roots(num)
                denARRoots = np.roots(den)

                for b in range(0,len(numMARoots)):
                    print(f"Roots of the numerators is/are- {numMARoots[b]}")

                for a in range(0,len(denARRoots)):
```

```
                    print(f"Roots of the denominator is/are-
{denARRoots[a]}")

                confIntna = thetacap[:na]
                confIntnb = thetacap[na:]
                for l in range(0,na):
                    print(f"The Confidence interval for a{l+1} is -
{confIntna[[l]] - 2*(np.sqrt(covDiag[[l]]))} < a{l+1} < {confIntna[[l]] +
2*(np.sqrt(covDiag[[l]]))} ")

                for m in range(0,nb):
                    print(f"The Confidence interval for b{m+1} is -
{confIntnb[[m]] - 2*(np.sqrt(covDiag[[m]]))} < b{m+1} < {confIntnb[[m]] +
2*(np.sqrt(covDiag[[m]]))} ")

                print(f"The Covariance Matrix is:\n {covariancethetacap}")
                plt.plot(SSEcompiled, label='S.S.E.', color='purple')
                plt.grid()
                plt.legend()
                plt.margins(x=0)
                plt.xticks(np.arange(0, numIter + 1, step=1))
                plt.title('Sum of Squared Errors ')
                plt.xlabel('Iterations')
                plt.ylabel('Magnitude of Error')
                plt.figure(figsize=(16, 8))
                plt.show()
                return "Correct"
            else:
                theta = thetaNew.copy()
                mu = mu/10

        while SSENew>=SSE:
            mu = mu*10
            if mu>mu_max:
                thetacap = thetaNew.copy()
                variancecapofError = (SSENew/(N-n))
                sdval = np.sqrt(variancecapofError)
                covariancethetacap = variancecapofError*(LA.inv(A))
                covDiag = np.diag(covariancethetacap)
                print(f"Estimated params are:\n {thetacap}")
                # print(f"True params are: {AN[1:na] + BN[1:nb]}")
                print(f"Standard Deviation of the Error is : {sdval}")
                numMARoots = np.roots(thetacap[na:].ravel())
                denARRoots = np.roots(thetacap[:na].ravel())

                for b in range(0,len(numMARoots)):
                    print(f"Roots of the numerators is/are -{numMARoots[b]}")

                for a in range(0,len(denARRoots)):
                    print(f"Roots of the denominator is/are -
{denARRoots[a]}")

                confIntna = thetacap[:na]
                confIntnb = thetacap[na:]
                for l in range(0,na):
                    print(f"The Confidence interval for a{l+1} is -
{confIntna[[l]] - 2*(np.sqrt(covDiag[[l]]))} < a{l+1} < {confIntna[[l]] +
```

```
2*(np.sqrt(covDiag[[l]]))} ")

                for m in range(0,nb):
                    print(f"The Confidence interval for b{m+1} is -
{confIntnb[[m]] - 2*(np.sqrt(covDiag[[m]]))} < b{m+1} < {confIntnb[[m]] +
2*(np.sqrt(covDiag[[m]]))} ")

                print(f"The Covariance Matrix is:\n {covariancethetacap}")

                print(f"mu greater than mumax")
                plt.plot(SSEcompiled, label='S.S.E.', color='purple')
                plt.grid()
                plt.legend()
                plt.margins(x=0)
                plt.xticks(np.arange(0, numIter + 1, step=1))
                plt.title('Sum of Squared Errors ')
                plt.xlabel('Iterations')
                plt.ylabel('Magnitude of Error')
                plt.figure(figsize=(16, 8))
                plt.show()
                return "Mu >Mu_Max"

            delta_theta = np.dot(LA.inv(A + (mu * np.identity(n))), g)
            thetaNew = theta + delta_theta

            MAPARAMNUMthetaNew = [0] * max(na, nb)
            ARPARAMDENthetaNew = [0] * max(na, nb)

            ARPARAMDENthetaNew[:na] = thetaNew[:na]
            MAPARAMNUMthetaNew[:nb] = thetaNew[na:]

            num = np.insert(MAPARAMNUMthetaNew, 0, 1).tolist()
            den = np.insert(ARPARAMDENthetaNew, 0, 1).tolist()

            system = (den, num, 1)  # den,num,1 because we are generating
synthetic error
            _, eNewthetaNew = signal.dlsim(system, y)

            SSENew = np.dot(eNewthetaNew.T, eNewthetaNew)

        numIter+=1
        if numIter>MAX:
            thetacap = thetaNew.copy()
            variancecapofError = (SSENew / (N - n))
            covariancethetacap = variancecapofError * (LA.inv(A))
            covDiag = np.diag(covariancethetacap)
            print(f"Estimated params are:\n {thetacap}")
            # print(f"True params are: {AN[1:na] + BN[1:nb]}")#--------------
---
            print(f"Variance of the Error is : {variancecapofError}")
            numMARoots = np.roots(thetacap[na:].ravel())
            denARRoots = np.roots(thetacap[:na].ravel())

            for b in range(0, len(numMARoots)):
                print(f"Roots of the numerator is/are -{numMARoots[b]}")

            for a in range(0, len(denARRoots)):
```

```
                print(f"Roots of the denominator is/ are -{denARRoots[a]}")

            confIntna = thetacap[:na]
            confIntnb = thetacap[na:]
            for l in range(0, na):
                print(
                    f"The Confidence interval for a{l + 1} is -
{confIntna[[l]] - 2 * (np.sqrt(covDiag[[l]]))} < a{l + 1} < {confIntna[[l]] +
2 * (np.sqrt(covDiag[[l]]))} ")

            for m in range(0, nb):
                print(
                    f"The Confidence interval for b{m + 1} is -
{confIntnb[[m]] - 2 * (np.sqrt(covDiag[[m]]))} < b{m + 1} < {confIntnb[m] + 2
* (np.sqrt(covDiag[[m]]))} ")

            print(f"The Covariance Matrix is:\n {covariancethetacap}")
            print(f"Iteration higher than MAX")

            plt.plot(SSEcompiled, label='S.S.E.', color='purple')
            plt.grid()
            plt.legend()
            plt.margins(x=0)
            plt.xticks(np.arange(0, numIter + 1, step=1))
            plt.title('Sum of Squared Errors ')
            plt.xlabel('Iterations')
            plt.ylabel('Magnitude of Error')
            plt.figure(figsize=(16, 8))
            plt.show()

