Appendix

Data Preprocessing

```
#%% ------------------------------ Import Packages and Libraries ---------
------------------------------------------------------------------
import numpy as np
import pandas as pd

#%% ------------------------------------ Load dataset --------------------
-----------------------------------------------
# read the xlsx format of the dataset
df = pd.read_excel("../data/AirQualityUCI.xlsx")


#%% ------------------------------------ Identifying missing data --------
--------------------------------------------------
# define a function to check for missing data
def nan_checker(df):
    """
    Parameters
    ----------
    df : dataframe

    Returns
    ----------
    The dataframe of variables with missing data,
    the proportion of missing data and dtype of variable
    """

    # Get the dataframe of variables with NaN, their proportion of NaN and
dtype
    df_nan = pd.DataFrame([[var, df[var].isna().sum() / df.shape[0],
df[var].dtype]
                           for var in df.columns if df[var].isna().sum() >
0],
                          columns=['var', 'proportion', 'dtype'])

    # Sort df_nan in accending order of the proportion of NaN
    df_nan = df_nan.sort_values(by='proportion',
ascending=False).reset_index(drop=True)

    return df_nan

# since missing values are replaced by -200, we need to convert them back
into NaN
df.replace(to_replace= -200, value= np.NaN, inplace= True)

# checking for missing data in dataset
df_nan = nan_checker(df)
print(df_nan)
```

```
#%% ------------------------------------- Remove missing data ------------
----------------------------------------------
# check for numerical variables that the majority of data is missing
(proportion of NaN > 80%)
df_rm = df_nan[(df_nan['dtype']== 'float64') & (df_nan['proportion'] >
0.8)].reset_index(drop=True)
print(df_rm)

# remove numerical variables with a lot of missing data since they do not
contribute to the prediction
df.drop(df_rm['var'], axis= 1, inplace= True)

# remaining variables with missing data
df_miss = df_nan[-df_nan['var'].isin(df_rm['var'])].reset_index(drop=True)

#%% ------------------------------------ Impute missing data -------------
-----------------------------------------------
# we fill missing data using the average of available values in a day
for var in df_miss['var']:
    df[var] = df.groupby("Date")[var].transform(lambda x:
x.fillna(x.mean()))

# check if there are still missing values since there may be no data
recorded in a day
print(nan_checker(df))

# There are still missing values due to no records in a particular date
# Assume that the current date data is related to the previous and next
date data
# We fill NaN of current date using the average of two closest available
data

for var in df_miss['var']:
    df[var] = (df[var].fillna(method='ffill', inplace = False) +
df[var].fillna(method='bfill', inplace = False))/2

# recheck for missing data
print(nan_checker(df))

#%% ------------------------------------- Handling Datetime Variable -----
------------------------------------------------
# we combine the hour and the date into 1 column
hr = "00:00:00"
for i in range(len(df)):
    # add hour to date
    df.loc[i,"Date"] = str(df.loc[i,"Date"]).replace(hr,
str(df.loc[i,"Time"]))
df["Date"] = pd.to_datetime(df["Date"]) # convert to datetime variable
df.drop('Time', axis=1, inplace=True) # drop the column of hour data

#%% --------------------------------- Summarize data and save preprocessed
version ------------------------------------------
# print the dimension of df_train
```

```
#print(pd.DataFrame([[df.shape[0], df.shape[1]]], columns=['# rows', '#
columns']))
df.info() # recheck the data
df.to_csv(r'../data/Preprocessed_AirQuality.csv', index = False) # save
the preprocessed data
```

Data Exploration

```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import matplotlib.dates as mdates
from pandas.plotting import register_matplotlib_converters
register_matplotlib_converters()
from statsmodels.tsa.stattools import adfuller
from statsmodels.tsa.seasonal import STL
from statsmodels.graphics.tsaplots import acf, plot_pacf, plot_acf
from MyFunctions import ADF_Cal, ACF_plot, autocorrelation_cal,
series_autocorrelation_cal, ts_strength
from sklearn.model_selection import train_test_split
import warnings
warnings.filterwarnings("ignore")

#%% ------------------------------------ Load dataset --------------------
-----------------------------------------------
# read the preprocessed dataset
df = pd.read_csv("../data/Preprocessed_AirQuality.csv", index_col="Date",
parse_dates=True)

# get the target variable
target = "NO2(GT)"

# splitting into training and testing sets
df_train, df_test = train_test_split(df, test_size=0.2, shuffle=False)

# getting the training time series
ts = df_train[target]


#%% ----------------------------------------- Visualization of the time
series -----------------------------------------------
# plot first 300 samples
fig, ax = plt.subplots()
ax.plot(ts[:300])
ax.xaxis.set_tick_params(reset=True)
ax.xaxis.set_major_locator(mdates.HourLocator(interval=48))
ax.xaxis.set_major_formatter(mdates.DateFormatter("%b/%d-%H"))
plt.setp(ax.get_xticklabels(), rotation=30, fontsize = 10)
plt.title("Figure 1. Hourly averaged NO2 concentration of 300 samples")
plt.xlabel("Date")
plt.ylabel(target + " concentration in microg/m^3")
```

```
plt.show()

#%% ------------------------------------------ ACF/PACF plot -------------
----------------------------------------------
y = ts.to_numpy() # convert to np.array

# ACF
plt.figure()
plot_acf(y, lags=48, title="Figure 2. ACF plot of the time series")
plt.xlabel("Lag")
plt.ylabel("Magnitude")
plt.show()

# PACF
plt.figure()
plt.figure()
plot_pacf(y, lags=48, title="PACF plot of the time series")
plt.xlabel("Lag")
plt.ylabel("Magnitude")
plt.show()

#%% ------------------------------------------ ADF test for stationarity -
---------------------------------------------------------------
# calculation of rolling mean and rolling variance
rolling_mean = []
rolling_var = []
for i in range(1,len(y)):
    rolling_mean.append(np.mean(y[:i]))
    rolling_var.append(np.var(y[:i]))

# ADF tests for the time series, rolling mean and rolling variance
print("\nADF test for the time series:")
ADF_Cal(y)
print("\nADF test for rolling mean:")
ADF_Cal(rolling_mean)
print("\nADF test for rolling variance:")
ADF_Cal(rolling_var)

#%% ------------------------------ Analysis Time series components -------
--------------------------------------------------------------
# STL decomposition
stl = STL(ts[:500])
res = stl.fit()
plt.figure()
fig = res.plot()
fig.axes[0].set_xticks([], [])
fig.axes[1].set_xticks([], [])
fig.axes[2].set_xticks([], [])
fig.axes[3].xaxis.set_major_locator(mdates.HourLocator(interval=48))
fig.axes[3].xaxis.set_major_formatter(mdates.DateFormatter("%m-%d-%H:00"))
plt.setp(fig.axes[3].get_xticklabels(), rotation=30, fontsize=7)
plt.suptitle("Figure 3. STL decomposition (showing 1000 samples)")
plt.xlabel("Date")
plt.show()
```

```
# plot time series component seperately in a graph
Tt = res.trend  ## trend-cycle component
St = res.seasonal ## seasonal component
Rt = res.resid ## remainder component

plt.figure()
plt.plot(Tt[:500], label="Trend")
plt.plot(St[:500], label="Seasonality")
plt.plot(Rt[:500], label="Reminder")
plt.title("Time series components for 1000 samples")
plt.xticks(rotation = 30)
plt.xlabel("Date")
plt.ylabel("Magnitude")
plt.legend()
plt.show()



#%% ------------------------------ Strength of Trend-cycle and Seasonality
---------------------------------------------
# calculate the strength of trend-cycle and seasonaliry
Ft, Fs = ts_strength(St,Rt,Tt)

# print results
print("\n")
print("The strength of trend-cycle is: {:.4f}".format(Ft))
print("The strength of seasonality is: {:.4f}".format(Fs))
```

Baseline models and Holt models

```
import random
import pandas as pd
import matplotlib.pyplot as plt
import statsmodels.tsa.holtwinters as ets
import scipy
import numpy as np
import matplotlib.dates as mdates
from pandas.plotting import register_matplotlib_converters
register_matplotlib_converters()
from sklearn.model_selection import train_test_split
from MyFunctions import simple_forecast_ts, forecasting_plot, stats,
Q_val_cal, ACF_error, ACF_plot, autocorrelation_cal,
series_autocorrelation_cal, corr_cal
import warnings
warnings.filterwarnings("ignore")

#%% ------------------------------------------ Set-up -------------------
--------------------------------------------------------
SEED = 42
```

```
random.seed(SEED)
np.random.seed(SEED)
#%% ----------------------------------- Load data and split training and
testing sets ----------------------------------------
df = pd.read_csv("../data/Preprocessed_AirQuality.csv", index_col="Date",
parse_dates=True)
target = "NO2(GT)"  # target variable

# splitting training and testing sets
df_train, df_test = train_test_split(df, test_size=0.2, shuffle=False)
train = df_train[target]
test = df_test[target]

#%% ------------------------------------------- Baseline models & Holt-
