Time series Analysis and Modeling DATS 6313

Final Term Project

Air Pollution Forecasting

Instructor: Reza Jafari

Author: Tanmay Vivek Kshirsagar

Date: 05/10/2023

Table of Contents

Table of Figures and Tables

Abstract

Air pollution is a major environmental concern affecting public health and the quality of life in many regions worldwide. To mitigate its adverse effects, accurate and timely forecasting of air pollution levels is essential. Time series analysis and modeling techniques have emerged as valuable tools for predicting air pollution levels, enabling policymakers, city planners, and the public to take proactive measures.

In this report, a comprehensive analysis of air pollution forecasting using time series analysis and modeling is presented. The process involves dataset description and preprocessing, including data cleaning techniques for time series datasets. Stationarity is checked and addressed using ADF-test, KPSS-test, and rolling mean and variance analysis. Time series decomposition is performed to approximate trend and seasonality components. Different models, such as Holt-Winters, base models (average, naive, drift, simple, exponential smoothing), multiple linear regression, ARMA, ARIMA, and SARIMA, are developed and evaluated. Additionally, a deep learning approach using a multivariate LSTM model is employed. The best-performing model is selected based on comparative analysis, and h-step ahead predictions are made to assess forecast accuracy.

1. Introduction

This report provides a comprehensive overview of the time series analysis and modeling process for air pollution forecasting. It begins with a description of the dataset, including the independent and dependent variables. The dataset is preprocessed, employing techniques for cleaning missing observations in time series data. A plot of the dependent variable over time is generated, along with observations. The autocorrelation function (ACF) and partial autocorrelation function (PACF) of the dependent variable are analyzed, and a correlation matrix with Pearson's correlation coefficient is constructed.
The stationarity of the dependent variable is examined through ADF-test, KPSS-test, and rolling mean and variance analysis. Time series decomposition techniques are applied to approximate the trend and seasonality components. The strength of the trend and seasonality is determined.
Several modeling approaches are explored. Holt-Winters method and base models like average, naive, drift, simple, and exponential smoothing are utilized for h-step prediction. Feature selection and elimination techniques are employed, such as backward stepwise regression, SVD, and condition number analysis. A multiple linear regression model is developed, and its accuracy is evaluated through regression analysis, one-step ahead prediction, hypothesis tests (F-test, t-test), and metrics such as AIC, BIC and RMSE. The Q-value, and variance and mean of the residuals are also assessed.
ARMA, ARIMA, and SARIMA models are developed, with the order determination process involving the GPAC table and discussion of the autocorrelation function. Parameter estimation for the ARMA model is carried out using the Levenberg-Marquardt algorithm, and diagnostic analysis is conducted, including confidence intervals, zero/pole cancellation, and chi-square tests. The derived model's bias and variance of errors are examined, and if ARIMA or SARIMA models prove to be a better fit, adjustments are made. Deep learning techniques are employed through a multivariate LSTM model, including h-step prediction. The final model is selected based on a comprehensive comparison of the performance of various models, including ARMA, ARIMA, SARIMA, and LSTM. A forecast function is developed for the chosen model (SARIMA), and multiple step-ahead predictions are made, with a comparison between predicted and true values.
The report concludes with a summary, limitations of the final model, and suggestions for other potential models that may improve performance.

2. Description of the dataset

This dataset contains of attributes related to weather conditions such as Dew, Temperature, Pressure, Wind Direction, Wind speed, Snow, Rain from January 2010 to December 2014 in a city in China. There is also a Date Time column called 'date' which has an hourly level frequency. The following is a comprehensive list of the features:

Table 2.1. Feature List

Column Name	**Description**
dew	Dew Point (Celsius Degree)
temp	Temperature (Celsius Degree)
press	Pressure (hPa)
wind_dir	Wind direction
wnd_spd	Wind speed (m/s)
snow	Hours of Snow Fall
rain	Hours of Rain

The target variable is column named 'pollution' which is in PM2.5 concentration. PM2.5 refers to particulate matter with a diameter of 2.5 micrometers or less. It is a type of air pollutant consisting of tiny particles suspended in the air. These particles are extremely small and can be composed of various substances, including dust, soot, smoke, organic chemicals, and metals.

	pollution	dew	temp	press	wnd_dir	wnd_spd	snow	rain
2010-01-02 00:00:00	129.0	-16	-4.0	1020.0	SE	1.79	0	0
2010-01-02 01:00:00	148.0	-15	-4.0	1020.0	SE	2.68	0	0
2010-01-02 02:00:00	159.0	-11	-5.0	1021.0	SE	3.57	0	0
2010-01-02 03:00:00	181.0	-7	-5.0	1022.0	SE	5.36	1	0
2010-01-02 04:00:00	138.0	-7	-5.0	1022.0	SE	6.25	2	0

Figure 2.1 Dataset snapshot

2.1. Pre-processing dataset

The dataset has a column called ‘date’ that is set as the index for performing the timeseries analysis.

	pollution	dew	temp	press	wnd_spd	\
count	43800.000000	43800.000000	43800.000000	43800.000000	43800.000000	
mean	94.013516	1.828516	12.459041	1016.447306	23.894307	
std	92.252276	14.429326	12.193384	10.271411	50.022729	
min	0.000000	-40.000000	-19.000000	991.000000	0.450000	
25%	24.000000	-10.000000	2.000000	1008.000000	1.790000	
50%	68.000000	2.000000	14.000000	1016.000000	5.370000	
75%	132.250000	15.000000	23.000000	1025.000000	21.910000	
max	994.000000	28.000000	42.000000	1046.000000	585.600000	

	snow	rain
count	43800.000000	43800.000000
mean	0.052763	0.195023
std	0.760582	1.416247
min	0.000000	0.000000
25%	0.000000	0.000000
50%	0.000000	0.000000
75%	0.000000	0.000000
max	27.000000	36.000000

Figure 2.1.1. basic statistics of the numerical features

Additionally, we can see from the following statistics that there are no null values present in the dataset.

#	Column	Non-Null Count	Dtype
0	pollution	43800 non-null	float64
1	dew	43800 non-null	int64
2	temp	43800 non-null	float64
3	press	43800 non-null	float64
4	wnd_dir	43800 non-null	int64
5	wnd_spd	43800 non-null	float64
6	snow	43800 non-null	int64
7	rain	43800 non-null	int64

dtypes: float64(4), int64(4)

Figure 2.1.2. datatype & non-null count of the features

For building models, Label encoding is performed on the column ‘wnd_dir’. The new labels are integers ranging from 1 to 4 for the four directions present in the dataset.