            return "NumIter > MAX"

        theta = thetaNew.copy()
def ACFPlot(array, lag, title):
    lag = list(range(-lag, lag + 1))
    numerator = 0
    denominator = 0
    mean = np.mean(array)

    for i in range(len(array)):
        denominator = (array[i] - mean) ** 2 + denominator  # cumalative sum

    totalacf = []
    for j in lag:
        j = abs(j)
        numerator = 0
        for k in range(j, len(array)):
            numerator = (array[k] - mean) * (array[k - j] - mean) + numerator
# num cumalative sum
        acfonelag = numerator / denominator
        totalacf.append(acfonelag)  # appending those sums in array

    plt.stem(lag, totalacf, markerfmt='ro', basefmt='b-',
use_line_collection=True)
    plt.axhspan(1.96 / len(array) ** 0.5, -1.96 / len(array) ** 0.5,
alpha=0.2, color='blue')
    plt.axhline(y=0, color='black', linestyle='--')
```

```
    plt.title('Autocorrelation Function of ' + title)
    plt.xlabel('Lags')
    plt.ylabel('Magnitude')
    plt.figure(figsize=(16, 8))
    plt.show()
    return totalacf


df.iloc[:,0:-1].describe()
df.iloc[:,-1].describe()


#Raw Data
subsetdf = df[['Power(kW)']]
ACF_PACF_Plot(subsetdf,200)

#Corr plot

X,y = df.iloc[:,1:-1],df[['Power(kW)']]

featureforcorr = SelectKBest(score_func = f_classif, k=10)
X1 = featureforcorr.fit_transform(X,y)
listfeatureforcorr = []
for columnname, true in zip(df.iloc[:,1:-1].columns,
featureforcorr.get_support()):
    if true:
        listfeatureforcorr.append(columnname)


print("The chosen features for plotting heatmap are-\n ", listfeatureforcorr)

corrdf = df.loc[:,listfeatureforcorr]
corrdf = pd.concat([corrdf,subsetdf],axis=1)
correlationplot = corrdf.corr(method='pearson', numeric_only=True)
sns.set(font_scale=5)
sns.set(rc={'figure.figsize':(20,20)})
sns.heatmap(correlationplot, cmap='coolwarm',annot=True)
plt.title("Heatmap of Data")
plt.tight_layout()
plt.show()


# model=pm.auto_arima(subsetdf[:5000],start_p=0,d=0,start_q=0,
#           max_p=15,max_q=5,
#           start_P=0,D=0, start_Q=0,
#           max_P=8, max_Q=8,
#           m=24, seasonal=True, stationary=True,
#           error_action='warn',trace=True,
#           supress_warnings=True,stepwise=True,
#           random_state=20,n_fits=100)

#Train test split
X_train, X_test, y_train, y_test = train_test_split(X,y,
                                    random_state=6313,
                                    test_size=0.20,
                                    shuffle=False)
```

```
print("Size of independent variable's training set is -", X_train.shape)
print("Size of dependent variables's training set is -", y_train.shape)
print("Size of independent variable's test set is -", X_test.shape)
print("Size of dependent variables's test set is -", y_test.shape)

subsetdf = y_train.copy(deep = true)
#Stationarity test
Cal_rolling_mean_var(subsetdf)
ADF_Cal(subsetdf)
kpss_test(subsetdf)
ACF_PACF_Plot(subsetdf,200)

#STL decomposition complete process
from statsmodels.tsa.seasonal import STL
#RAW DATA
STL = STL(subsetdf, period=24)
res = STL.fit()
fig,ax = plt.subplots(nrows=4,ncols=1,figsize = (16,8))
res.observed.plot(ax =ax[0])
ax[0].set_ylabel('Observed Value')
ax[0].margins(x=0)
res.trend.plot(ax =ax[1])
ax[1].set_ylabel('Trend')
ax[1].margins(x=0)
res.seasonal.plot(ax =ax[2])
ax[2].set_ylabel('Seasonal')
ax[2].margins(x=0)
res.resid.plot(ax =ax[3])
ax[3].set_ylabel('Residual')
ax[3].margins(x=0)
plt.tight_layout()
plt.show()

T = res.trend
S = res.seasonal
R = res.resid
plt.figure(figsize=(16,8))
plt.plot(T,label = 'trend')
plt.plot(S,label = 'Seasonal')
plt.plot(R,label = 'residuals')
plt.plot(subsetdf, label = 'original data')
plt.legend()
plt.title("STL Decomposition Combined Plot Power Magnitude v/s Samples")
plt.xticks(rotation = 45)
plt.grid()
plt.xlabel("Samples")
plt.ylabel("Power(kW)")
plt.tight_layout()
plt.margins(x=0)
plt.show()

num = np.var(R)
deno = np.var(T+R)
strengthofTrendFt =max(0,(1- (num/deno)))
print("The strength of trend for this data set is - ",
np.round(strengthofTrendFt,4))
num = np.var(R)
```

```
deno = np.var(R+S)
strengthofSeasonalityFs =max(0,(1- (num/deno)))
print("The strength of seasonality for this data set is - ",
np.round(strengthofSeasonalityFs,4))


# #We find SOT, SOS high even though the adf kpss and rolling mean var say
otherwise so we perform one order non seasonal diff
#TRANSFORMED DATA 1
subsetdf = norderdiff(subsetdf,'Power(kW)',1)
subsetdf = subsetdf.dropna()
subsetdf = subsetdf.reset_index()
subsetdf = subsetdf.iloc[:,-1]
subsetdf = pd.DataFrame(subsetdf)
Cal_rolling_mean_var(subsetdf)
ADF_Cal(subsetdf)
kpss_test(subsetdf)
ACF_PACF_Plot(subsetdf,200)