Winter ----------------------------------------------------------------

# plot training, testing sets and the forecasts
forecasting_plot(train, test, "Average", "Date", target, 2, "%Y-%m-%d", 0)
forecasting_plot(train, test, "Naive", "Date", target, 2, "%Y-%m-%d",  0)
forecasting_plot(train, test, "Drift", "Date", target, 2, "%Y-%m-%d",  0)
forecasting_plot(train, test, "Simple Exponential Smoothing", "Date",
target ,2, "%Y-%m-%d", 0)
forecasting_plot(train, test, "Holt's Linear", "Date", target, 2, "%Y-%m-
%d", 0)
forecasting_plot(train, test, "Holt-Winter", "Date", target, 2, "%Y-%m-
%d", 24)

#%% -------------------------------------------- Evaluation metrics --------
---------------------------------------------------
# MSE, variances of prediction and forecast errors, Q value and
correlation coefficient
stats(train, test, "Average",0)
stats(train, test, "Naive",0)
stats(train, test, "Drift",0)
stats(train, test, "Simple Exponential Smoothing",0)
stats(train, test, "Holt's Linear",0)
stats(train, test, "Holt-Winter",24)

#%% ------------------------------------------------ ACF plots for residuals
----------------------------------------------
# ACF plots
ACF_error("Average", train, test,0)
ACF_error("Naive", train, test,0)
ACF_error("Drift", train, test,0)
ACF_error("Simple Exponential Smoothing", train, test,0)
ACF_error("Holt's Linear", train, test,0)
ACF_error("Holt-Winter", train, test,24)

#%% -------------------------------------- Critical Q-value ----------------
----------------------------------------------------
DOF = 48
alpha = 0.05
critical_Q = scipy.stats.chi2.ppf(1-alpha, DOF)
print("Critical Q-value = ", critical_Q)
```

Multiple Linear Regression

```
import pandas as pd
import matplotlib.pyplot as plt
import statsmodels.tsa.holtwinters as ets
import numpy as np
from sklearn.model_selection import train_test_split
import statsmodels.api as sm
from sklearn.preprocessing import StandardScaler
import seaborn as sns
from scipy import stats
from MyFunctions import Q_val_cal, ACF_error, ACF_plot,
autocorrelation_cal, series_autocorrelation_cal, corr_cal
import warnings
warnings.filterwarnings("ignore")


#%%------------------------------------- Data preparation ----------------
---------------------------------------------------
# load data
data = pd.read_csv("../data/Preprocessed_AirQuality.csv",
index_col="Date", parse_dates=True)

target = "NO2(GT)"  ## target variable
features = np.setdiff1d(data.columns, [target]).tolist()   ## features
mete_features = ["T", "RH", "AH"]     ## meteorological variables

# split the data into training and testing sets
df_train, df_test = train_test_split(data, shuffle=False, test_size=0.2)

#%% ----------------------------------------- Standardizing data ---------
-----------------------------------------------
# since the variables are not in the same scale, we need to standardize
them
# The StandardScaler
ss = StandardScaler()
# fit and standardize the training data
train = pd.DataFrame(ss.fit_transform(df_train), columns=df_train.columns)
# apply standardization to testing data
test = pd.DataFrame(ss.transform(df_test), columns=df_test.columns)

#%%  --------------------------------------- Getting feature matrix and
target variable --------------------------------
X_train = train[features] # getting the feature matrix for the training
set
Y_train = train[target] # the target variable for the training set

X_test = test[features]  # getting the feature matrix for the testing set
Y_test = test[target] # the target variable for the testing set


#%% ---------------------------------------------- Correlation matrix -----
-----------------------------------------------
```

```
# plot the correlation matrix to observe the relationships between the
target variable and the features
# correlation matrix
corr_matrix = data.corr()
# plt correlation matrix
ax = sns.heatmap(corr_matrix,vmin =-1, vmax=1, center=0,
            cmap=sns.diverging_palette(20,220,n=200),
            square = True, annot = False)
plt.title("Figure 9. Correlation matrix")
bottom, top =ax.get_ylim()
ax.set_ylim(bottom+0.5,top-0.5)
ax.set_xticklabels(ax.get_xticklabels(), rotation = 45,
horizontalalignment='right')
plt.show()

#%% ----------------------------- Estimate the coefficients of MLR using
normal equation ------------------------------
# number of samples
T = X_train.shape[0]
# number of features
k = len(features)
# we will create X matrix with T rows and k+1 columns
X = []

# the 1st column of matrix X are 1s
for i in range(T):
    X.append(1)
X = [X]

# other rows
for var in features:
    X.append(X_train[var])

# transform X into a matrix
X = np.array(X).T

# create matrix Y
Y = np.array([Y_train]).T

# calculate the coefficient using LSE equation
B = np.dot(np.dot(np.linalg.inv(np.dot(X.T,X)),X.T),Y)
print("Matrix of regression coefficients is:\n", B,"\n")
count = 1
print("The intercept is: ", B[0,0])
for var in X_train.columns:
    print("The coefficient of "+var+" is: ", B[count,0])
    count = count +1

#%% ----------------------------------------------- MLR using OLS command
----------------------------------------------
# since the model need an intercept, we add a column of 1s to X_train
X_train = sm.add_constant(X_train)

# fit the model to training set
```

```
model = sm.OLS(Y_train, X_train).fit()

# evaluation metrics
print("\n",model.summary(),"\n")
print("Adj R2 : ", model.rsquared_adj)
print("AIC : ", model.aic)
print("BIC : ", model.bic)

#%% ---------------------------------------------- Predictions & Forecasts
----------------------------------------------
# predictions
predictions = model.predict(X_train)

# transform predictions to original scale
train_copy = train.copy(deep=True)
train_copy[target] = predictions
train_copy = pd.DataFrame(ss.inverse_transform(train_copy),
                              columns=train_copy.columns)
# getting fitted values
predictions = train_copy[target]

#
==========================================================================
============================================
# forecasts
X_test = sm.add_constant(X_test)
forecasts = model.predict(X_test)

# transform forecasts to original scale
test_copy = test.copy(deep=True)
test_copy[target] = forecasts
test_copy = pd.DataFrame(ss.inverse_transform(test_copy),
                              columns=test_copy.columns)
# getting forecasts in the original scale
forecasts = test_copy[target]

# plot the predictions and forecasts
inds = data.index
train_inds, test_inds = train_test_split(inds, test_size=0.2,
shuffle=False)
plt.figure()
plt.plot(train_inds[-100:], df_train[target][-100:], label ="Training
set")
plt.plot(train_inds[-100:], predictions[-100:], label ="Predictions")
plt.plot(test_inds[:200],df_test[target][:200], label="Testing set")
plt.plot(test_inds[:200],forecasts[:200], label="Forecasts")
plt.xlabel("Date")
plt.xticks(rotation=30)
plt.ylabel(target)
plt.title("Forecasting of NO2 concentration using MLR")
plt.legend(loc='best')
plt.show()
```

```
#%% ------------------------------------ Forecasted Errors and Residuals -
----------------------------------------------
# errors
errors =  np.array(df_test[target]) -np.array(forecasts)
SSE_test = np.square(errors).sum()
MSE_test = np.square(errors).mean()
est_var_test = SSE_test/(len(Y_test)-k-1)

# residuals
residuals =  np.array(df_train[target]) -np.array(predictions)
SSE_train = np.square(residuals).sum()
MSE_train = np.square(residuals).mean()
est_var_train = SSE_train/(T-k-1)

# print statistics
print("Mean of residuals is: ", np.mean(residuals))
print("MSE of fitted values using LME is: ", MSE_train)
print("MSE of forecasted values using LME is: ", MSE_test)
print("The estimated variance of prediction errors is:", est_var_train)
print("The estimated variance of forecast errors is:", est_var_test)
print("The variance of residuals is:", np.var(residuals))
print("The variance of forecast errors is:", np.var(errors))

#%% --------------------------------------- Diagnostics for Residuals ----
-----------------------------------------------
# ACF plot
title = "ACF plot for residuals"
h = 48
ACF_plot(residuals,h, title)
Q_val = Q_val_cal(residuals, h,T)
print("Q-value of residuals is: ", Q_val)

# Fitted values vs True values
r = corr_cal(np.array(predictions),np.array(df_train[target]))
plt.figure()
sns.regplot(np.array(predictions), np.array(df_train[target]),
label="Predictions vs Targets",line_kws={"color": "red"})
plt.title("True values vs Predictions with correlation
r={:.4f}".format(r))
plt.xlabel("True values")
plt.ylabel("Fitted values")
plt.show()

# Residuals vs Fitted values
r = corr_cal(np.array(predictions),residuals)
plt.figure()
sns.regplot(np.array(predictions), residuals, label="Predictions vs
Residuals",line_kws={"color": "red"})
plt.title("Residuals vs Fitted values with correlation
r={:.4f}".format(r))
plt.xlabel("Fitted values")
plt.ylabel("Residuals")
plt.show()
```

```
#%% ----------------------------------- Backward stepwise using AIC, BIC,
Adj R2 ---------------------------------------
# create a deep copy of the training set so updates wont affect the
original data.