2.2. Plot of dependent variable over time

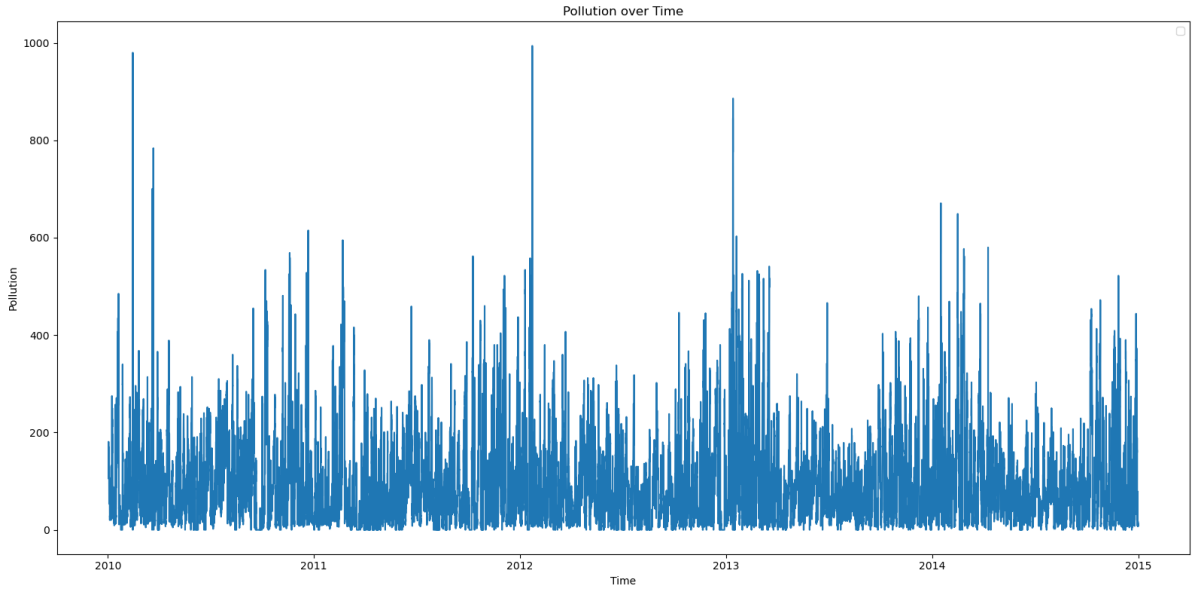


Figure 2.2. target variable over time

In the graph above, we can see that there is up & down trend over a certain period and fluctuations of pollution levels over time. The plot suggests that there might be a seasonality present in the dataset.

2.3. Correlation Matrix

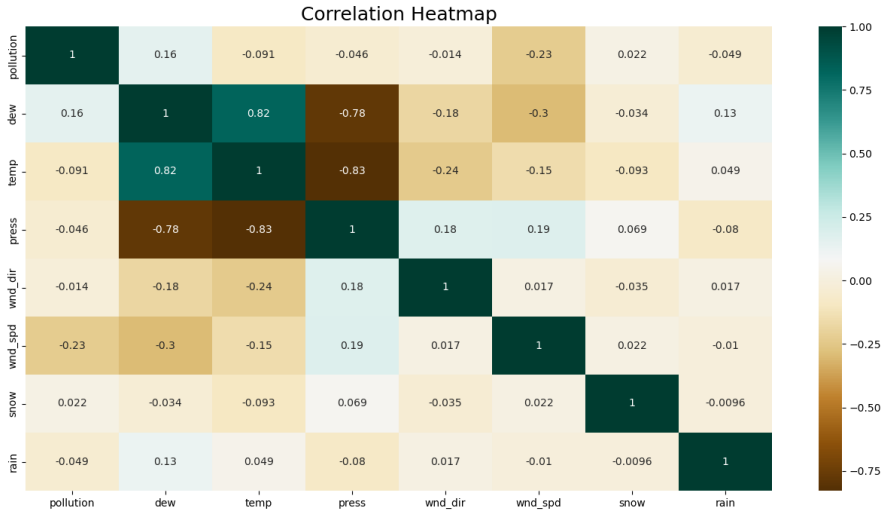


Figure 2.2. Correlation Heatmap plot

Looking at the correlation matrix, we can see that there is moderate to strong correlation between dew & temperature, dew & pressure, and temperature & pressure. But we can also observe that there is not much correlation between our target variable and the features.

3. Stationarity

A stationary time series is desirable for accurate forecasting as it allows for the application of various statistical techniques and models that assume constant statistical properties.

The ACF/PACF plot of the raw data shows that there are slight lumps around every lag=24 which Is visible more clearly in ACF plot with 100 lags.

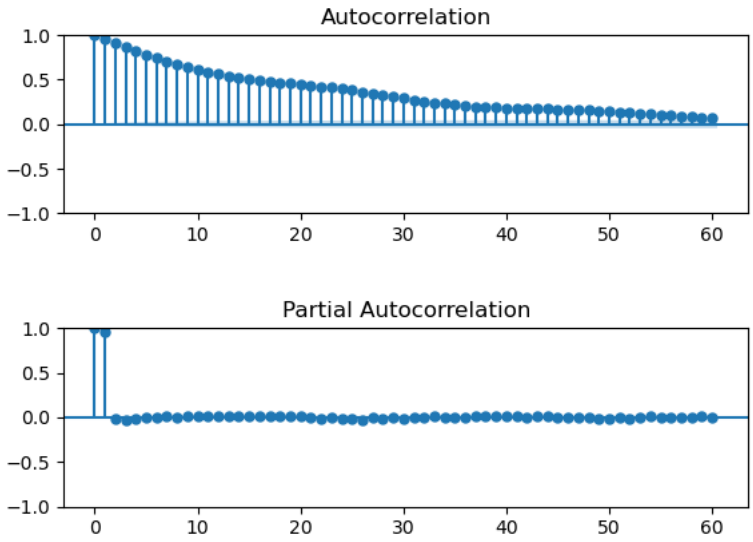


Figure 3.1.1. ACF/PACF plot raw data

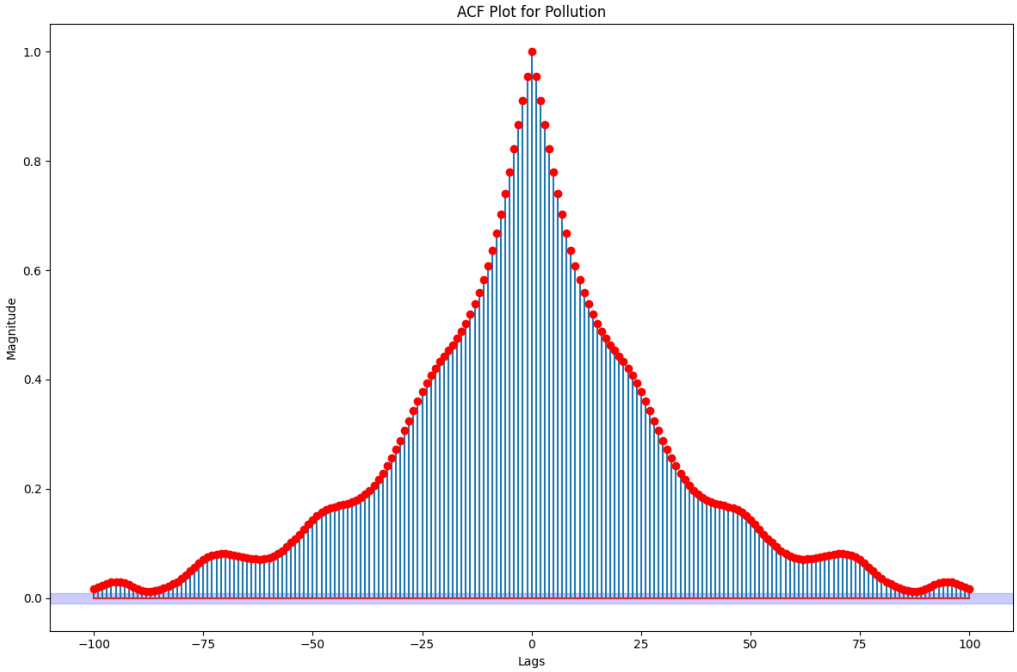


Figure 3.1.2. ACF plot (100 lags) raw data

We can also see that the rolling mean and rolling variance do not get stable as the number of samples increase, suggesting non-stationarity. Although, the ADF-test and KPSS-test suggest that the dataset is stationary, there is strong evidence of seasonality present in the data.