#Then we check STL decomposition again
from statsmodels.tsa.seasonal import STL
STL = STL(subsetdf, period=24)
res = STL.fit()
fig,ax = plt.subplots(nrows=4,ncols=1,figsize = (16,8))
res.observed.plot(ax =ax[0])
ax[0].set_ylabel('Observed Value')
ax[0].margins(x=0)
res.trend.plot(ax =ax[1])
ax[1].set_ylabel('Trend')
ax[1].margins(x=0)
res.seasonal.plot(ax =ax[2])
ax[2].set_ylabel('Seasonal')
ax[2].margins(x=0)
res.resid.plot(ax =ax[3])
ax[3].set_ylabel('Residual')
ax[3].margins(x=0)
plt.tight_layout()
plt.show()

T = res.trend
S = res.seasonal
R = res.resid

plt.figure(figsize=(16,8))
plt.plot(T,label = 'trend')
plt.plot(S,label = 'Seasonal')
plt.plot(R,label = 'residuals')
plt.plot(subsetdf, label = 'original data')
plt.legend()
plt.xticks(rotation = 45)
plt.grid()
plt.title("STL Decomposition Combined Plot Power Magnitude v/s Samples")
plt.xlabel("Samples")
plt.ylabel("Power(kW)")
plt.tight_layout()
plt.margins(x=0)
plt.show()
```

```
num = np.var(R)
deno = np.var(T+R)
strengthofTrendFt =max(0,(1- (num/deno)))
print("The strength of trend for this data set is - ",
np.round(strengthofTrendFt,4))
num = np.var(R)
deno = np.var(R+S)
strengthofSeasonalityFs =max(0,(1- (num/deno)))
print("The strength of seasonality for this data set is - ",
np.round(strengthofSeasonalityFs,4))
seasonAdj = subsetdf['Power(kW)_diff_1']-S
plt.figure(figsize=(16,8))
plt.plot(subsetdf, label = 'Original data', color = 'green')
plt.plot(seasonAdj, label = 'Seasonally Adjusted Data', color = 'brown')
plt.grid()
plt.legend()
plt.xlabel("Samples")
plt.ylabel("Power(kW)")
plt.title("Original data v/s Seasonally Adjusted Data")
plt.tight_layout()
plt.show()
trendAdj = subsetdf['Power(kW)_diff_1']-T
plt.figure(figsize=(16,8))
plt.plot(subsetdf, label = 'Original data', color = 'green')
plt.plot(trendAdj, label = 'Trend Adjusted Data', color = 'brown')
plt.grid()
plt.legend()
plt.xlabel("Samples")
plt.ylabel("Power(kW)")
plt.title("Original data v/s Trend Adjusted Data")
plt.tight_layout()
plt.show()


#Holt Winters
holtwintersubset = df[['Power(kW)']]
holtwintersubset.index = date
yt, yf = train_test_split(holtwintersubset, shuffle= False, test_size=0.2)
holtt =
ets.ExponentialSmoothing(yt,trend='add',damped_trend=True,seasonal='add',
seasonal_periods=24).fit()
holtf = holtt.forecast(steps=len(yf))
holtf = pd.DataFrame(holtf).set_index(yf.index)
holtwinterRMSE =
np.sqrt(np.square(np.subtract(yf.values,np.ndarray.flatten(holtf.values))).me
an())
print(f'Root Mean square error for Holt-Winter method is
{holtwinterRMSE:.2f}')
fig, ax = plt.subplots(figsize = (16,8))
ax.plot(df['Timestamp'][:len(yt)],yt,label= "Train Data")
ax.plot(df['Timestamp'][len(yt):],yf,label= "Test Data")
ax.plot(df['Timestamp'][len(yt):],holtf,label= "Holt-Winter")
plt.legend(loc='upper left')
plt.title(f'Holt-Winter- MSE = {holtwinterRMSE:.2f}')
plt.xlabel('Time')
plt.ylabel('Power(kW)')
```

```
plt.grid()
plt.show()


#Feature Selection/Elimination
X,y = df.iloc[:,1:-1],df[['Power(kW)']]
vif_data = pd.DataFrame()
vif_data["feature"] = X.columns
vif_data["VIF"] = [variance_inflation_factor(X.values, i)
                   for i in range(len(X.columns))]

print(np.round(vif_data,4))
#https://www.geeksforgeeks.org/detecting-multicollinearity-with-vif-python/

matH = (X.transpose())@X
s, d, v = np.linalg.svd(matH)
print("SingularValues = ", np.round(d,4))

condNo = LA.cond(X)
print("Condition Number = ", np.round(condNo,4) )
#Scaler Transformation
scaler = StandardScaler()
XtrainScaled = scaler.fit_transform(X_train)
XtestScaled = scaler.transform(X_test)
print("Transformation successfull, fit transform for train sets and then
transform for test set")
XtrainScaled = sm.add_constant(XtrainScaled)
xcols = X_test.columns.tolist()
xcols = ['Constant']+xcols
XtrainScaleddf = pd.DataFrame(XtrainScaled, columns=xcols)
#https://statisticsbyjim.com/regression/interpret-constant-y-intercept-
regression/#:~:text=The%20reason%20I%20just%20discussed,the%20constant%20to%2
0equal%20zero.



model = sm.OLS(y_train,XtrainScaleddf).fit()
print(model.params)
print(model.summary())

#dropping features
XtrainScaleddf2 = XtrainScaleddf.drop(columns=["Blade-3 Set Value_Degree"])
model2 = sm.OLS(y_train,XtrainScaleddf2).fit()
print(model2.summary())

XtrainScaleddf3 = XtrainScaleddf2.drop(columns=["Temperature Battery Box-2"])
model3 = sm.OLS(y_train,XtrainScaleddf3).fit()
print(model3.summary())


XtrainScaleddf4 = XtrainScaleddf3.drop(columns=["Temperature Axis Box-1"])
model4 = sm.OLS(y_train,XtrainScaleddf4).fit()
print(model4.summary())

XtrainScaleddf5 = XtrainScaleddf4.drop(columns=["Temperature Ambient"])
model5 = sm.OLS(y_train,XtrainScaleddf5).fit()
print(model5.summary())
```

```
XtrainScaleddf6 =
XtrainScaleddf5.drop(columns=["Gearbox_Distributor_Temperature"])
model6 = sm.OLS(y_train,XtrainScaleddf6).fit()
print(model6.summary())

XtrainScaleddf7 = XtrainScaleddf6.drop(columns=["Converter Control Unit
Voltage"])
model7 = sm.OLS(y_train,XtrainScaleddf7).fit()
print(model7.summary())