Xtrain = X_train.copy(deep=True)
AIC = model.aic
BIC = model.bic
R2 = model.rsquared_adj
print("\nBackward stepwise using AIC, BIC and Adjusted R_square:")
# threshold for adj R2
# if adj R2 increase while AIC and BIC improved, we definitely update the
model
# if adj R2 decreases an amount below this threshold while the AIC and BIC
improved,
# we update the model since the decrease of adj R2 does not hurt the model
performance
threshold = 0.005

for var in features:
    print("\n" + "-"*10 +" Dropping " + var + "-"*10)
    print("Previous AIC ={:.4f}".format(AIC), ",BIC ={:.4f}".format(BIC),
",Adj_R2 = {:.4f}".format(R2))
    X_opt = Xtrain.drop(var, axis =1, inplace = False)
    model = sm.OLS(Y_train, X_opt).fit()
    print("New AIC ={:.4f}".format(model.aic), ",BIC
={:.4f}".format(model.bic), ",Adj_R2 = {:.4f}".format(model.rsquared_adj))
    R2_diff = np.subtract(R2, model.rsquared_adj)
    if (model.aic < AIC) and (model.bic < BIC) and (R2_diff < threshold):
        print("The feature "+ var +" is not important, we can drop it from
the model.")
        Xtrain.drop(var, axis =1, inplace = True)
        AIC = model.aic
        BIC = model.bic
        R2 = model.rsquared_adj
    else:
        print("The feature "+ var +" is important, we need to keep it in
the model.")
        model = sm.OLS(Y_train, Xtrain).fit()

features_to_be_eliminated = []
for var in features:
    if var not in Xtrain.columns:
        features_to_be_eliminated.append(var)
print("\nFeatures to be eliminated are ", features_to_be_eliminated)
print("\n",model.summary(),"\n")
print("Adj R2 : ", model.rsquared_adj)
print("AIC : ", model.aic)
print("BIC : ", model.bic)

#%% ------------------------------------ Backward stepwise using p-values
of t-test -----------------------------------
print("\nBackward stepwise using t-test p_values:\n")
Xtrain = X_train.copy(deep=True) # copy the original training set
alpha = 0.05 # significant level aplha - confident level = 95%
```

```
# retrain the model with all features
model = sm.OLS(Y_train, X_train).fit()

# get the p values
l = model.pvalues
features_to_be_eliminated = []

while max(l[1:]) > alpha or max(l[1:])  == alpha: # while there is a p-
value larger than significane level
    features_copy = l.index # get the feature index
    index = np.argmax(l[1:]) # get the index of the feature with max p-
value
    var = features_copy[index+1] # get the name of the feature with max p-
value
    print("The t-test p_value for "+var+ " is {:.4f}".format(max(l)))
    print("The regression coefficient of feature "+ var+" is not
statistically different than 0.")
    print("Dropping " + var + " from the model .............")
    features_to_be_eliminated.append(var)
    print(" \n--------------------- Training new regression model --------
-------------")

    # drop the feature with the max p-value if p-value > 0.05
    Xtrain.drop(var, axis =1, inplace=True)
    # training new model
    model = sm.OLS(Y_train, Xtrain).fit()
    l = model.pvalues # get the new list of p-values
print("\nBackward stepwise completed!")
print("\nFeatures to be eliminated are ", features_to_be_eliminated)
print("\n",model.summary(),"\n")
print("Adj R2 : ", model.rsquared_adj)
print("AIC : ", model.aic)
print("BIC : ", model.bic)

#%% ---------------------------------------- Forward stepwise using AIC,
BIC, Adj R2 ------------------------------------
Xtrain = X_train[X_train.columns[0]].copy(deep=True) # intercept
# retrain the intercept model
model1 = sm.OLS(Y_train, Xtrain).fit()
# metrics
AIC = model1.aic
BIC = model1.bic
R2 = model1.rsquared_adj
print("\nForward stepwise using AIC, BIC and Adjusted R_square:")
threshold = 0.005
features = [var for var in features if var != X_train.columns[0]]
feature_opt = []
feature_opt.append(X_train.columns[0])
feature_not_add = []
for var in features:
    print("\n" + "-"*10 +" Adding " + var + "-"*10)
    feature_opt.append(var) # add var to feature
```

```
    Xtrain = X_train[feature_opt].copy(deep=True) # create feature matrix
to be trained
    model1 = sm.OLS(Y_train, Xtrain).fit()
    print("New AIC ={:.4f}".format(model1.aic), ",BIC
={:.4f}".format(model1.bic), ",Adj_R2 =
{:.4f}".format(model1.rsquared_adj))
    #R2_diff = np.subtract(R2, model.rsquared_adj)
    if (model1.aic < AIC) and (model1.bic < BIC) and (model1.rsquared_adj
> R2):
        print("The feature "+ var +" is important, we add it to the
model.")
        AIC = model1.aic
        BIC = model1.bic
        R2 = model1.rsquared_adj
    else:
        feature_opt = [x for x in feature_opt if x!= var]
        feature_not_add.append(var)
        print("The feature "+ var +" is not important, we do not add it to
the model.")

print("\nFeatures are not added are ", feature_not_add)
print("\n",model1.summary(),"\n")
print("Adj R2 : ", model1.rsquared_adj)
print("AIC : ", model1.aic)
print("BIC : ", model1.bic)

#%% ----------------------------------- Forward stepwise using p-values of
t-test ---------------------------------------
print("\nForward stepwise using t-test p_values:\n")
alpha = 0.05 # significant level aplha - confident level = 95%
Xtrain = X_train[X_train.columns[0]].copy(deep=True) # intercept
feature_opt = []
feature_opt.append(X_train.columns[0])
feature_not_add = []

for var in features:
    print("\n" + "-"*10 +" Adding " + var + "-"*10)
    feature_opt.append(var) # add var to feature
    Xtrain = X_train[feature_opt].copy(deep=True) # create feature matrix
to be trained
    model1 = sm.OLS(Y_train, Xtrain).fit()
    pval= model1.pvalues.loc[var]
    if  pval < alpha :
        print("The feature "+ var +" is important, we add it to the
model.")
        AIC = model1.aic
        BIC = model1.bic
        R2 = model1.rsquared_adj
    else:
        feature_opt = [x for x in feature_opt if x!= var]
        feature_not_add.append(var)
        print("The feature "+ var +" causes a feature not significant due
to p-value {:.4f}, we do not add it to the model.".format(pval))
```

```
print("\nForward stepwise completed!")
print("\nFeatures not added are ", feature_not_add)
print("\n",model1.summary(),"\n")
print("Adj R2 : ", model1.rsquared_adj)
print("AIC : ", model1.aic)
print("BIC : ", model1.bic)

#%% ------------------------------------------ Evaluation --------------
---------------------------------------------
# After feature selection, the best model is from backward stepwise
regression
# remaining features after feature selection
features = [x for x in X_test.columns if x not in
features_to_be_eliminated]

# getting new feature matrix
X_train = X_train[features]
X_test = X_test[features]

#%% ------------------------------------------ Predictions & Forecasts
---------------------------------------------
# predictions
predictions = model.predict(X_train)

# transform predictions to original scale
train_copy = train.copy(deep=True)
train_copy[target] = predictions
train_copy = pd.DataFrame(ss.inverse_transform(train_copy),
                          columns=train_copy.columns)
# getting fitted values
predictions = train_copy[target]

#
==========================================================================
============================================
# forecasts
#X_test = sm.add_constant(X_test)
forecasts = model.predict(X_test)

# transform forecasts to original scale
test_copy = test.copy(deep=True)
test_copy[target] = forecasts
test_copy = pd.DataFrame(ss.inverse_transform(test_copy),
                          columns=test_copy.columns)
# getting forecasts in the original scale
forecasts = test_copy[target]

# plot the predictions and forecasts
inds = data.index
train_inds, test_inds = train_test_split(inds, test_size=0.2,
shuffle=False)
plt.figure()
plt.plot(train_inds[-100:], df_train[target][-100:], label ="Training
set")
```

```
plt.plot(train_inds[-100:], predictions[-100:], label ="Predictions")
plt.plot(test_inds[:200],df_test[target][:200], label="Testing set")
plt.plot(test_inds[:200],forecasts[:200], label="Forecasts")
plt.xlabel("Date")
plt.xticks(rotation=30)
plt.ylabel(target)
plt.title("Figure 4. Forecasting of NO2 concentration using MLR")
plt.legend(loc='best')
plt.show()

#%% ------------------------------------ Forecasted Errors and Residuals -
-----------------------------------------------
# errors
errors =  np.array(df_test[target]) -np.array(forecasts)
SSE_test = np.square(errors).sum()
MSE_test = np.square(errors).mean()
est_var_test = SSE_test/(len(Y_test)-k-1)

# residuals
residuals =  np.array(df_train[target]) -np.array(predictions)
SSE_train = np.square(residuals).sum()
MSE_train = np.square(residuals).mean()
est_var_train = SSE_train/(T-k-1)

# print statistics
print("Mean of residuals is: ", np.mean(residuals))
print("MSE of fitted values using LME is: ", MSE_train)
print("MSE of forecasted values using LME is: ", MSE_test)
print("The estimated variance of prediction errors is:", est_var_train)
print("The estimated variance of forecast errors is:", est_var_test)
print("The variance of residuals is:", np.var(residuals))