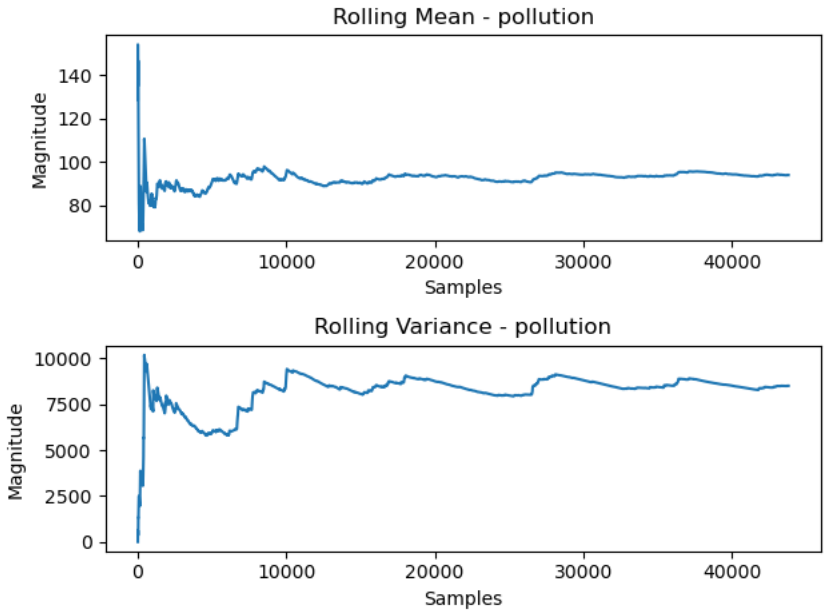


Figure 3.1.3. Rolling mean-Rolling variance plot raw data

```
ADF Statistic: -21.004109
p-value: 0.000000
Critical Values:
    1%: -3.430
    5%: -2.862
    10%: -2.567
```

```
Results of KPSS Test:
Test Statistic            0.078133
p-value                   0.100000
Lags Used               114.000000
Critical Value (10%)      0.347000
Critical Value (5%)       0.463000
Critical Value (2.5%)     0.574000
Critical Value (1%)       0.739000
dtype: float64
```

Figure 3.1.4. ADF test & KPSS test on raw data

Thus, a seasonal differencing with period=24 is performed to make the dataset stationary. We can see from the initial glance that the seasonal fluctuations have been removed from the data.

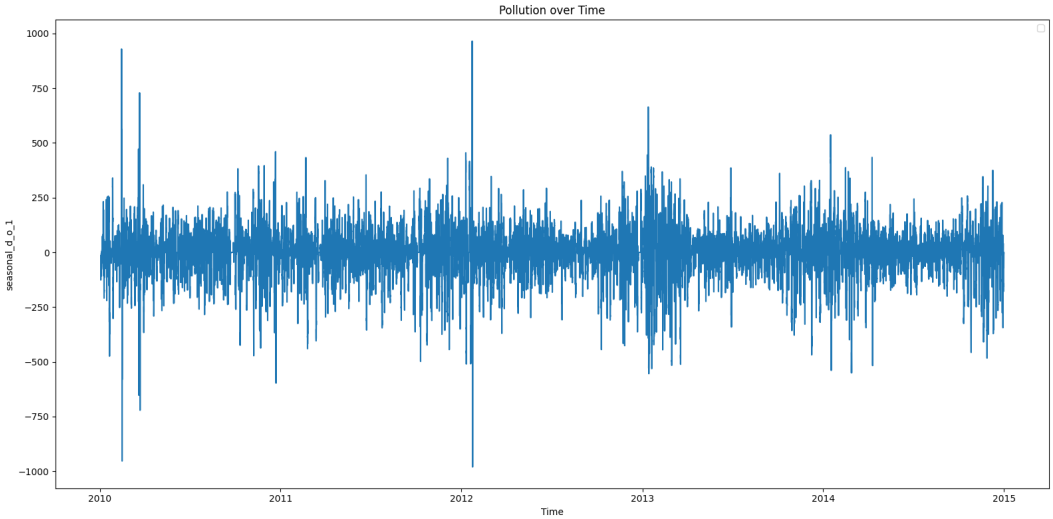


Figure 3.2.1. target variable over time after 1st transformation

Now, let's perform the stationarity check on this transformed data. In the ACF/PACF plot, the ACF still has some correlations present but seems to be slowly drifting towards zero. There might be need of further transformation.

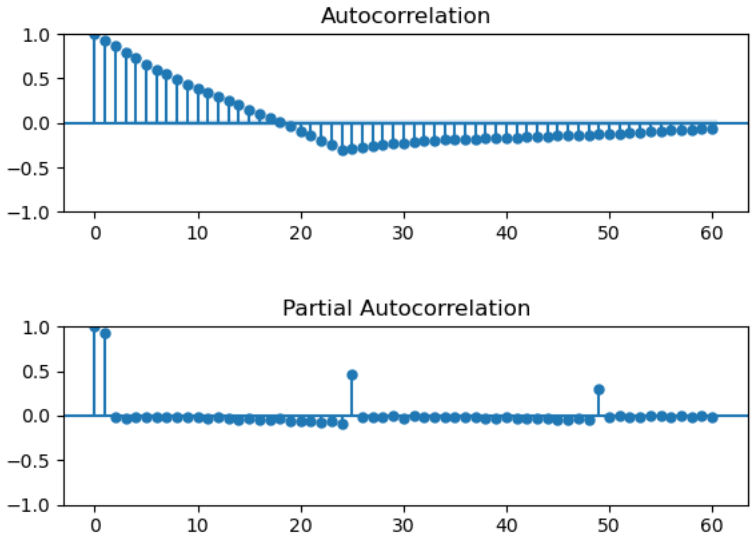


Figure 3.2.2. ACF/PACF plot 1st transformation

We can still see that the rolling variance does not get stable as the number of samples increase, suggesting non-stationarity. Although, the ADF-test and KPSS-test suggest that the dataset is stationary, there is a need of non-seasonal differencing transformation to the data.