XtrainScaleddf8 = XtrainScaleddf7.drop(columns=["Power Factor"])
model8 = sm.OLS(y_train,XtrainScaleddf8).fit()
print(model8.summary())

XtrainScaleddf9 = XtrainScaleddf8.drop(columns=["Temperature Axis Box-3"])
model9 = sm.OLS(y_train,XtrainScaleddf9).fit()
print(model9.summary())

XtrainScaleddf10 = XtrainScaleddf9.drop(columns=["Hydraulic Prepressure"])
model10 = sm.OLS(y_train,XtrainScaleddf10).fit()
print(model10.summary())

XtrainScaleddf11 = XtrainScaleddf10.drop(columns=["Proxy Sensor_Degree-135"])
model11 = sm.OLS(y_train,XtrainScaleddf11).fit()
print(model11.summary())

XtrainScaleddf12 = XtrainScaleddf11.drop(columns=["Internal Power Limit"])
model12 = sm.OLS(y_train,XtrainScaleddf12).fit()
print(model12.summary())

XtrainScaleddf13 = XtrainScaleddf12.drop(columns=["Blade-1 Set
Value_Degree"])
model13 = sm.OLS(y_train,XtrainScaleddf13).fit()
print(model13.summary())

XtrainScaleddf14 = XtrainScaleddf13.drop(columns=["Pitch Offset Tower
Feedback"])
model14 = sm.OLS(y_train,XtrainScaleddf14).fit()
print(model14.summary())

XtrainScaleddf15 = XtrainScaleddf14.drop(columns=["Moment Q Filltered"])
model15 = sm.OLS(y_train,XtrainScaleddf15).fit()
print(model15.summary())

XtrainScaleddf16 = XtrainScaleddf15.drop(columns=["Temperature Tower Base"])
model16 = sm.OLS(y_train,XtrainScaleddf16).fit()
print(model16.summary())

XtrainScaleddf17 =
XtrainScaleddf16.drop(columns=["Gearbox_T3_Intermediate_Speed_Shaft_Temperatu
re"])
model17 = sm.OLS(y_train,XtrainScaleddf17).fit()
print(model17.summary())

XtrainScaleddf18 = XtrainScaleddf17.drop(columns=["Circuit Breaker cut-ins"])
```

```
model18 = sm.OLS(y_train,XtrainScaleddf18).fit()
print(model18.summary())

XtrainScaleddf19 = XtrainScaleddf18.drop(columns=["Scope CH 4"])
model19 = sm.OLS(y_train,XtrainScaleddf19).fit()
print(model19.summary())


XtrainScaleddf20 = XtrainScaleddf19.drop(columns=["Moment Q Direction"])
model20 = sm.OLS(y_train,XtrainScaleddf20).fit()
print(model20.summary())

XtrainScaleddf21 = XtrainScaleddf20.drop(columns=["Tower Deflection"])
model21 = sm.OLS(y_train,XtrainScaleddf21).fit()
print(model21.summary())


XtrainScaleddf22 = XtrainScaleddf21.drop(columns=["Blade-3 Actual
Value_Angle-A"])
model22 = sm.OLS(y_train,XtrainScaleddf22).fit()
print(model22.summary())

XtrainScaleddf23 = XtrainScaleddf22.drop(columns=["Torque Offset Tower
Feedback"])
model23 = sm.OLS(y_train,XtrainScaleddf23).fit()
print(model23.summary())


XtrainScaleddf24 = XtrainScaleddf23.drop(columns=["Pitch Offset-2 Asymmetric
Load Controller"])
model24 = sm.OLS(y_train,XtrainScaleddf24).fit()
print(model24.summary())

XtrainScaleddf25 = XtrainScaleddf24.drop(columns=["Line Frequency"])
model25 = sm.OLS(y_train,XtrainScaleddf25).fit()
print(model25.summary())

XtrainScaleddf26 = XtrainScaleddf25.drop(columns=["Wind Deviation 10
seconds"])
model26 = sm.OLS(y_train,XtrainScaleddf26).fit()
print(model26.summary())

XtrainScaleddf27 = XtrainScaleddf26.drop(columns=["Nacelle Position_Degree"])
model27 = sm.OLS(y_train,XtrainScaleddf27).fit()
print(model27.summary())

XtrainScaleddf28 = XtrainScaleddf27.drop(columns=["Temperature Heat Exchanger
Converter Control Unit"])
model28 = sm.OLS(y_train,XtrainScaleddf28).fit()
print(model28.summary())

XtrainScaleddf29 = XtrainScaleddf28.drop(columns=["Blade-3 Actual
Value_Angle-B"])
model29 = sm.OLS(y_train,XtrainScaleddf29).fit()
print(model29.summary())
```

```
XtrainScaleddf30 = XtrainScaleddf29.drop(columns=["Temperature Axis Box-2"])
model30 = sm.OLS(y_train,XtrainScaleddf30).fit()
print(model30.summary())

XtrainScaleddf31 =
XtrainScaleddf30.drop(columns=["Gearbox_T3_High_Speed_Shaft_Temperature"])
model31 = sm.OLS(y_train,XtrainScaleddf31).fit()
print(model31.summary())

XtrainScaleddf32 = XtrainScaleddf31.drop(columns=["Tower Acceleration
Lateral"])
model32 = sm.OLS(y_train,XtrainScaleddf32).fit()
print(model32.summary())


XtrainScaleddf33 = XtrainScaleddf32.drop(columns=["Gearbox_Oil-
1_Temperature"])
model33 = sm.OLS(y_train,XtrainScaleddf33).fit()
print(model33.summary())


XtrainScaleddf34 = XtrainScaleddf33.drop(columns=["Voltage B-N"])
model34 = sm.OLS(y_train,XtrainScaleddf34).fit()
print(model34.summary())


XtrainScaleddf35 = XtrainScaleddf34.drop(columns=["Temperature_Nacelle"])
model35 = sm.OLS(y_train,XtrainScaleddf35).fit()
print(model35.summary())


XtrainScaleddf36 = XtrainScaleddf35.drop(columns=["Blade-2 Actual
Value_Angle-B"])
model36 = sm.OLS(y_train,XtrainScaleddf36).fit()
print(model36.summary())