print("The variance of forecast errors is:", np.var(errors))

#%% ---------------------------------------- Diagnostics for Residuals ----
------------------------------------------------
# ACF plot
title = "Figure 5. ACF plot for residuals"
h = 48
ACF_plot(residuals,h, title)
Q_val = Q_val_cal(residuals, h,T)
print("Q-value of residuals is: ", Q_val)

# Fitted values vs True values
r = corr_cal(np.array(predictions),np.array(df_train[target]))
plt.figure()
sns.regplot(np.array(predictions), np.array(df_train[target]),
label="Predictions vs Targets",line_kws={"color": "red"})
plt.title("Figure 6. True values vs Predictions with correlation
r={:.4f}".format(r))
plt.xlabel("True values")
plt.ylabel("Fitted values")
plt.show()

# Residuals vs Fitted values
```

```
r = corr_cal(np.array(predictions),residuals)
plt.figure()
sns.regplot(np.array(predictions), residuals, label="Predictions vs
Residuals",line_kws={"color": "red"})
plt.title("Figure 7. Residuals vs Fitted values with correlation
r={:.4f}".format(r))
plt.xlabel("Fitted values")
plt.ylabel("Residuals")
plt.show()

#%% -------------------------------- Realistic MLR using meteorological
features ---------------------------------------------------------------------------
---------
X_train = X_train[mete_features]
X_train = sm.add_constant(X_train) # add column of 1s

X_test = X_test[mete_features]
X_test = sm.add_constant(X_test)

# fit the model to training set
model = sm.OLS(Y_train, X_train).fit()

# evaluation metrics
print("\n",model.summary(),"\n")
print("Adj R2 : ", model.rsquared_adj)
print("AIC : ", model.aic)
print("BIC : ", model.bic)

#%% --------------------------------------------- Predictions & Forecasts
----------------------------------------------
# predictions
predictions = model.predict(X_train)

# transform predictions to original scale
train_copy = train.copy(deep=True)
train_copy[target] = predictions
train_copy = pd.DataFrame(ss.inverse_transform(train_copy),
                              columns=train_copy.columns)
# getting fitted values
predictions = train_copy[target]

#
===========================================================================
============================================
# forecasts
forecasts = model.predict(X_test)

# transform forecasts to original scale
test_copy = test.copy(deep=True)
test_copy[target] = forecasts
test_copy = pd.DataFrame(ss.inverse_transform(test_copy),
                              columns=test_copy.columns)
# getting forecasts in the original scale
forecasts = test_copy[target]
```

```
# plot the predictions and forecasts
inds = data.index
train_inds, test_inds = train_test_split(inds, test_size=0.2,
shuffle=False)
plt.figure()
plt.plot(train_inds[-100:], df_train[target][-100:], label ="Training
set")
plt.plot(train_inds[-100:], predictions[-100:], label ="Predictions")
plt.plot(test_inds[:200],df_test[target][:200], label="Testing set")
plt.plot(test_inds[:200],forecasts[:200], label="Forecasts")
plt.xlabel("Date")
plt.xticks(rotation=30)
plt.ylabel(target)
plt.title("Figure 8. MLR model using meteorological variables as
predictors")
plt.legend(loc='best')
plt.show()

#%% ------------------------------------ Forecasted Errors and Residuals -
---------------------------------------------
# errors
errors =  np.array(df_test[target]) - np.array(forecasts)
SSE_test = np.square(errors).sum()
MSE_test = np.square(errors).mean()
est_var_test = SSE_test/(len(Y_test)-k-1)

# residuals
residuals =  np.array(df_train[target]) - np.array(predictions)
SSE_train = np.square(residuals).sum()
MSE_train = np.square(residuals).mean()
est_var_train = SSE_train/(T-k-1)

# print statistics
print("Mean of residuals is: ", np.mean(residuals))
print("MSE of fitted values using LME is: ", MSE_train)
print("MSE of forecasted values using LME is: ", MSE_test)
print("The estimated variance of prediction errors is:", est_var_train)
print("The estimated variance of forecast errors is:", est_var_test)
print("The variance of residuals is:", np.var(residuals))
print("The variance of forecast errors is:", np.var(errors))

#%% ---------------------------------------- Diagnostics for Residuals ----
-----------------------------------------------
# ACF plot
title = "ACF plot for residuals"
h = 48
ACF_plot(residuals,h, title)
Q_val = Q_val_cal(residuals, h,T)
print("Q-value of residuals is: ", Q_val)
DOF = h - len(mete_features)
alpha = 0.05
critical_Q = stats.chi2.ppf(1-alpha, DOF)
print("Critical Q-value = ", critical_Q)
```

```
# Fitted values vs True values
r = corr_cal(np.array(predictions),np.array(df_train[target]))
plt.figure()
sns.regplot(np.array(predictions), np.array(df_train[target]),
label="Predictions vs Targets",line_kws={"color": "red"})
plt.title("Figure 10. True values vs Predictions with correlation
r={:.4f}".format(r))
plt.xlabel("True values")
plt.ylabel("Fitted values")
plt.show()

# Residuals vs Fitted values
r = corr_cal(np.array(predictions),residuals)
plt.figure()
sns.regplot(np.array(predictions), residuals, label="Predictions vs
Residuals",line_kws={"color": "red"})
plt.title("Figure 11. Residuals vs Fitted values with correlation
r={:.4f}".format(r))
plt.xlabel("Fitted values")
plt.ylabel("Residuals")
plt.show()
```

ARMA

```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import matplotlib.dates as mdates
from pandas.plotting import register_matplotlib_converters
register_matplotlib_converters()
from scipy import signal, stats
import copy
from statsmodels.tsa.stattools import adfuller
from statsmodels.tsa.seasonal import STL
from statsmodels.graphics.tsaplots import acf, plot_pacf, plot_acf
from MyFunctions import ADF_Cal, differencing, Q_val_cal, ACF_plot,
autocorrelation_cal, series_autocorrelation_cal, corr_cal, phi_cal,
GPAC_cal, LME, one_step_ARMA, h_step_ARMA
from sklearn.model_selection import train_test_split
import warnings
warnings.filterwarnings("ignore")


#%% ---------------------------------------- Load data ------------------
---------------------------------------------------
df = pd.read_csv("../data/Preprocessed_AirQuality.csv", index_col="Date",
parse_dates=True)
target = "NO2(GT)"

# splitting training and testing sets
```

```
df_train, df_test = train_test_split(df, test_size=0.2, shuffle=False)
train = df_train[target]
test = df_test[target]

#%% ----------------------------------------- Stationary -----------------
-------------------------------------------------
y = train.to_numpy()
# check stationary of original data
rolling_mean = []
rolling_var = []
for i in range(1,len(y)):
    rolling_mean.append(np.mean(y[:i]))
    rolling_var.append(np.var(y[:i]))

# ADF test
print("\nADF test for original data:")
ADF_Cal(y)
print("\nADF test for rolling mean:")
ADF_Cal(rolling_mean)
print("\nADF test for rolling variance:")
ADF_Cal(rolling_var)

plt.figure()
plot_acf(y, lags=48, title="ACF plot of NO2 concentrations")
plt.xlabel("Lag")
plt.ylabel("Magnitude")
plt.show()

# 1st differencing
y1 = differencing(y, 1)
# check for stationary
rolling_mean = []
rolling_var = []
for i in range(1,len(y)):
    rolling_mean.append(np.mean(y1[:i]))
    rolling_var.append(np.var(y1[:i]))

# ADF test
print("\nADF test for transformed data:")
ADF_Cal(y)
print("\nADF test for rolling mean:")
ADF_Cal(rolling_mean)
print("\nADF test for rolling variance:")
ADF_Cal(rolling_var)

# acf plot
plt.figure()
plot_acf(y1, lags=10, title="ACF plot of 1st differenced time series")
plt.xlabel("Lag")
plt.ylabel("Magnitude")
plt.show()

# pacf plot
plt.figure()
```

```
plt.figure()
plot_pacf(y1, lags=10, title="PACF plot of 1st differenced time series")
plt.xlabel("Lag")
plt.ylabel("Magnitude")
plt.show()

#%% -------------------------------------- GPAC table --------------------
-----------------------------------------------
# use GPAC table to find orders of AR and MA
lags = 48
# ACF of y(t)
acf = series_autocorrelation_cal(y1, lags)

# retrieve Ry
Ry = acf.loc[acf["lag"] > -1]
Ry.set_index("lag", inplace=True)
Ry = Ry["autocorrelation"].to_numpy()

# print and plot GPAC table
table = GPAC_cal(Ry,8,8)

#%% ---------------------------------- Levenberg-Marquardt algorithm -----
-----------------------------------------------
# use LM algorithm to estimate the parameters

# ARMA (2,1)
# na = 2
# nb = 1

# ARMA (5,5)
# na = 5
# nb = 5

# ARMA (1,1)
na = 1
nb = 1
a, b, running_SSE, var_e, cov_theta = LME(y1,na,nb,100)


#%% ------------------------------------------- 1-step prediction --------
----------------------------------------------------
y_train_pred, e_train = one_step_ARMA(y1,a,b)

# transform back to original order
y_pred = np.zeros(len(y))
for i in range(2, len(y)):
    y_pred[i] = y[i-1] + y_train_pred[i-1]

#%% ------------------------------------------ h-step prediction ---------
-----------------------------------------------------
y_test_pred = h_step_ARMA(y1, y_train_pred, len(test),a,b)

# convert back to original order
y_forecast = np.zeros(len(test))
```

```
y_forecast[0] = y[-1] + y_test_pred[0]
for i in range(1,len(test)):