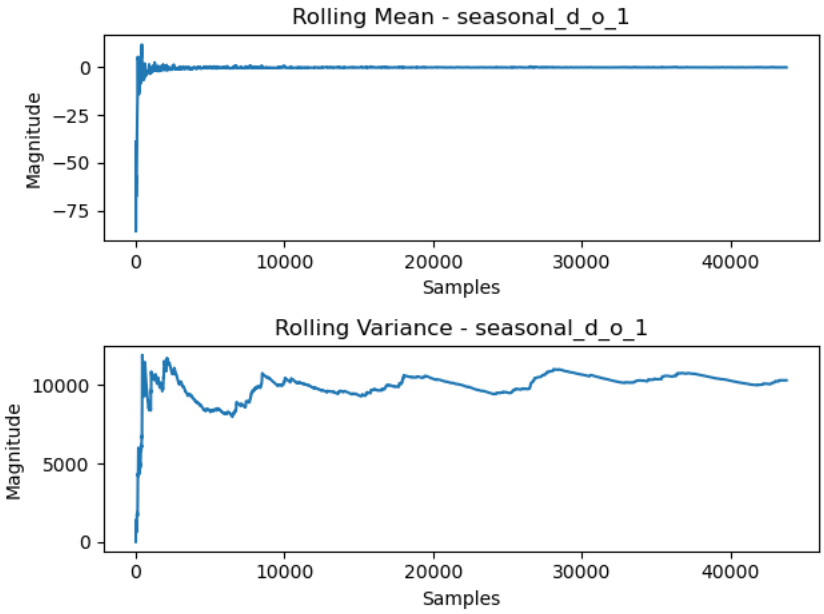


Figure 3.2.3. Rolling mean-Rolling variance plot 1st transformation

```
ADF Statistic: -31.066014
p-value: 0.000000
Critical Values:
    1%: -3.430
    5%: -2.862
    10%: -2.567
```

```
Results of KPSS Test:
Test Statistic            0.001666
p-value                   0.100000
Lags Used               108.000000
Critical Value (10%)      0.347000
Critical Value (5%)       0.463000
Critical Value (2.5%)     0.574000
Critical Value (1%)       0.739000
```

Figure 3.2.4. ADF test & KPSS test on 1st transformation

We can see from the plot of target variable vs time that the data seems to be stationary with almost constant statistics.

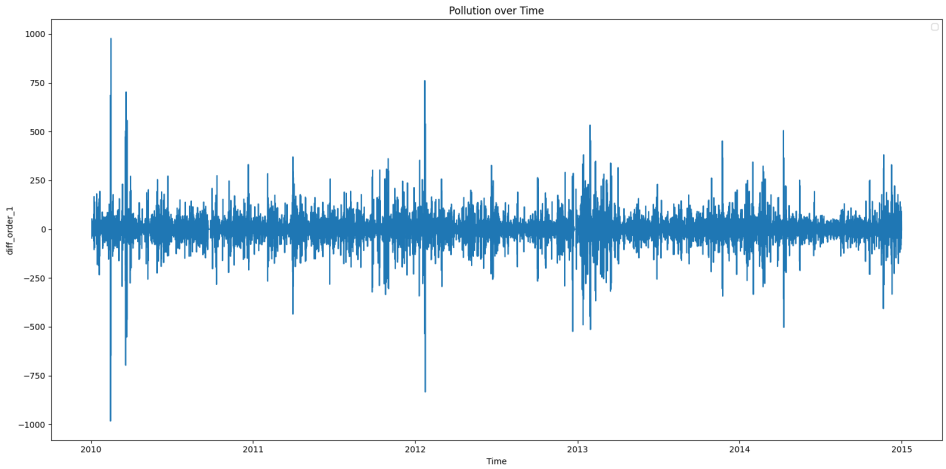


Figure 3.3.1. target variable over time stationary data

Now, let's perform the stationarity check on this transformed data. In the ACF/PACF plot, the ACF has clear cut-off at lag=24 and PACF has a tail-off at every multiple of lag=24. This means that there is a strong possibility that the order will have a MA=1 and AR=0.

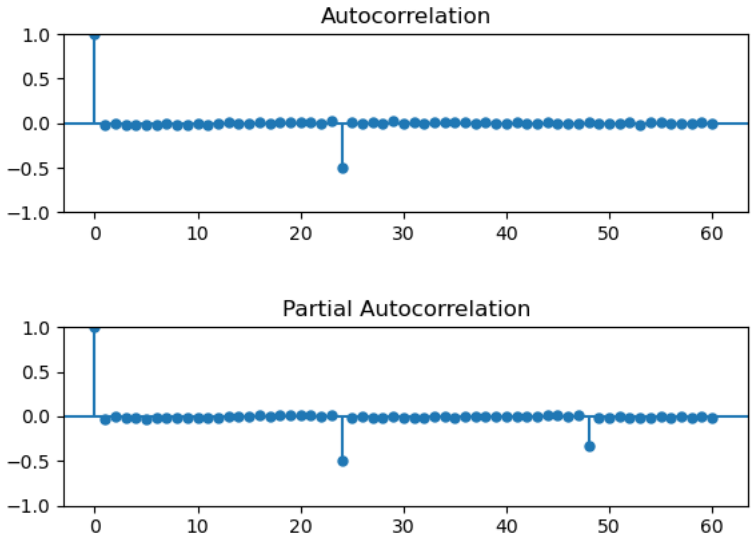


Figure 3.3.2. ACF/PACF plot stationary data

The rolling mean and rolling variance both get stable as the number of samples increase. Additionally, the ADF-KPSS test also indicate that data is stationary.

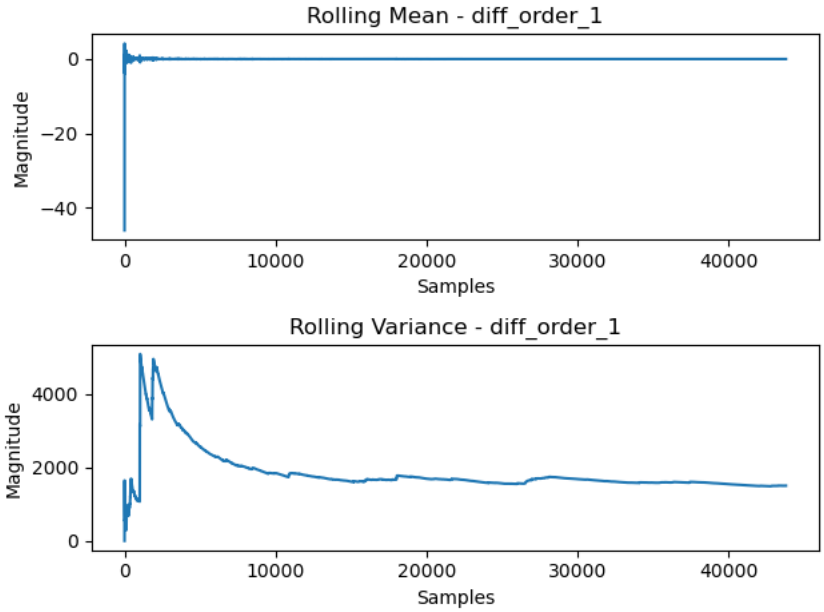


Figure 3.3.3. Rolling mean-Rolling variance plot stationary data

```
ADF Statistic: -47.967359
p-value: 0.000000
Critical Values:
    1%: -3.430
    5%: -2.862
    10%: -2.567
```

```
Results of KPSS Test:
Test Statistic            0.000916
p-value                   0.100000
Lags Used                76.000000
Critical Value (10%)      0.347000
Critical Value (5%)       0.463000
Critical Value (2.5%)     0.574000
Critical Value (1%)       0.739000
```