XtrainScaleddf37 = XtrainScaleddf36.drop(columns=["State and Fault"])
model37 = sm.OLS(y_train,XtrainScaleddf37).fit()
print(model37.summary())

XtrainScaleddf38 = XtrainScaleddf37.drop(columns=["Temperature Trafo-3"])
model38 = sm.OLS(y_train,XtrainScaleddf38).fit()
print(model38.summary())


XtrainScaleddf39 = XtrainScaleddf38.drop(columns=["Tower Accelaration Normal
Raw"])
model39 = sm.OLS(y_train,XtrainScaleddf39).fit()
print(model39.summary())

XtrainScaleddf40 = XtrainScaleddf39.drop(columns=["Blade-2 Set
Value_Degree"])
model40 = sm.OLS(y_train,XtrainScaleddf40).fit()
print(model40.summary())
```

```
XtrainScaleddf41 =
XtrainScaleddf40.drop(columns=["Gearbox_T1_Intermediate_Speed_Shaft_Temperatu
re"])
model41 = sm.OLS(y_train,XtrainScaleddf41).fit()
print(model41.summary())


XtrainScaleddf42 = XtrainScaleddf41.drop(columns=["Voltage A-N"])
model42 = sm.OLS(y_train,XtrainScaleddf42).fit()
print(model42.summary())


XtrainScaleddf43 = XtrainScaleddf42.drop(columns=["Temperature Battery Box-
3"])
model43 = sm.OLS(y_train,XtrainScaleddf43).fit()
print(model43.summary())

XtrainScaleddf44 =
XtrainScaleddf43.drop(columns=["Gearbox_T1_High_Speed_Shaft_Temperature"])
model44 = sm.OLS(y_train,XtrainScaleddf44).fit()
print(model44.summary())


XtrainScaleddf45 = XtrainScaleddf44.drop(columns=["Operating State"])
model45 = sm.OLS(y_train,XtrainScaleddf45).fit()
print(model45.summary())

XtrainScaleddf46 = XtrainScaleddf45.drop(columns=["Proxy Sensor_Degree-315"])
model46 = sm.OLS(y_train,XtrainScaleddf46).fit()
print(model46.summary())


XtrainScaleddf47 = XtrainScaleddf46.drop(columns=["Pitch Offset-1 Asymmetric
Load Controller"])
model47 = sm.OLS(y_train,XtrainScaleddf47).fit()
print(model47.summary())


XtrainScaleddf48 = XtrainScaleddf47.drop(columns=["Blade-1 Actual
Value_Angle-A"])
model48 = sm.OLS(y_train,XtrainScaleddf48).fit()
print(model48.summary())

XtrainScaleddf49 = XtrainScaleddf48.drop(columns=["Turbine State"])
model49 = sm.OLS(y_train,XtrainScaleddf49).fit()
print(model49.summary())


XtrainScaleddf50 = XtrainScaleddf49.drop(columns=["Blade-2 Actual
Value_Angle-A"])
model50 = sm.OLS(y_train,XtrainScaleddf50).fit()
print(model50.summary())

XtrainScaleddf51 = XtrainScaleddf50.drop(columns=["External Power Limit"])
model51 = sm.OLS(y_train,XtrainScaleddf51).fit()
print(model51.summary())
```

```
XtrainScaleddf52 = XtrainScaleddf51.drop(columns=["Gearbox_Oil_Temperature"])
model52 = sm.OLS(y_train,XtrainScaleddf52).fit()
print(model52.summary())


XtrainScaleddf53 = XtrainScaleddf52.drop(columns=["Voltage C-N"])
model53 = sm.OLS(y_train,XtrainScaleddf53).fit()
print(model53.summary())

XtrainScaleddf54 = XtrainScaleddf53.drop(columns=["Proxy Sensor_Degree-45"])
model54 = sm.OLS(y_train,XtrainScaleddf54).fit()
print(model54.summary())


XtrainScaleddf55 = XtrainScaleddf54.drop(columns=["Temperature Bearing_A"])
model55 = sm.OLS(y_train,XtrainScaleddf55).fit()
print(model55.summary())

XtrainScaleddf56 = XtrainScaleddf55.drop(columns=["Pitch Demand
Baseline_Degree"])
model56 = sm.OLS(y_train,XtrainScaleddf56).fit()
print(model56.summary())


XtrainScaleddf57 = XtrainScaleddf56.drop(columns=["Gearbox_Oil-
2_Temperature"])
model57 = sm.OLS(y_train,XtrainScaleddf57).fit()
print(model57.summary())


XtrainScaleddf58 = XtrainScaleddf57.drop(columns=["Temperature Battery Box-
1"])
model58 = sm.OLS(y_train,XtrainScaleddf58).fit()
print(model58.summary())

XtrainScaleddf59 = XtrainScaleddf58.drop(columns=["Proxy Sensor_Degree-225"])
model59 = sm.OLS(y_train,XtrainScaleddf59).fit()
print(model59.summary())
#----------------------------------------------------------------------------
- removed torque by mistake added back and
# then removed moment d filtered now seeing double torque but removed in next
line

XtrainScaleddf60 = XtrainScaleddf59.drop(columns=["Moment D Filtered"])
model60 = sm.OLS(y_train,XtrainScaleddf60).fit()
print(model60.summary())

XtrainScaleddf61 = XtrainScaleddf60.drop(columns=["Torque"])
model61 = sm.OLS(y_train,XtrainScaleddf61).fit()
print(model61.summary())


XtrainScaleddf62 = XtrainScaleddf61.drop(columns=["Nacelle Revolution"])
model62 = sm.OLS(y_train,XtrainScaleddf62).fit()
print(model62.summary())
```

```
XtrainScaleddf63 = XtrainScaleddf62.drop(columns=["Tower Accelaration Lateral
Raw"])
model63 = sm.OLS(y_train,XtrainScaleddf63).fit()
print(model63.summary())



XtrainScaleddf64 = XtrainScaleddf63.drop(columns=["Pitch Offset-3 Asymmetric
Load Controller"])
model64 = sm.OLS(y_train,XtrainScaleddf64).fit()
print(model64.summary())