    y_forecast[i] = y_forecast[i-1] + y_test_pred[i]

#%% ------------------------------------------- Diagnostics for residuals
--------------------------------------------------
res = np.array(train[2:]) - y_pred[2:]
title = "Figure 13. ACF plot of residuals"
T = len(train)
h = 48
ACF_plot(res, 20, title)
DOF = h - na - nb
alpha = 0.05
Q_val = Q_val_cal(res,h,T)
print("Q-value = ", Q_val)
critical_Q = stats.chi2.ppf(1-alpha, DOF)
print("Critical Q-value = ", critical_Q)

#%% ------------------------------------ Confidence interval -------------
---------------------------------
if na != 0:
    for i in range(1,na + 1):
        print("The confidence interval for parameter a{:} = [{:.4f} ,
{:.4f}]".format(i, a[i] - 2*np.sqrt(cov_theta[i-1,i-1]), a[i] +
2*np.sqrt(cov_theta[i-1,i-1]) ) )

if nb != 0:
    for i in range(1,nb + 1):
        print("The confidence interval for parameter b{:} = [{:.4f} ,
{:.4f}]".format(i, b[i] - 2*np.sqrt(cov_theta[i+na-1,i+na-1]), b[i] +
2*np.sqrt(cov_theta[i+na-1,i+na-1]) ) )


#%% ------------------------------------------- Zero/ Pole cancellation --
----------------------------------------------------
root_b = np.roots(b)
root_a = np.roots(a)

if nb != 0:
    for i in range(nb):
        print("The roots of numerators are: ", np.real(root_b[i]) )
if na != 0:
    for i in range(na):
        print("The roots of denominators are: ", np.real(root_a[i]) )

#%% -------------------------------------------- Statistics --------------
----------------------------------------------
# training set
SSE_train = np.square(res).sum()
MSE_train = np.square(res).mean()
est_var_train = SSE_train/(len(train)-na-nb)
print("MSE of fitted values using LME is: ", MSE_train)
print("Mean of residuals is: ", np.mean(res))
print("The estimated variance of residuals is:", var_e[0,0])
```

```
print("Variance of residuals is:", np.var(res))

# testing set
error = np.array(test) - y_forecast
SSE_test = np.square(error).sum()
MSE_test = np.square(error).mean()
est_var_test = SSE_test/(len(test)-na-nb)
print("MSE of testing set using LME is: ", MSE_test)
print("Mean of forecasted errors is: ", np.mean(error))
print("The estimated variance of errors is:", est_var_test)
print("Variance of errors is:", np.var(error))

# covariance matrix of parameters
print("The covariance matrix of estimated parameter is\n:", cov_theta)


#%% ---------------------------------------- Plot predictions and
forecasts ------------------------------------------------
inds1 = train[-100:].index
inds2 = test[:200].index
plt.figure()
plt.plot(inds1, y[-100:], label = "True values")
plt.plot(inds1, y_pred[-100:], label = "Fitted values")
plt.plot(inds2, np.array(test)[:200], label = "True values")
plt.plot(inds2, y_forecast[:200], label = "Forecasted values")
plt.title("Figure 12. 1-step and Multi-step prediction using ARMA
({:},{:})".format(na,nb))
plt.xticks(rotation=40)
plt.ylabel(target)
plt.xlabel("Date")
plt.legend(loc = "best")
plt.show()
```

ARIMA

```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import matplotlib.dates as mdates
from pandas.plotting import register_matplotlib_converters
register_matplotlib_converters()
from statsmodels.tsa.stattools import adfuller
from statsmodels.tsa.seasonal import STL
from statsmodels.graphics.tsaplots import acf, plot_pacf, plot_acf
from MyFunctions import ADF_Cal, differencing, Q_val_cal, ACF_plot,
autocorrelation_cal, series_autocorrelation_cal, corr_cal, phi_cal,
GPAC_cal, LME
from sklearn.model_selection import train_test_split
from scipy import stats
import warnings
warnings.filterwarnings("ignore")
```

```
#%% ------------------------------------------ Load data  ---------------
---------------------------------------
df = pd.read_csv("../data/Preprocessed_AirQuality.csv", index_col="Date",
parse_dates=True)
target = "NO2(GT)"
ts = df[target]

# splitting training and testing sets
df_train, df_test = train_test_split(df, test_size=0.2, shuffle=False)
train = df_train[target]
test = df_test[target]

#%% --------------------------------------- De-seasonalize --------------
---------------------------------------
# decompose training data
stl = STL(train)
res = stl.fit()
plt.figure()
fig = res.plot()
fig.axes[0].set_xticks([], [])
fig.axes[1].set_xticks([], [])
fig.axes[2].set_xticks([], [])
fig.axes[3].xaxis.set_major_locator(mdates.MonthLocator(interval=1))
fig.axes[3].xaxis.set_major_formatter(mdates.DateFormatter("%Y-%m"))
plt.setp(fig.axes[3].get_xticklabels(), rotation=30, fontsize=7)
plt.suptitle("STL decomposition for training set")
plt.xlabel("Date")
plt.show()

Tt = res.trend
St = res.seasonal
Rt = res.resid

plt.figure()
plt.plot(Tt[:500], label="Trend")
plt.plot(St[:500], label="Seasonality")
plt.plot(Rt[:500], label="Reminder")
plt.title("Time series components for 500 samples")
plt.xticks(rotation = 30)
plt.xlabel("Date")
plt.ylabel("Magnitude")
plt.legend()
plt.show()

#%%-------------------------------- Seasonal Adjusted data --------------
------------------------------------------------
At = train - St

plt.figure()
plt.plot(train[:300], label = "Original data")
plt.plot(At[:300], label ="De-seasonalized data")
plt.title("Figure 14. Seasonal adjusted data (300 samples)")
```

```
plt.xticks(rotation = 30)
plt.xlabel("Date")
plt.ylabel("Magnitude")
plt.legend()
plt.show()


#%% --------------------------------------------- Stationary -------------
---------------------------------------------------

y = At.to_numpy()
# 1st differencing
y = differencing(y,lag=1)
#y = differencing(y,lag=1)

# check for stationary
rolling_mean = []
rolling_var = []
for i in range(1,len(y)):
    rolling_mean.append(np.mean(y[:i]))
    rolling_var.append(np.var(y[:i]))

# ADF test
print("\nADF test for transformed data:")
ADF_Cal(y)
print("\nADF test for rolling mean:")
ADF_Cal(rolling_mean)
print("\nADF test for rolling variance:")
ADF_Cal(rolling_var)

plt.figure()
plot_acf(y, lags=20)
plt.title("ACF plot for seasonal adjusted data after 1st differencing")
plt.show()


plt.figure()
plot_pacf(y, lags=20)
plt.title("PACF plot for seasonal adjusted data after 1st differencing")
plt.show()

#%% ------------------------------------------------- GPAC table ---------
--------------------------------------------------------
acf = series_autocorrelation_cal(y, lags=96)
# retrieve Ry
Ry = acf.loc[acf["lag"] > -1]
Ry.set_index("lag", inplace=True)
Ry = Ry["autocorrelation"].to_numpy()

# print and plot GPAC table
table = GPAC_cal(Ry,7,7)

# there are two possible ARIMA model: ARIMA (1,1,1) and ARIMA (3,1,3)
```

```
#%% -------------------------------- ARIMA (3,1,3) zero/pole cancellation
-----------------------------------------------------------------
a, b, running_SSE, var_e, cov_theta = LME(y,3,3,100)

# zero/pole cancellation
root_b = np.roots(b)
root_a = np.roots(a)

for i in range(3):
    print("The roots of numerators are: ", np.real(root_b[i]) )

for i in range(3):
    print("The roots of denominators are: ", np.real(root_a[i]) )


#%% ARIMA(1,1,1)

# ARIMA(1,1,1)
d = 1
p = 1
q = 1
from statsmodels.tsa.forecasting.stl import STLForecast
from statsmodels.tsa.arima.model import ARIMA

# getting index freq
train.index.freq = train.index.inferred_freq

# apply ARIMA model in stlf command
stlf = STLForecast(train, ARIMA, model_kwargs=dict(order=(p,d,q)))
# fit the model
stlf_res = stlf.fit()
# make forecasts
forecast = stlf_res.forecast(len(test))
# model summary
stlf_res.summary()


#%% ------------------------------------------- Testing set statistics ---
----------------------------------------------------
# estimated variance
na = stlf_res.model.k_ar
nb = stlf_res.model.k_ma

errors = np.array(test) - np.array(forecast)
SSE_test = np.square(errors).sum()
MSE_test = np.square(errors).mean()
est_var_test = SSE_test/(len(test)-na-nb)
r = corr_cal(np.array(test),np.array(forecast))
print("Correlation coefficient between forecasted values and testing data
is: ", r)
print("MSE of forecasted values is: ", MSE_test)
print("The estimated variance of forecasted errors is:", est_var_test )
print("The variance of forecasted errors is:", np.var(errors) )
```

```
#%% ------------------------------------------ Training set statistics ---
---------------------------------------------------
prediction =
stlf_res.get_prediction(train.index[0],train.index[len(train)-
1]).predicted_mean
residuals = np.array(train) - np.array(prediction)
SSE_train = np.square(residuals).sum()
MSE_train = np.square(residuals).mean()
est_var_train = SSE_train/(len(train)-na-nb)
r = corr_cal(np.array(train),np.array(prediction))
print("Correlation coefficient between fitted values and training data is:
", r)
print("Mean of residuals is:" , np.mean(residuals))
print("MSE of fitted values is: ", MSE_train)
print("The estimated variance of residuals is:", est_var_train )
print("The variance of residuals is:", np.var(residuals))

#%% -------------------------------------------- Residuals and Box-Pierce
test ---------------------------------------------------------------------