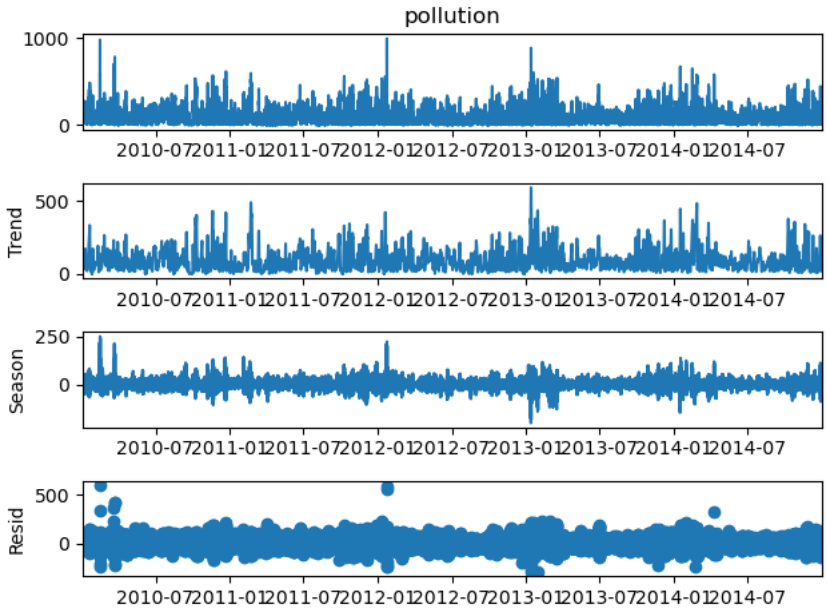
Figure 3.3.4. ADF test & KPSS test on 1st transformation

Thus, we can safely say that target variable becomes stationary after a seasonal differencing with period=24 and a non-seasonal differencing with order 1.

4. Time Series Decomposition

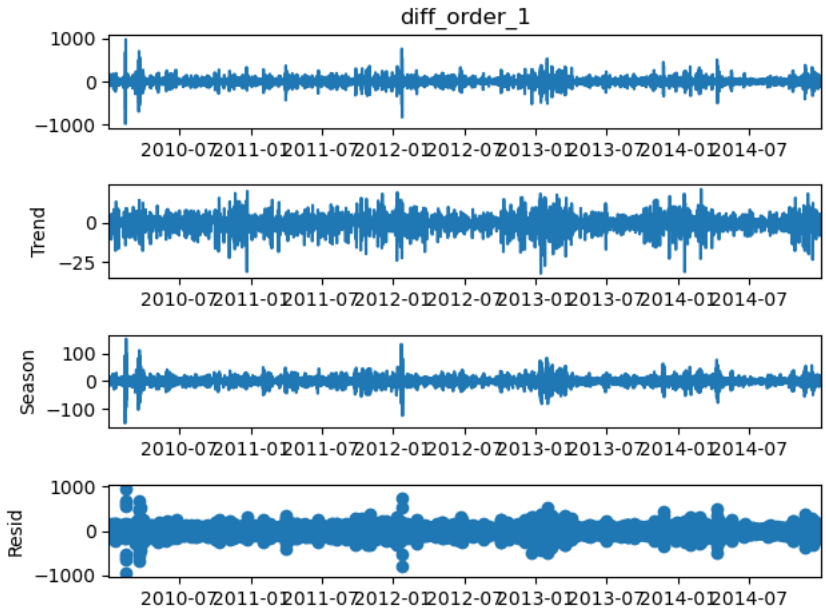
Time series decomposition is a technique used to separate a time series into its underlying components, such as trend, seasonality, and residual, to better understand and analyze the patterns and behaviors within the data.

Thus, let's apply the STL decomposition function on the raw data first and final transformed data to check for the strength of trend and strength of seasonality.




```
The strength of trend for pollution is 0.86.
The strength of seasonality for pollution is 0.407.
```

Figure 4.1. STL on raw data




```
The strength of trend for diff_order_1 is 0.036.
The strength of seasonality for diff_order_1 is 0.028.
```

Figure 4.2. STL on transformed data

Looking at the STL plots and strength of the trend and strength of seasonality, there is no need to perform detrend or removing seasonality as the transformation has already processed it.

5. Base models

Let's start by building some base models namely: Average, Naïve, Drift, SES and Holt-winters methods and check the results of each of them.

5.1. Average method

Average method is a simple forecasting method that predicts future values based on the average of past observations.

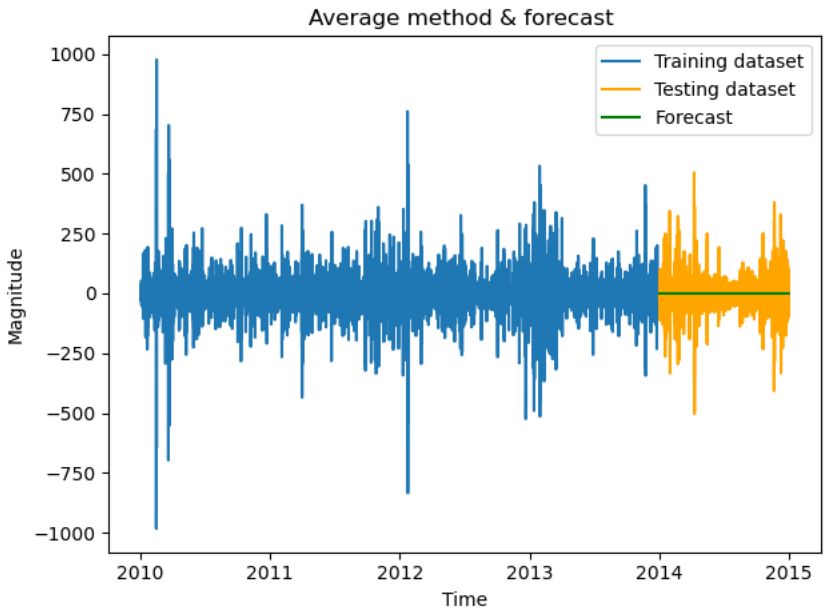


Figure 5.1.1. Average method forecast

The average method forecasts well on the test data by looking at the graph briefly. The MSE & variance of predicted and forecasted data seems to be moderate. The mean residual error is also approximately zero which is good. The Q-value seems to be high but we will keep it in mind while checking the other models. The ratio of variance of forecast errors versus variance of residual errors also seems to be good.

```
MSE Prediction data:  1581.65
MSE Forecasted data:  1170.43
Variance Prediction data:  1581.65
Variance Forecasted data:  1170.43
mean_res_train:  0.0
Q-value:  9202.88
var(forecast errors)/var(Residual errors): 0.74
```

Figure 5.1.2. Average method errors

5.2. Naive method

Naïve method is a basic forecasting method that assumes the future value will be the same as the most recent observation.