XtrainScaleddf65 = XtrainScaleddf64.drop(columns=["Temperature Shaft Bearing-
2"])
model65 = sm.OLS(y_train,XtrainScaleddf65).fit()
print(model65.summary())


XtrainScaleddf66 = XtrainScaleddf65.drop(columns=["Temperature Shaft Bearing-
1"])
model66 = sm.OLS(y_train,XtrainScaleddf66).fit()
print(model66.summary())

XtrainScaleddf67 = XtrainScaleddf66.drop(columns=["Temperature Gearbox
Bearing Hollow Shaft"])
model67 = sm.OLS(y_train,XtrainScaleddf67).fit()
print(model67.summary())

XtrainScaleddf68 = XtrainScaleddf67.drop(columns=["Temperature Trafo-2"])
model68 = sm.OLS(y_train,XtrainScaleddf68).fit()
print(model68.summary())


XtrainScaleddf69 = XtrainScaleddf68.drop(columns=["Angle Rotor Position"])
model69 = sm.OLS(y_train,XtrainScaleddf69).fit()
print(model69.summary())

XtrainScaleddf70 = XtrainScaleddf69.drop(columns=["Blade-1 Actual
Value_Angle-B"])
model70 = sm.OLS(y_train,XtrainScaleddf70).fit()
print(model70.summary())

#model70 is final model now we do all other analysis and prediction

XtestScaled = sm.add_constant(XtestScaled)
XtestScaleddf = pd.DataFrame(XtestScaled,index=X_test.index, columns=xcols)

XtestScaleddf = XtestScaleddf[['Constant','Tower Acceleration
Normal','Converter Control Unit Reactive Power',
                               'Reactive Power','Moment D Direction','N-set
1','Particle Counter','Wind Deviation 1 seconds']]

OLSPredict = model70.predict(XtestScaleddf)

plt.figure(figsize=(16,8))
plt.plot(df['Timestamp'][len(y_train):],y_test['Power(kW)'], 'r',
```

```
label="Original Test data ")
plt.plot(df['Timestamp'][len(y_train):], OLSPredict, 'b', label="Predicted
data")
plt.xlabel("Time")
plt.ylabel("Power(kW)")
plt.legend(loc = 'lower right')
plt.title("Test Data v/s One Step Ahead Prediction : OLS Model")
plt.tight_layout()
plt.show()

rmseOLS = np.sqrt(mean_squared_error(y_test, OLSPredict))
print("The root mean squared error of OLS Model is = ", np.round(rmseOLS,4))
print("The AIC and BIC scores of the model respectively are =
",np.round(model70.aic,4),"and ",np.round(model70.bic,4))
print("The R-Sqaured and Adjusted R-sqaured values of the model respectively
are = ",np.round(model70.rsquared,4),"and\n
",np.round(model70.rsquared_adj,4))
#COMPLETE T test
p_val_in_T_test = model70.pvalues
conf95 = 0.05
for i,var in enumerate(XtrainScaleddf70.columns):
      if p_val_in_T_test[i] < conf95:
            print(var," is statistically significant where p value of the " ,
var ," is ",np.round(p_val_in_T_test[i],4),"\n")
      else:
            print(var + " is not statistically significant\n")
#COMPLETE F TEST
F_Test_Stat = model70.f_pvalue
columns = ', '.join(XtrainScaleddf70.columns.tolist())
print("The F Test value for the model 70(Final OLS Model) for \nWind Turbine
Power Prediction with features", columns,"is-\n", np.round(F_Test_Stat,4))

acf = ACFPlot(model70.resid,48,"Residuals of OLS")
re = acf[(len(acf)//2):]
Q = len(y) * np.sum(np.square(re[1:]))
print("The Q value of the OLS Model Residuals is -", np.round(Q,4))

print("Mean of Residuals = ",np.mean(model70.resid),"and Variance of
residuals =",np.round(np.var(model70.resid),4))




#BASE MODELS
#AVERAGE METHOD
averagesubset = df[['Power(kW)']]
ytavg, yfavg = train_test_split(averagesubset, shuffle= False, test_size=0.2)
yhstep = np.mean(ytavg.values)
yhstep = len(yfavg)*[yhstep]
print("Average Method - h Step prediction =", yhstep)
yhstep = pd.DataFrame(yhstep, columns = ['Power(kW)'], index=yfavg.index)


plt.figure(figsize=(16,8))
plt.plot(df['Timestamp'][len(ytavg):],yfavg['Power(kW)'], 'r',
label="Original Test data ")
plt.plot(df['Timestamp'][len(ytavg):], yhstep['Power(kW)'], 'b',
```

```
label="Predicted data")
plt.xlabel("Time")
plt.ylabel("Power(kW)")
plt.legend(loc = 'lower right')
plt.title("Test Data v/s h-Step Ahead Prediction- AVERAGE METHOD")
plt.tight_layout()
plt.show()

rmseAVG = np.sqrt(mean_squared_error(yfavg, yhstep))
print("The root mean squared error of AVERAGE Model is = ",
np.round(rmseAVG,4))



#NAIVE METHOD
naivesubset = df[['Power(kW)']]
ytnaive, yfnaive = train_test_split(naivesubset, shuffle= False,
test_size=0.2)
ynaivehstep = ytnaive.values[-1]
ynaivehstep = len(yfnaive)*[ynaivehstep[0]]
print("Naive Method - h Step prediction =", ynaivehstep)
ynaivehstep = pd.DataFrame(ynaivehstep, columns = ['Power(kW)'],
index=yfnaive.index)


plt.figure(figsize=(16,8))
plt.plot(df['Timestamp'][len(ytnaive):],yfnaive['Power(kW)'], 'r',
label="Original Test data ")
plt.plot(df['Timestamp'][len(ytnaive):], ynaivehstep['Power(kW)'], 'b',
label="Predicted data")
plt.xlabel("Time")
plt.ylabel("Power(kW)")
plt.legend(loc = 'lower right')
plt.title("Test Data v/s h-Step Ahead Prediction: NAIVE METHOD")
plt.tight_layout()
plt.show()

rmseNAIVE = np.sqrt(mean_squared_error(yfnaive, ynaivehstep))
print("The root mean squared error of NAIVE Model is = ",
np.round(rmseNAIVE,4))