# acf plot and Q value
h = 48
title = "Figure 16. ACF plot of residuals using ARIMA (1,1,1)"
ACF_plot(residuals, 20, title)
Q_val = Q_val_cal(residuals, h,len(test))
print("Q-value of residuals is: ", Q_val)
DOF = h -na - nb
alpha = 0.05
critical_Q = stats.chi2.ppf(1-alpha, DOF)
print("Critical Q-value = ", critical_Q)

#%% ------------------------------------ Plot predicitons and forecasts --
------------------------------------------------
plt.figure()
plt.plot(train[-100:], label ="Training data")
plt.plot(prediction[-100:], label ="Fitted values")
plt.plot(test[:200], label="Testing data")
plt.plot(forecast[:200], label="Forecasted values")
plt.xlabel("Date")
plt.xticks(rotation=30)
plt.ylabel(target)
plt.title("Figure 15. Forecasting of NO2 concentration using ARIMA
(1,1,1)")
plt.legend(loc='best')
plt.show()
```

SARIMA

```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import matplotlib.dates as mdates
from pandas.plotting import register_matplotlib_converters
register_matplotlib_converters()
```

```
from scipy import signal, stats
import copy
from statsmodels.tsa.stattools import adfuller
from statsmodels.tsa.seasonal import STL
from statsmodels.graphics.tsaplots import acf, plot_pacf, plot_acf
from MyFunctions import ADF_Cal, differencing, Q_val_cal, ACF_plot,
autocorrelation_cal, series_autocorrelation_cal, corr_cal, phi_cal,
GPAC_cal, LME, one_step_ARMA, h_step_ARMA
from sklearn.model_selection import train_test_split
from statsmodels.tsa.statespace.sarimax import SARIMAX
import warnings
warnings.filterwarnings("ignore")


#%% ------------------------------------------------- Load data ----------
--------------------------------------------------
df = pd.read_csv("../data/Preprocessed_AirQuality.csv", index_col="Date",
parse_dates=True)
target = "NO2(GT)"

# splitting training and testing sets
df_train, df_test = train_test_split(df, test_size=0.2, shuffle=False)
train = df_train[target]
test = df_test[target]

#%% ----------------------------------------------- Seasonal differencing
--------------------------------------------------
# at first, we take a seasonal differencing of the time series
y = train.to_numpy()

# the seasonal period is 24, so we take a differencing of lag 24
y = differencing(y, lag = 24)

# ADF test for seasonal differenced data
rolling_mean = []
rolling_var = []
for i in range(1,len(y)):
    rolling_mean.append(np.mean(y[:i]))
    rolling_var.append(np.var(y[:i]))

# ADF test
print("\nADF test for seasonal differenced data:")
ADF_Cal(y)
print("\nADF test for rolling mean:")
ADF_Cal(rolling_mean)
print("\nADF test for rolling variance:")
ADF_Cal(rolling_var)

#%% --------------------------------------------------- 1st differencing -
--------------------------------------------------
# Since the time series is not stationary after seasonal differencing
# We apply an additional 1st differencing

y = differencing(y, lag = 1)
```

```
# ADF test for seasonal differenced data
rolling_mean = []
rolling_var = []
for i in range(1,len(y)):
    rolling_mean.append(np.mean(y[:i]))
    rolling_var.append(np.var(y[:i]))

# ADF test
print("\nADF test for the time series after seasonal differencing followed
by a 1st differencing:")
ADF_Cal(y)
print("\nADF test for rolling mean:")
ADF_Cal(rolling_mean)
print("\nADF test for rolling variance:")
ADF_Cal(rolling_var)

# differencing terms: d = 1, D = 1
#%% -------------------------------------------------- ACF, PCAF plot -----
-------------------------------------------------
plt.figure()
plot_acf(y, lags=100, title="Figure 17. ACF plot of differenced data")
plt.xlabel("Lag")
plt.ylabel("Magnitude")
plt.show()


plt.figure()
plot_pacf(y, lags=100, title="Figure 18. PACF plot of differenced data")
plt.xlabel("Lag")
plt.ylabel("Magnitude")
plt.show()

# seasonal terms:
# a spike at lag 24 of ACF plot but no other spikes
# exponential decay in the seasonal lags of the PACF at lag= 24, 48, ...
P = 0
Q = 1

#%% ------------------------------------------------- GPAC table ----------
-------------------------------------------------
# use GPAC table to find non-seasonal terms
lags = 20
# ACF of y(t)
acf = series_autocorrelation_cal(y, lags)

# retrieve Ry
Ry = acf.loc[acf["lag"] > -1]
Ry.set_index("lag", inplace=True)
Ry = Ry["autocorrelation"].to_numpy()

# print and plot GPAC table
table = GPAC_cal(Ry,8,8)
```

```
# according to GPAC table:
# possible non-seasonal AR order is 1
# possible non-seasonal MA order is 1
p = 1
q = 1

#%% --------------------------------------------------- SARIMA model -----
----------------------------------------------------
# SARIMA (1,1,1) (0,1,1)24
my_non_seasonal_order = (1,1,1)
my_seasonal_order = (0,1,1,24)

model = SARIMAX(train, order = my_non_seasonal_order, seasonal_order =
my_seasonal_order, measurement_error=True)
model_fit = model.fit()

#%% ------------------------------------------------------ Forecasting ---
--------------------------------------------------
forecast = model_fit.forecast(len(test))

#%% ------------------------------------------------------ Prediction ----
----------------------------------------------
prediction = model_fit.predict(start = train.index[0], end =
train.index[len(train)-1])

#%% --------------------------------------------------- Plot predictions
and forecasts ----------------------------------
# plot the prediction
plt.figure()
plt.plot(train[-100:], label ="Training data")
plt.plot(prediction[-100:], label ="Fitted values")
plt.plot(test[:200], label="Testing data")
plt.plot(forecast[:200], label="Forecasted values")
plt.xlabel("Date")
plt.xticks(rotation=30)
plt.ylabel(target)
plt.title("Figure 19. NO2 concentration forecasting using
SARIMA(1,1,1)(0,1,1)24")
plt.legend(loc='best')
plt.show()

#%% ------------------------------------------ Evaluation metrics --------
--------------------------------------------------
# training set
residual = np.array(train) - np.array(prediction) # residuals
SSE_train = np.square(residual).sum()
MSE_train = np.square(residual).mean()
est_var_train = SSE_train/(len(train)-p-q-P-Q)
print("MSE of fitted values is: ", MSE_train)
print("Mean of residuals is: ", np.mean(residual))
print("The estimated variance of residuals is:", est_var_train )
print("Variance of residuals is:", np.var(residual))

# testing set
```

```
error = np.array(test) - np.array(forecast) # forecasted errors
SSE_test = np.square(error).sum()
MSE_test = np.square(error).mean()
est_var_test = SSE_test/(len(test)-p-q-P-Q)
print("MSE of testing set is: ", MSE_test)
print("Mean of forecasted errors is: ", np.mean(error))
print("The estimated variance of forecasted errors is:", est_var_test )
print("Variance of errors is:", np.var(error))

#%% ---------------------------------- ACF plot for residuals ------------
---------------------------------------------
# acf plot and Q value
h = 48
title = "Figure 20. ACF plot of residuals using SARIMA model"
ACF_plot(residual, 20, title)
Q_val = Q_val_cal(residual, h, len(test))
print("Q-value of residuals is: ", Q_val)
DOF = h - p - q - P - Q
alpha = 0.05
critical_Q = stats.chi2.ppf(1-alpha, DOF)
print("Critical Q-value = ", critical_Q)

#%% ----------------------------------- Summary of SARIMA ----------------
---------------------------------------------
print(model_fit.summary())
```

My functions

```
#%% --------------------------------------------- Import -----------------
--------------------------------------------------------
import pandas as pd
import matplotlib.pyplot as plt
import statsmodels.tsa.holtwinters as ets
import seaborn as sns
import numpy as np
from scipy import signal
import copy
import matplotlib.dates as mdates
from pandas.plotting import register_matplotlib_converters
register_matplotlib_converters()
from statsmodels.tsa.stattools import adfuller
import warnings
warnings.filterwarnings("ignore")

# correlation coefficient
def corr_cal(x,y):
    """
    Parameters
    ----------
    x : data
    y : data with the same length as x

    Returns : the correlation coefficient between x and y
    -------
```

```
    """
    if len(x) != len(y):
        print("Two datasets do not have the same length. Recheck the
datasets")
    else:
        length = len(x)
        # calculate means
        x_mean = np.mean(x)
        y_mean = np.mean(y)

        # calculate the numerator of the correlation coefficient function
        numerator = 0
        for i in range(0,length):
            numerator = numerator + (x[i]-x_mean)*(y[i]-y_mean)

        # calculate the denominator of the correlation coefficient
function
        X = 0
        Y = 0
        for i in range(0, length):
            X = X + (x[i]-x_mean)**2
            Y = Y + (y[i]-y_mean)**2
        denominator = (np.sqrt(X))*(np.sqrt(Y))

        # calculate the correlation coefficient
        r = numerator/denominator

        return r

# define a function to calculate the autocorrelation at lag k
def autocorrelation_cal(y,k):
    """
    calculate the autocorrelation between lagged values with distance k