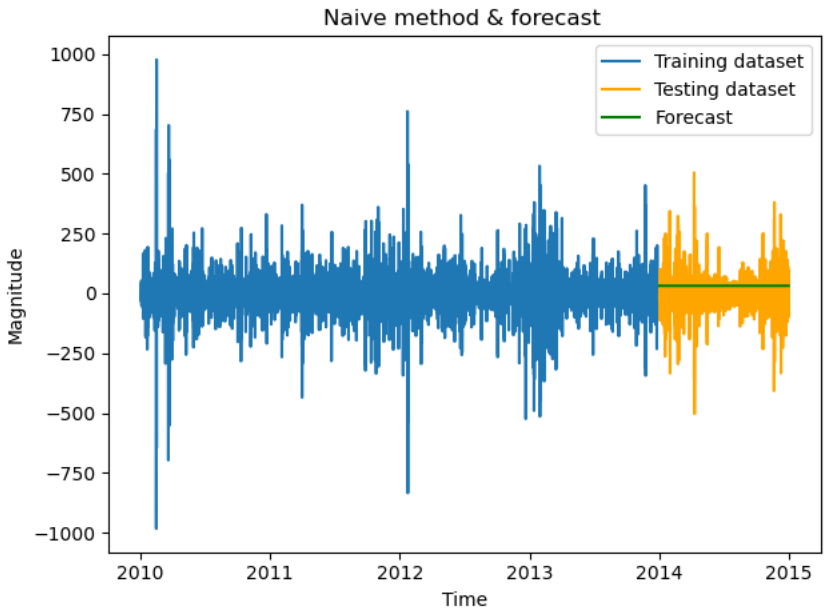


Figure 5.2.1. Naive method forecast

The Naive method forecasts well on the test data by looking at the graph briefly. The MSE & variance of predicted and forecasted data are higher than the Average model. The mean residual error is approximately zero which is good. The Q-value is too high than the average model. The ratio of variance of forecast errors versus variance of residual errors is also worse than the Average model.

```
MSE Prediction data:  3276.49
MSE Forecasted data:  2195.34
Variance Prediction data:  3276.49
Variance Forecasted data:  1170.43
mean_res_train:  0.0
Q-value:  23392.04
var(forecast errors)/var(Residual errors): 0.36
```

Figure 5.2.2. Naive method errors

5.3. Drift method

Drift method is a linear forecasting method that assumes a constant rate of change over time.

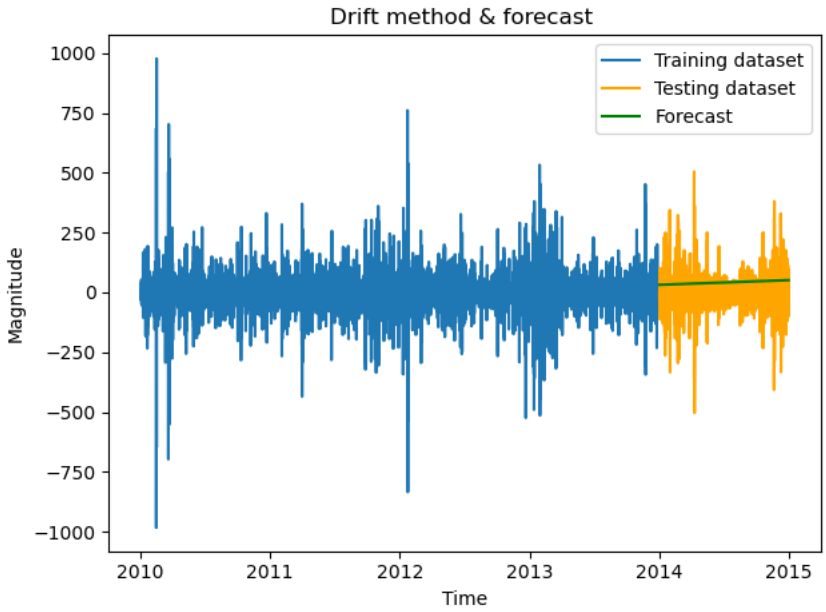


Figure 5.3.1. Drift method forecast

The Drift method does not forecast well on the test data by looking at the graph briefly. The MSE & variance of predicted and forecasted data are higher than the Average model. The mean residual error is approximately zero which is good. Again, the Q-value is too high than the average model. The ratio of variance of forecast errors versus variance of residual errors is also worse than the Average model.

```
MSE Prediction data:  3277.64
MSE Forecasted data:  2943.72
Variance Prediction data:  3277.64
Variance Forecasted data:  1201.68
mean_res_train:  -0.01
Q-value:  23392.54
var(forecast errors)/var(Residual errors): 0.37
```

Figure 5.3.2. Drift method errors

5.4. SES method

Simple Exponential Smoothing method is a forecasting method that assigns exponentially decreasing weights to past observations, giving more importance to recent data points. Here we will take the alpha value as 0.5.

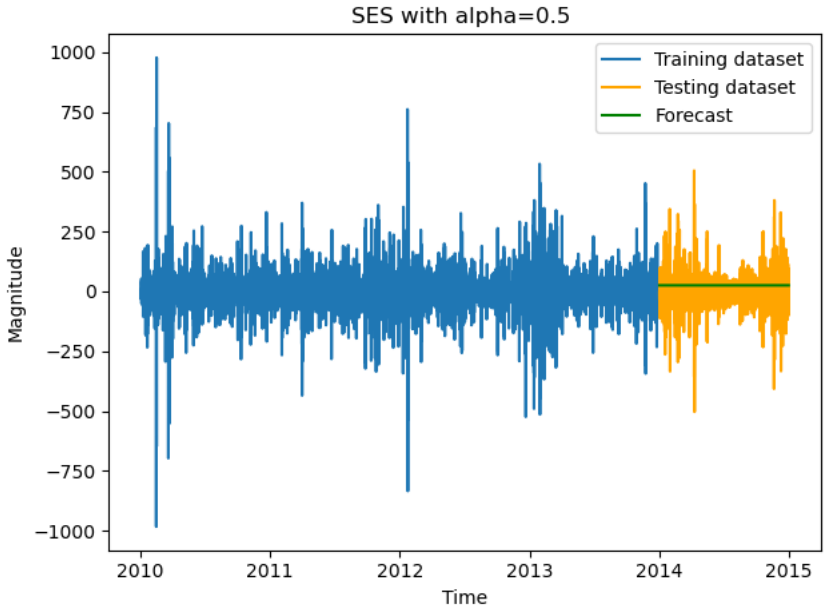


Figure 5.4.1. SES method forecast

The SES method forecasts well on the test data by looking at the graph briefly. The MSE & variance of predicted and forecasted data are higher than the Average model. The mean residual error is approximately zero which is good. Again, the Q-value is higher than the average model. The ratio of variance of forecast errors versus variance of residual errors is also worse than the Average model.

```
MSE Prediction data:  2160.22
MSE Forecasted data:  1826.43
Variance Prediction data:  2160.22
Variance Forecasted data:  1170.43
mean_res_train:  0.0
Q-value:  13736.18
var(forecast errors)/var(Residual errors): 0.54
```

Figure 5.4.2. SES method errors

5.5. Holt-Winters method

Holt-Winters method is a forecasting technique that incorporates trend, seasonality, and level components of a time series to provide more accurate predictions by using exponential smoothing with separate smoothing factors for each component.