#DRIFT METHOD
driftsubset = df[['Power(kW)']]
ytdrift, yfdrift = train_test_split(driftsubset, shuffle= False,
test_size=0.2)
ydrifthstep =[]
for i in range(1,len(yfdrift)):
    ydrifthstep.append((ytdrift.values[-1]+i*((ytdrift.values[-1]-
ytdrift.values[0])/(len(ytdrift.values)-1))))

ydrifthstep = [i[0] for i in ydrifthstep]

print("Drift Method - h Step prediction =", ydrifthstep)
```

```
yfdrift = yfdrift.iloc[1:,]
ydrifthstep = pd.DataFrame(ydrifthstep, columns = ['Power(kW)'],
index=yfdrift.index)


plt.figure(figsize=(16,8))
plt.plot(df['Timestamp'][len(ytdrift)+1:],yfdrift['Power(kW)'], 'r',
label="Original Test data")
plt.plot(df['Timestamp'][len(ytdrift)+1:], ydrifthstep['Power(kW)'], 'b',
label="Predicted data")
plt.xlabel("Time")
plt.ylabel("Power(kW)")
plt.legend(loc = 'lower right')
plt.title("Test Data v/s h-Step Ahead Prediction: DRIFT METHOD")
plt.tight_layout()
plt.show()

rmseDRIFT = np.sqrt(mean_squared_error(yfdrift, ydrifthstep))
print("The root mean squared error of DRIFT Model is = ",
np.round(rmseDRIFT,4))



#SES METHOD
SeSsubset = df[['Power(kW)']]
ytses, yfses = train_test_split(SeSsubset, shuffle= False, test_size=0.2)
alpha = 0.5
yses1step = [ytses.iloc[0].values.tolist()[0]]
for i in range(len(ytses)-1):
    yses1step.append(ytses.iloc[i].values.tolist()[0]*alpha+ (1-
alpha)*yses1step[i])

yseshstep = ytses.iloc[-1].values.tolist()[0]*alpha + (1-alpha)*yses1step[-1]
yseshstep = len(yfses)*[yseshstep]
print("Simple Exponential Smoothing - h Step prediction =", yseshstep)


yseshstep = pd.DataFrame(yseshstep, columns = ['Power(kW)'],
index=yfses.index)


plt.figure(figsize=(16,8))
plt.plot(df['Timestamp'][len(ytses):],yfses['Power(kW)'], 'r',
label="Original Test data ")
plt.plot(df['Timestamp'][len(ytses):],yseshstep['Power(kW)'], 'b',
label="Predicted data")
plt.xlabel("Time")
plt.ylabel("Power(kW)")
plt.legend(loc = 'lower right')
plt.title("Test Data v/s h-Step Ahead Prediction: Simple Exponential
Smoothing Method")
plt.tight_layout()
plt.show()

rmseSES = np.sqrt(mean_squared_error(yfses, yseshstep))
print("The root mean squared error of SES Model is = ", np.round(rmseSES,4))
```

```
#TRANSFORMED DATA 2 - 24 diff seasonal on top of 1 diff for GPAC and LM
Algorithm use later
#subsetdf = norderdiff(subsetdf,'Power(kW)',24)
subsetdf = norderdiff(subsetdf,'Power(kW)_diff_1',24)
subsetdf = subsetdf.dropna()
subsetdf = subsetdf.reset_index()
subsetdf = subsetdf.iloc[:,-1]
subsetdf = pd.DataFrame(subsetdf)


#GPAC
ry = sm.tsa.stattools.acf(subsetdf, nlags=100, fft=False)
ry1 = ry[::-1]
ry2 = np.concatenate((np.reshape(ry1, 101), ry[1:]))
ry3 = pd.Series(ry2)
calGPAC(ry3,30,30)

ACF_PACF_Plot(subsetdf,200)

theta = np.zeros((24))

paramEst(subsetdf,theta,len(subsetdf),0,24)
paramEst(subsetdf,theta,len(subsetdf),24,0)


#ARIMAmodel = ARIMA(subsetdftrain, order=(0, 0, 1), trend='n').fit() # d and
D not to be filled when your data is differenced
SARIMAmodel = sm.tsa.SARIMAX(y_train,
order=(0,1,1),seasonal_order=(0,1,1,24), trend='n').fit()
print(SARIMAmodel.summary())
y_model_hat = SARIMAmodel.predict(start = 1, end = len(y_train)-1)#----------
--param match but value shit in prediction
# y_train_forresid = y_train["Power(kW)"].iloc[1:].squeeze()
# y_model_hat.index = y_train_forresid.index
res_eSARIMAX = (y_train['Power(kW)'].iloc[1:])- y_model_hat
acf = ACFPlot(res_eSARIMAX.values,200,'residuals')
acf = pd.Series(acf)
calGPAC(acf,30,30)

y_hat_h_step = SARIMAmodel.forecast(steps=(len(y_test)))
y_hat_h_step.index = y_test.index

fore_errorSARIMAX = y_test['Power(kW)'] - y_hat_h_step

varResidSARIMAX = SARIMAmodel.resid.var()
varforecastSARIMAX = np.var(fore_errorSARIMAX)

print("variance of residual error is =",np.round(varResidSARIMAX,4))
print("variance of forecast error is =",np.round(varforecastSARIMAX,4))
re = acf[(len(acf)//2):]
Q = len(y) * np.sum(np.square(re[1:]))
DOF = 48 - 0 - 1
alfa = 0.01
chi_critical = chi2.ppf(1 - alfa, DOF)
```

```
print('Chi critical:', chi_critical)
print('Q Value:', Q)
print('Alfa value for 99% accuracy:', alfa)
if Q < chi_critical:
    print("The residual is white ")
else:
    print("The residual is NOT white ")

plt.figure(figsize=(16,8))
plt.plot(df['Timestamp'][:len(y_train)],y_train, 'r', label="Original Train
data ")
plt.plot(df['Timestamp'][:len(y_train)-1],y_model_hat, 'b', label="Model Fit
data")
plt.xlabel("Time")
plt.ylabel("Power(kW)")
plt.legend()
plt.title(" Train versus One Step Prediction")
plt.tight_layout()
plt.show()

plt.figure(figsize=(16,8))
plt.plot(df['Timestamp'][len(y_train):],y_test['Power(kW)'], 'r',
label="Original Test data ")
plt.plot(df['Timestamp'][len(y_train):], y_hat_h_step, 'b', label="Predicted
data")
plt.xlabel("Time")
plt.ylabel("Power(kW)")
plt.legend()
plt.title("Test Data v/s h-Step Ahead Prediction: SARIMA Model")
plt.tight_layout()
plt.show()

rmseSARIMAX = np.sqrt(mean_squared_error(y_test, y_hat_h_step))
print("The root mean squared error of SARIMA Model is = ",
np.round(rmseSARIMAX,4))