    :param y: lagged values
    :param k: distance (k time units) that observations of a time series
are separated
    :return: autocorrelation
    """
    # initiate the numerator and denominator for the autocorrelation
    numerator = 0
    denominator = 0
    meanY = np.mean(y)

    # calculate the numerator for autocorrelation
    for i in range(k,len(y)):
        numerator = numerator + (y[i]-meanY)*(y[i-k]-meanY)

    # calculate the denominator for autocorrelation
    for i in range(len(y)):
        denominator = denominator + (y[i]-meanY)**2
    # calculate the autocorrelation
    t = numerator/denominator
```

```
    return t

# define a function to calculate autocorrelations for all lags (two sides)
def series_autocorrelation_cal(y, lags):
    """
    calculate the autocorrelation between lagged values with every
distance
    :param y: lagged values
    :return: a series of autocorrelations for the whole time domain
    """
    autocorrs = []
    nlags = []

    # we also consider the autocorrelation at (-k) which is equal to the
auto correlation at k
    for i in range(-lags,lags+1):
        k = abs(i)
        autocorrs.append(autocorrelation_cal(y,k))
        nlags.append(i)

    df = pd.DataFrame({"lag":nlags,"autocorrelation":autocorrs})

    return df

# define a function to plot ACF
def ACF_plot(Y,lags,title):
    """
    # plot the ACF of white noise
    :param Y: the lagged values
    :param lags: number of lags
    :param title: title for the plot
    :return: ACF plot
    """
    df = series_autocorrelation_cal(Y, lags)
    # create values for x axis using the number of lags
    x = np.arange(-lags,lags + 1,1)
    # create a list to store values of y
    y = []
    for i in x:
        # get the autocorrelation at lag i
        ac = df.loc[df["lag"] == i]["autocorrelation"]
        # aapend the autocorrelation to y
        y.append(ac.values[0])
    # plot the ACF

    plt.figure(figsize=(7,7))
    plt.stem(x,y)
    plt.title(title, fontsize = 13)
    plt.xlabel("Lags", fontsize =10)
    plt.ylabel("Magnitude",fontsize = 10)
    plt.grid()
    plt.show()

# define a function to forcast the data using different methods
```

```
def simple_forecast_ts(train, test, method ,period):
    """
    :param ytrain: training data
    :param ytest:  testing data
    :param method: simple forecast method
    :param period: only use for Holt-Winter Additive method to define the
seasonal_periods
    :return: dataframes containing prediction/forecast, errors and square
errors
    """
    T = len(train)  # number of observations in the training set
    h = len(test)  # number of observations to be forecast4ed in the
testing set

    ytrain = train.values
    ytest = test.values

    xtrain = train.index
    xtest = test.index

    ytrain_hat = [] # prediction
    ytest_hat = []  # forecast
    etrain = []  # prediction error
    etest = []  # forecast error
    setrain = []  # square of prediction error
    setest = []  # square of forecast error


    if method == 'Average': # using average method
        for i in range(0, 1):
            ytrain_hat.append(np.nan)
            etrain.append(np.nan)
            setrain.append(np.nan)
        for i in range(1,T):
            prediction = np.mean(ytrain[0:i])
            ytrain_hat.append(prediction)
            error = ytrain[i] - prediction
            etrain.append(error)
            setrain.append(error**2)

        for i in range(0,h):
            forecast = np.mean(ytrain)
            ytest_hat.append(forecast)
            error = ytest[i]-forecast
            etest.append(error)
            setest.append(error**2)


    elif method == 'Naive':  # naive method
        for i in range(0, 1):
            ytrain_hat.append(np.nan)
            etrain.append(np.nan)
            setrain.append(np.nan)
        for i in range(1, T):
```

```
            prediction = ytrain[i-1]
            ytrain_hat.append(prediction)
            error = ytrain[i]-prediction
            etrain.append(error)
            setrain.append(error**2)

        for i in range(0, h):
            forecast = ytrain[T-1]
            ytest_hat.append(forecast)
            error = ytest[i] - forecast
            etest.append(error)
            setest.append(error**2)

    elif method == 'Drift':  # drift method
        for i in range(0, 2):
            ytrain_hat.append(np.nan)
            etrain.append(np.nan)
            setrain.append(np.nan)
        for i in range(2, T):
            prediction = ytrain[i - 1] + (ytrain[i-1]-ytrain[0])/(i-1)
            ytrain_hat.append(prediction)
            error = ytrain[i] - prediction
            etrain.append(error)
            setrain.append(error**2)

        for i in range(0, h):
            forecast = ytrain[T - 1] + (ytrain[T-1]-ytrain[0])*(i+1)/(T-1)
            ytest_hat.append(forecast)
            error = ytest[i] - forecast
            etest.append(error)
            setest.append(error**2)

    elif method == "Simple Exponential Smoothing":  # simple exponential
smoothing method
        alpha = 0.5
        ytrain_hat.append(ytrain[0]) # the first observation is the
initial condition
        etrain.append(np.nan)  # for the initial condition, there is no
error
        setrain.append(np.nan)

        for i in range(1, T):
            prediction = alpha*ytrain[i - 1] + (1-alpha)*ytrain_hat[i-1]
            ytrain_hat.append(prediction)
            error = ytrain[i] - prediction
            etrain.append(error)
            setrain.append(error**2)

        for i in range(0, h):
            forecast = alpha*ytrain[T - 1] + (1-alpha)*ytrain_hat[T-1]
            ytest_hat.append(forecast)
            error = ytest[i] - forecast
            etest.append(error)
            setest.append(error**2)
```

```
    elif method == "Holt's Linear Multiplicative": # Holt linear using
multiplicative
        holt = ets.ExponentialSmoothing(ytrain, trend='multiplicative',
damped=True,seasonal=None).fit()
        ytrain_hat = holt.fittedvalues
        ytest_hat = holt.forecast(steps=h)

        for i in range(0,T):
            error = ytrain[i]-ytrain_hat[i]
            etrain.append(error)
            setrain.append(error**2)
        for i in range(0,h):
            error = ytest[i] - ytest_hat[i]
            etest.append(error)
            setest.append(error**2)

    elif method == "Holt's Linear":
        holt = ets.ExponentialSmoothing(ytrain, trend=None, damped=False,
seasonal=None).fit(smoothing_level=0.1)
        ytrain_hat = holt.fittedvalues
        ytest_hat = holt.forecast(steps=h)

        for i in range(0, T):
            error = ytrain[i] - ytrain_hat[i]
            etrain.append(error)
            setrain.append(error ** 2)
        for i in range(0, h):
            error = ytest[i] - ytest_hat[i]
            etest.append(error)
            setest.append(error**2)



    elif method == "Holt-Winter":
        holt = ets.ExponentialSmoothing(train, seasonal_periods=period,
trend='add', damped_trend=True, seasonal='additive')
        holt = holt.fit(smoothing_level=0.1,smoothing_seasonal=0.2,
smoothing_trend=None)
        ytrain_hat = holt.fittedvalues
        ytrain_hat = ytrain_hat.values
        ytest_hat = holt.forecast(steps=h)
        ytest_hat =ytest_hat.values


        for i in range(0, T):
            error = ytrain[i] - ytrain_hat[i]
            etrain.append(error)
            setrain.append(error**2)
        for i in range(0, h):
            error = ytest[i] - ytest_hat[i]
            etest.append(error)
            setest.append(error**2)
```

```
    else:
        print("Method is not applicable")

    # create dataframes containing results after predicting the training
data and forecasting the testing data
    df_train = pd.DataFrame({"time": xtrain , "y_t": ytrain,
"hat_y_t":ytrain_hat,
                             "error": etrain, "square_error":setrain})

    df_test = pd.DataFrame({"time": xtest, "y_t": ytest, "hat_y_t":
ytest_hat,
                             "error": etest, "square_error": setest})

    return df_train, df_test

# Q value of Box-pierce test
def Q_val_cal(y, lags, T):  # calculating the Q-value in Box-Pierce test
    """
    :param y: residuals
    :param lags: number of lags
    :param T: size of training set
    :return: Q-value of Box-Pierce test
    """
    Q = 0
    for i in range(1, lags + 1):
        r = autocorrelation_cal(y, i)
        Q = Q + (r**2)*T

    return Q

# define a function to plot yearly data and the forecast values
def forecasting_plot(train,test, method, xlabel, ylabel, interval,
dateformat, period):
    """
    :param train: training set
    :param test: testing set
    :param method: forecasting method
    :param xlabel: x axis label
    :param ylabel: y axis label
    :param interval: the difference between the ticks on the x axis
    :param dateformat: the format of tick labes on the x axis
    :param period: seasonal period for Holt-Winter additive method
    :return: plot of training, testing data and forecasts