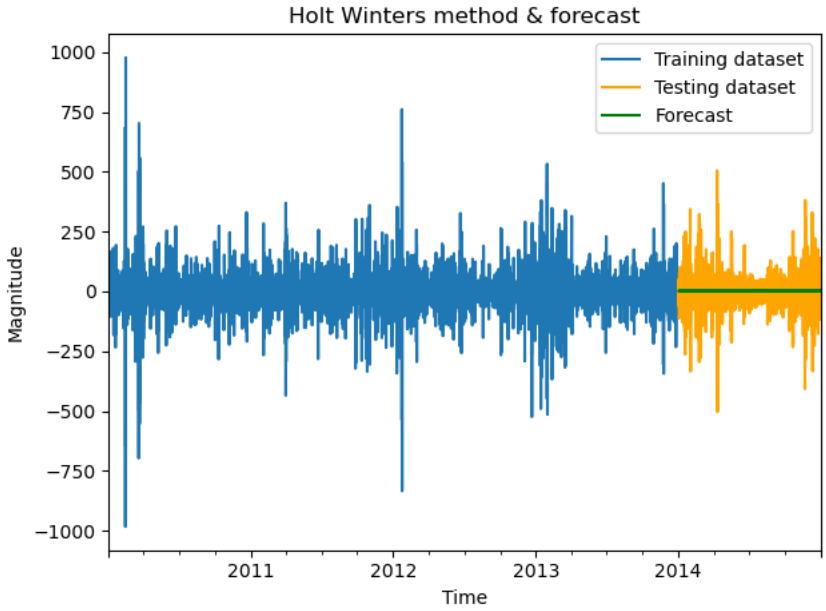


Figure 5.5.1. Holt-Winters method forecast

The Holt-Winters method forecasts well on the test data by looking at the graph briefly. The MSE & variance of predicted and forecasted data are good. The mean residual error is approximately zero which is good. The Q-value is also less. The ratio of variance of forecast errors versus variance of residual errors is also good.

```
MSE Prediction data:  1627.69
MSE Forecasted data:  1171.27
Variance Prediction data:  1627.69
Variance Forecasted data:  1170.74
mean_res_train:  -0.01
Q-value:  9188.84
var(forecast errors)/var(Residual errors): 0.72
```

Figure 5.5.2. Holt-Winters method errors

6. Feature selection/elimination

```
Singular values = [97868.71777248 36524.81118131 34847.5144482  33898.80350439
 29613.5147789   7517.33354556  4869.30476916]
Condition number is 4.48
        VIF variable
0  3.999558      dew
1  4.700797     temp
2  3.427424    press
3  1.066408  wnd_dir
4  1.141609  wnd_spd
5  1.020850     snow
6  1.031133     rain
```

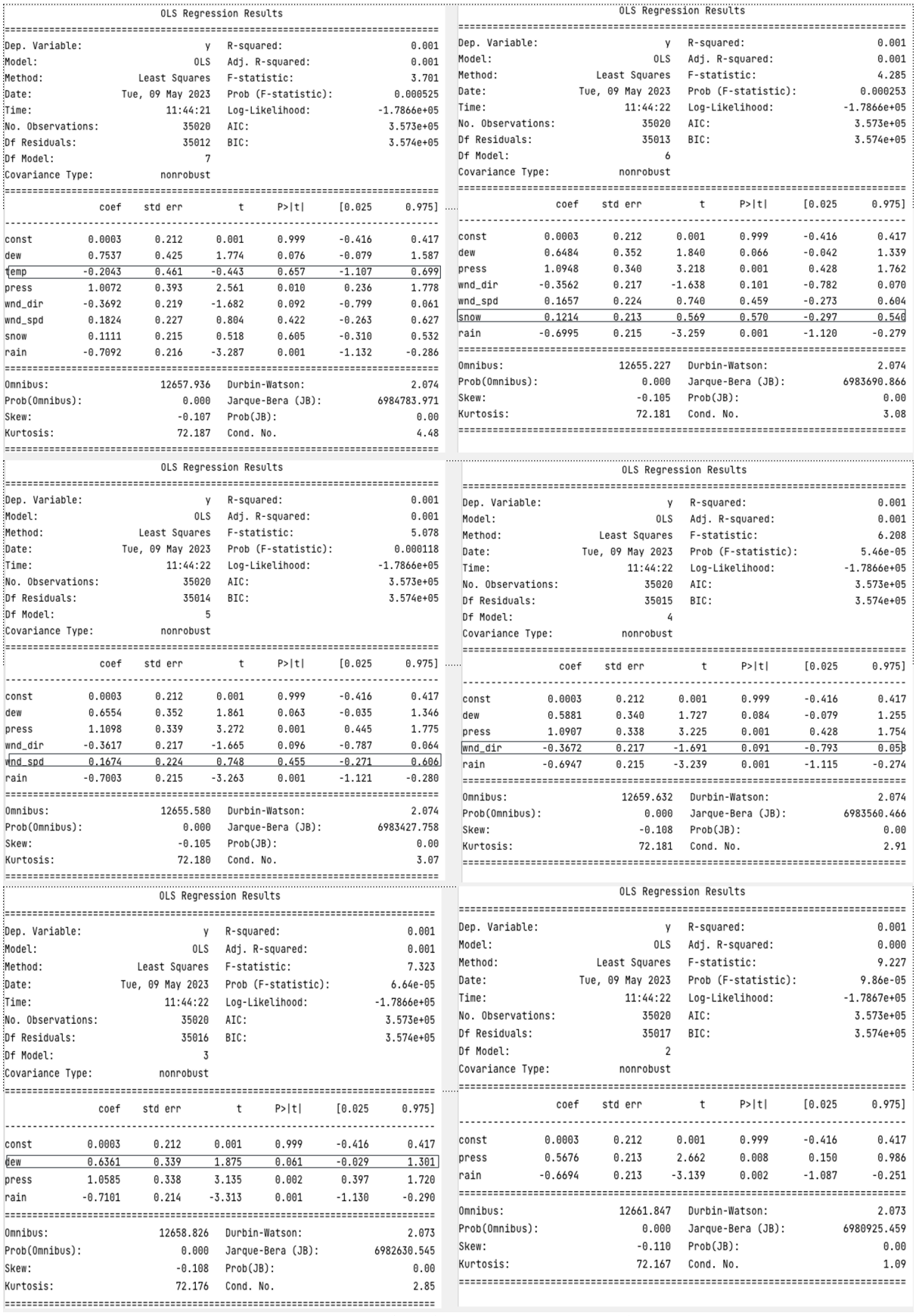
Figure 6.1. SVD and VIF

The singular values represent the importance or significance of each component in the data. Larger singular values indicate stronger contributions of the corresponding components to the overall variation in the data. The Singular values that we get suggest presence of collinearity.

The condition number helps assess the degree of multicollinearity among the predictor variables. The condition number that we get suggests that there is a weak degree of multicollinearity present.

VIF (Variance Inflation Factor) is a measure used to assess the severity of multicollinearity in a regression model, indicating the extent to which the variance of the estimated regression coefficients is inflated due to high correlation between predictor variables. The VIF values for the features confirm that there is no strong collinearity present in the data.

Next, backward stepwise regression is performed for feature selection. Backward Stepwise regression is a feature selection method that starts with a full model and iteratively removes insignificant variables based on statistical tests, aiming to find the most relevant predictors for a regression model.