#LSTM
import tensorflow as tf
physical_devices = tf.config.list_physical_devices('GPU')
#tf.config.experimental.set_memory_growth(physical_devices[0],True)
CUDA_VISIBLE_DEVICES = '0,1'
from tensorflow.keras import Sequential
# from keras.layers import CuDNNLSTM
from tensorflow.keras.layers import Dense, LSTM, Dropout  # ,CuDNNLSTM
from tensorflow.keras.preprocessing.sequence import TimeseriesGenerator
from tensorflow.keras.optimizers import Adam


scaler = StandardScaler()
scaler = scaler.fit(df.iloc[:,1:])
dfscaled = scaler.transform(df.iloc[:,1:])

trainsize = int(len(df)*0.80)
testsize = int(len(df)*0.20)

traindata = dfscaled[:trainsize,:]
```

```
subsetdflen = len(df[['Power(kW)']])
npast = subsetdflen-trainsize
time_step = 24 # because of seasonality pattern in my data
Xtrain, ytrain = [],[]

for i in range(npast, len(traindata)):
    Xtrain.append(traindata[i - time_step:i, 0:traindata.shape[1]-1])
    ytrain.append(traindata[i, traindata.shape[1]-1])

train_X = np.array(Xtrain)
train_y = np.array(ytrain)


model = Sequential()
model.add(LSTM(64, activation="relu", input_shape=(train_X.shape[1],
train_X.shape[2]), return_sequences=True))
model.add(LSTM(50, return_sequences=False))
model.add(Dropout(0.2))
model.add(Dense(1))
model.compile(optimizer='adam', loss='mse')
model.summary()
history = model.fit(train_X, train_y,  batch_size=32, validation_split=.1,
epochs=30, verbose=1)
plt.figure()
plt.plot(history.history['loss'], 'r', label='Training loss')
plt.plot(history.history['val_loss'], 'b', label='Validation loss')
plt.title('Training and Validation Loss')
plt.legend()
plt.show()

testdata = dfscaled[trainsize - time_step:,:]
Xtest = []
ytest = df.iloc[trainsize:,-1]


for i in range(time_step, len(testdata)):
    Xtest.append(testdata[i-time_step:i, 0:traindata.shape[1]-1])

Xtest = np.array(Xtest)
predictions = model.predict(Xtest)
forecast_copies = np.repeat(predictions, traindata.shape[1], axis=-1)
predictions = scaler.inverse_transform(forecast_copies)[:, -1]


train = df.iloc[:trainsize]
valid = df.iloc[trainsize:]
valid['predictions'] = predictions

fig = plt.figure(figsize=(16, 8))
ax = fig.add_subplot(111)
ax.set_title('Power Output prediction of Wind turbine using LSTM')
ax.set_xlabel("Date", fontsize=18)
ax.set_ylabel('Power(kW)')
#ax.plot(df['Timestamp'][:len(train)],train['Power(kW)'], 'blue')
ax.plot(df['Timestamp'][len(train):], valid['Power(kW)'],  'orange')
ax.plot(df['Timestamp'][len(train):],valid['predictions'], 'Green')
ax.legend(["Train", "Val", "predictions"], loc="lower right", fontsize=18)
```

```
ax.grid()
plt.show()

rmseLSTM = np.sqrt(mean_squared_error(ytest, valid[['predictions']]))
print("The root mean squared error of long Short Term Memory Neural Network
is =\n ", np.round(rmseLSTM,4))
diffminmax = np.max(ytest) - np.min(ytest)
accuracy = (1-(rmseLSTM/diffminmax))*100
print("The accuracy of long Short Term Memory Neural Network is =\n ",
np.round(accuracy,4))


flist = ['Gearbox_T1_Intermediate_Speed_Shaft_Temperature',
 'Tower Acceleration Lateral',
 'Temperature Shaft Bearing-1',
 'Temperature Axis Box-1',
 'Voltage B-N',
 'Temperature Battery Box-3',
 'Internal Power Limit',
 'Temperature Ambient',
 'Pitch Offset-1 Asymmetric Load Controller',
 'Turbine State']

Xhailmarydf = X_train.loc[:,flist]
Xhailmarydftestingforaccuracy = X_test.loc[:,flist]
ytestingforRMSE = y_test.loc[:,'Power(kW)']




GBRModel =
GradientBoostingRegressor(n_estimators=150,max_depth=4,learning_rate = 0.2
,random_state=44)
GBRModel.fit(Xhailmarydf, y_train)
ypredgbr =GBRModel.predict(Xhailmarydftestingforaccuracy)
print('Gradient Boosting Regression Model Test Score/R2 value is = ' ,
np.round(GBRModel.score(Xhailmarydftestingforaccuracy, y_test),4))
print('The Adjusted R2 value for Gradient Boosting Regression is = ',
np.round(r2_score(ytestingforRMSE, ypredgbr),4))
RMSEGBR = np.sqrt(mean_squared_error(y_test, ypredgbr))
print('The RMSE for Gradient Boosting Regression Model is =',
np.round(RMSEGBR,4))

plt.figure(figsize=(16,8))
plt.plot(df['Timestamp'][len(y_train):],ytestingforRMSE, 'r', label="Original
Test data ")
plt.plot(df['Timestamp'][len(y_train):], ypredgbr, 'b', label="Predicted
data")
plt.xlabel("Time")
plt.ylabel("Power(kW)")
plt.legend()
plt.title("Test Data v/s h-Step Ahead Prediction: Gradient Boosting
Regression Model")
plt.tight_layout()
plt.show()
```

```
#https://www.makeareadme.com/
#https://towardsdatascience.com/all-you-need-to-know-about-gradient-boosting-algorithm-part-1-regression-2520a34a502
```