    """
    prediction = simple_forecast_ts(train, test, method, period)[0]
    forecast = simple_forecast_ts(train, test, method, period)[1]

    # plot the training, testing sets and forecasting values
    fig, ax = plt.subplots()
    ax.plot(train[-100:], label = "Training data")
    ax.plot(train[-100:].index, prediction["hat_y_t"][-100:],
label="Predictions")
    ax.plot(test[:200], label ="Testing data")
```

```
    ax.plot(test[:200].index, forecast["hat_y_t"][:200],
label="Forecasts")

    ax.xaxis.set_tick_params(reset=True)
    ax.xaxis.set_major_locator(mdates.DayLocator(interval=interval))
    ax.xaxis.set_major_formatter(mdates.DateFormatter(dateformat))
    plt.setp(ax.get_xticklabels(), rotation=40, fontsize=10)
    plt.legend(loc="best")
    plt.title(method + " method forecasts", fontsize = 13)
    plt.xlabel(xlabel, fontsize = 13)
    plt.ylabel(ylabel, fontsize = 13)
    plt.show()


# calculate the MSE, variance, Q value and correlation coefficient
def stats(train, test, method, period):
    """
    :param train: training set
    :param test: testing set
    :param method: forecasting method
    :return: print MSE of forecasts, var of prediction, var of forecast,
Q-value
            and correlation coefficient between forecast errors and
testing data
    """
    prediction = simple_forecast_ts(train, test, method, period)[0]
    forecast = simple_forecast_ts(train, test, method, period)[1]

    # Mean square of errors
    MSE_test = np.mean(forecast["square_error"])
    MSE_train = np.mean(prediction["square_error"])

    # variance of prediction and forecast errors
    pred_var = np.var(prediction["error"])
    forecast_var = np.var(forecast["error"])


    # Q value
    lags = 48
    res = prediction["error"][3:]
    res = res.reset_index(drop=True)
    T = len(res)
    Q = Q_val_cal(res, lags, T)

    # correlation coefficient between forecast errors and the test set
    r = corr_cal(forecast["error"],test.values)

    print("\n")
    print("The mean of residuals is:", np.mean(res))
    print("The MSE of predictions using " + method + " is:
{:.4f}".format(MSE_train))
    print("The MSE of forecasts using " + method + " is:
{:.4f}".format(MSE_test))
```

```
    print("The variance of prediction errors using " + method + " is:
{:.4f}".format(pred_var))
    print("The variance of forecast errors using " + method + " is:
{:.4f}".format(forecast_var))
    print("The Q-value of residuals using " + method + " is:
{:.4f}".format(Q))
    print("The correlation coefficient between forecast errors and the
testing data using " + method + " is: {:.4f}".format(r))


# define a function to plot ACF for forecast errors
def ACF_error(method, train, test, period):
    prediction = simple_forecast_ts(train, test, method, period)[0]
    res = prediction["error"][3:]
    res = res.reset_index(drop=True)
    T = len(res)
    lags = 20
    ACF_plot(res, lags, "The ACF plot of residuals using " + method)


# define the ADF-test calculation
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))

# differencing (default 1st differencing
def differencing(y, lag=1):
      diff = np.zeros(len(y)-lag)
      for i in range(lag, len(y)):
            diff[i-lag] = y[i] - y[i - lag]
      return diff

# define a function to calculate phi(j,kk) component of GPAC table
def phi_cal(Ry, k, j):
    num = []
    den = []
    for i_r in np.arange(j, j + k):
        r_num = []
        r_den = []
        for i_c in np.arange(i_r + 1 - k, i_r + 1):
            index = abs(i_c)
            r_num.append(Ry[index])
            r_den.append(Ry[index])

        # reversing the list
        r_num = r_num[::-1]
        r_num[-1] = Ry[i_r + 1] # replace last value of r_num

        r_den = r_den[::-1]
```

```
        # appending to num and den
        num.append(r_num)
        den.append(r_den)


    # convert to matrix
    num = np.array(num)
    den = np.array(den)
    num = np.float64(np.linalg.det(num))
    den = np.linalg.det(den)

    if abs(den) > 1e-6:
        phi = float("{:.3f}".format(num/den))
    else:
        phi = float("inf")

    return phi

# define a function to calculate GPAC table
def GPAC_cal(Ry, K, J):

    phi = np.zeros(shape=(J+1,K))
    for k in range(1,K+1):
        for j in range(J+1):
            phi[j,k-1] = phi_cal(Ry,k,j)
    table  = pd.DataFrame(data=phi,
                          index=[i for i in range(J+1)],
                          columns=[i for i in range(1,K+1)])


    ax = sns.heatmap(table, vmin=-1, vmax=1, center=0,
                     cmap=sns.diverging_palette(20, 220, n=200),
                     annot=True,fmt=".3f")
    ax.set_title("Generalized Partial Autocorrelation (GPAC) Table")
    plt.xlabel("k")
    plt.ylabel("j")
    plt.show()

    print("\nGPAC table:")
    print(table)

# Levenberg-Marquardt estimation
def LME(y,na, nb, nepoch):
    var_e = 0.0
    cov_theta = 0.0
    N = len(y)
    n = na + nb
    #y = y - np.mean(y)

    l_max = max(na,nb)
    #print(l_max)
    num = np.zeros(l_max + 1)
```

```
den = np.zeros(l_max + 1)
num[0] = 1
den[0] = 1

#print(num)
#print(den)

delta = 1e-6
epsilon = 1e-3
u = 0.01

# simulating errors
sys = (den, num, 1)
_, e = signal.dlsim(sys, y)
SSE = np.dot(e.T, e)
running_SSE = []

for epoch in range(nepoch):

    # step 1
    X = np.zeros(shape=(N, n))
    for i in range(1, na + 1):
        den_temp = copy.deepcopy(den)
        den_temp[i] = den[i] + delta
        sys = (den_temp, num, 1)
        _, e_theta = signal.dlsim(sys, y)
        X[:, i - 1] = (e - e_theta)[:,0] / delta

    for i in range(1, nb + 1):
        num_temp = copy.deepcopy(num)
        num_temp[i] = num[i] + delta
        sys = (den, num_temp, 1)
        _, e_theta = signal.dlsim(sys, y)
        X[:, i + na - 1] =(e - e_theta)[:,0] / delta

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

    # step 2
    I = np.identity(n)
    delta_theta = np.dot(np.linalg.inv(A + u * I), g)

    # update coefficients
    den_new = copy.deepcopy(den)
    num_new = copy.deepcopy(num)
    for i in range(1,na+1):
        den_new[i] = den[i] + delta_theta[:na, :][i-1]

    for i in range(1,nb+1):
        num_new[i] = num[i] + delta_theta[na:, :][i-1]

    sys = (den_new, num_new, 1)
    _, e_new = signal.dlsim(sys, y)
    SSE_new = np.dot(e_new.T, e_new)
```

```
        running_SSE.append(SSE_new[0,0])

        if SSE_new < SSE:
            if np.linalg.norm(delta_theta,2) < epsilon:
                den = den_new
                num = num_new
                e = e_new
                var_e = SSE_new/(N-n)
                cov_theta = var_e*np.linalg.inv(A)
                break
            else:
                u = u/10
                den = den_new
                num = num_new
                e = e_new
                SSE=SSE_new

        else:
            u = u*10
            #print(u)
            if u > 1e10:
                print("Errors in Program!")
                break

        if epoch == nepoch-1:
            print("Errors in Programs!")



    if na > 0:
        for i in range(1,1+na):
            print("AR process estimated parameter a{:} = {:.4f}".format(i,
den[i]))
    if nb > 0:
        for i in range(1,1+nb):
            print("MA process estimated parameter b{:} = {:.4f}".format(i,
num[i]))


    return den, num, running_SSE, var_e, cov_theta


# 1-step ahead prediction
def one_step_ARMA(y_train, a, b):
    y_train_pred = np.zeros(len(y_train))
    e = np.zeros(len(y_train))
    na = len(a) - 1
    nb = len(b) - 1
    for i in range(len(y_train)-1):
        sum_ar = 0
        sum_ma = 0
        for j in range(1,na+1):
            if (i - j + 1 > 0) or (j - j +1) == 0:
                sum_ar = sum_ar + y_train[i-j+1]*a[j]
```

```
        for k in range(1,nb+1):
            if (i - k + 1 > 0) or (i - k +1) == 0:
                sum_ma = sum_ma + b[k]*(y_train[i-k+1]-y_train_pred[i-
k+1])
        y_train_pred[i+1] = -sum_ar + sum_ma


        e = y_train - y_train_pred

    return  y_train_pred, e


# h-step prediction (h>1)
def h_step_ARMA(y_train, y_train_pred, length_test, a, b):
    y_test_pred = np.zeros(length_test)
    na = len(a) - 1
    nb = len(b) - 1

    t = len(y_train) - 1
    for h in range(1,length_test + 1):
        sum_ar = 0
        sum_ma = 0
        for i in range(1,na+1):
            if h - i == 1:
                sum_ar = sum_ar + a[i]*y_train_pred[t]
            elif h - i > 1:
                sum_ar = sum_ar + a[i]*y_test_pred[h-i-2]
            else:
                k = i - h
                sum_ar = sum_ar + a[i]*y_train[t-k]
        for i in range(1,nb+1):
            if h - i > 0:
                sum_ma = sum_ma
            else:
                k = i - h
                sum_ma = sum_ma + b[i]*(y_train[t-k]-y_train_pred[t-k])

        y_test_pred[h-2] = -sum_ar + sum_ma

    return y_test_pred

# strength of trend and seasonality
def ts_strength(S, R, T):
    Ft = max([0, 1 - np.var(R)/np.var(T+R)])
    Fs = max([0, 1 - np.var(R)/np.var(S+R)])
    return Ft, Fs
```