OLS Regression Results

Dep. Variable:	y	R-squared:	0.001
Model:	OLS	Adj. R-squared:	0.001
Method:	Least Squares	F-statistic:	3.701
Date:	Tue, 09 May 2023	Prob (F-statistic):	0.000525
Time:	11:44:21	Log-Likelihood:	-1.7866e+05
No. Observations:	35020	AIC:	3.573e+05
Df Residuals:	35012	BIC:	3.574e+05
Df Model:	7		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
const	0.0003	0.212	0.001	0.999	-0.416	0.417
dew	0.7537	0.425	1.774	0.076	-0.079	1.587
temp	-0.2043	0.461	-0.443	0.657	-1.107	0.699
press	1.0072	0.393	2.561	0.010	0.236	1.778
wnd_dir	-0.3692	0.219	-1.682	0.092	-0.799	0.061
wnd_spd	0.1824	0.227	0.804	0.422	-0.263	0.627
snow	0.1111	0.215	0.518	0.605	-0.310	0.532
rain	-0.7092	0.216	-3.287	0.001	-1.132	-0.286

Omnibus:	12657.936	Durbin-Watson:	2.074
Prob(Omnibus):	0.000	Jarque-Bera (JB):	6984783.971
Skew:	-0.107	Prob(JB):	0.00
Kurtosis:	72.187	Cond. No.	4.48

OLS Regression Results

Dep. Variable:	y	R-squared:	0.001
Model:	OLS	Adj. R-squared:	0.001
Method:	Least Squares	F-statistic:	4.285
Date:	Tue, 09 May 2023	Prob (F-statistic):	0.000253
Time:	11:44:22	Log-Likelihood:	-1.7866e+05
No. Observations:	35020	AIC:	3.573e+05
Df Residuals:	35013	BIC:	3.574e+05
Df Model:	6		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
const	0.0003	0.212	0.001	0.999	-0.416	0.417
dew	0.6484	0.352	1.840	0.066	-0.042	1.339
press	1.0948	0.340	3.218	0.001	0.428	1.762
wnd_dir	-0.3562	0.217	-1.638	0.101	-0.782	0.070
wnd_spd	0.1657	0.224	0.740	0.459	-0.273	0.604
snow	0.1214	0.213	0.569	0.570	-0.297	0.540
rain	-0.6995	0.215	-3.259	0.001	-1.120	-0.279

Omnibus:	12655.227	Durbin-Watson:	2.074
Prob(Omnibus):	0.000	Jarque-Bera (JB):	6983690.866
Skew:	-0.105	Prob(JB):	0.00
Kurtosis:	72.181	Cond. No.	3.08

OLS Regression Results

Dep. Variable:	y	R-squared:	0.001
Model:	OLS	Adj. R-squared:	0.001
Method:	Least Squares	F-statistic:	5.078
Date:	Tue, 09 May 2023	Prob (F-statistic):	0.000118
Time:	11:44:22	Log-Likelihood:	-1.7866e+05
No. Observations:	35020	AIC:	3.573e+05
Df Residuals:	35014	BIC:	3.574e+05
Df Model:	5		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
const	0.0003	0.212	0.001	0.999	-0.416	0.417
dew	0.6554	0.352	1.861	0.063	-0.035	1.346
press	1.1098	0.339	3.272	0.001	0.445	1.775
wnd_dir	-0.3617	0.217	-1.665	0.096	-0.787	0.064
wnd_spd	0.1674	0.224	0.748	0.455	-0.271	0.606
rain	-0.7003	0.215	-3.263	0.001	-1.121	-0.280

Omnibus:	12655.580	Durbin-Watson:	2.074
Prob(Omnibus):	0.000	Jarque-Bera (JB):	6983427.758
Skew:	-0.105	Prob(JB):	0.00
Kurtosis:	72.180	Cond. No.	3.07

OLS Regression Results

Dep. Variable:	y	R-squared:	0.001
Model:	OLS	Adj. R-squared:	0.001
Method:	Least Squares	F-statistic:	6.208
Date:	Tue, 09 May 2023	Prob (F-statistic):	5.46e-05
Time:	11:44:22	Log-Likelihood:	-1.7866e+05
No. Observations:	35020	AIC:	3.573e+05
Df Residuals:	35015	BIC:	3.574e+05
Df Model:	4		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
const	0.0003	0.212	0.001	0.999	-0.416	0.417
dew	0.5881	0.340	1.727	0.084	-0.079	1.255
press	1.0907	0.338	3.225	0.001	0.428	1.754
wnd_dir	-0.3672	0.217	-1.691	0.091	-0.793	0.058
rain	-0.6947	0.215	-3.239	0.001	-1.115	-0.274

Omnibus:	12659.632	Durbin-Watson:	2.074
Prob(Omnibus):	0.000	Jarque-Bera (JB):	6983560.466
Skew:	-0.108	Prob(JB):	0.00
Kurtosis:	72.181	Cond. No.	2.91

OLS Regression Results

Dep. Variable:	y	R-squared:	0.001
Model:	OLS	Adj. R-squared:	0.001
Method:	Least Squares	F-statistic:	7.323
Date:	Tue, 09 May 2023	Prob (F-statistic):	6.64e-05
Time:	11:44:22	Log-Likelihood:	-1.7866e+05
No. Observations:	35020	AIC:	3.573e+05
Df Residuals:	35016	BIC:	3.574e+05
Df Model:	3		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
const	0.0003	0.212	0.001	0.999	-0.416	0.417
dew	0.6361	0.339	1.875	0.061	-0.029	1.301
press	1.0585	0.338	3.135	0.002	0.397	1.720
rain	-0.7101	0.214	-3.313	0.001	-1.130	-0.290

Omnibus:	12658.826	Durbin-Watson:	2.073
Prob(Omnibus):	0.000	Jarque-Bera (JB):	6982630.545
Skew:	-0.108	Prob(JB):	0.00
Kurtosis:	72.176	Cond. No.	2.85

OLS Regression Results

Dep. Variable:	y	R-squared:	0.001
Model:	OLS	Adj. R-squared:	0.000
Method:	Least Squares	F-statistic:	9.227
Date:	Tue, 09 May 2023	Prob (F-statistic):	9.86e-05
Time:	11:44:22	Log-Likelihood:	-1.7867e+05
No. Observations:	35020	AIC:	3.573e+05
Df Residuals:	35017	BIC:	3.574e+05
Df Model:	2		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
const	0.0003	0.212	0.001	0.999	-0.416	0.417
press	0.5676	0.213	2.662	0.008	0.150	0.986
rain	-0.6694	0.213	-3.139	0.002	-1.087	-0.251

Omnibus:	12661.847	Durbin-Watson:	2.073
Prob(Omnibus):	0.000	Jarque-Bera (JB):	6980925.459
Skew:	-0.110	Prob(JB):	0.00
Kurtosis:	72.167	Cond. No.	1.09

Figure 6.2. Backward Stepwise Regression results

As observed in fig 6.2., the features are removed one by one till the point where we get all the features as significant with a good AIC, BIC and Adjusted R-squared value. The sequence in which the columns are removed is ‘temp’=>’snow’ =>’wnd_spd’ =>’wnd_dir’ =>’dew’. There is not much change in AIC, BIC and the adjusted R-squared value. This means that we can use only two features ‘pressure’ and ‘rain’ to build the multiple linear regression model.

7. Multiple linear regression model

Multiple linear regression model is a statistical technique that examines the linear relationship between a dependent variable and multiple independent variables, allowing for the prediction and interpretation of the dependent variable based on the values of the independent variables.

The features selected from the previous analysis is used to build this model.

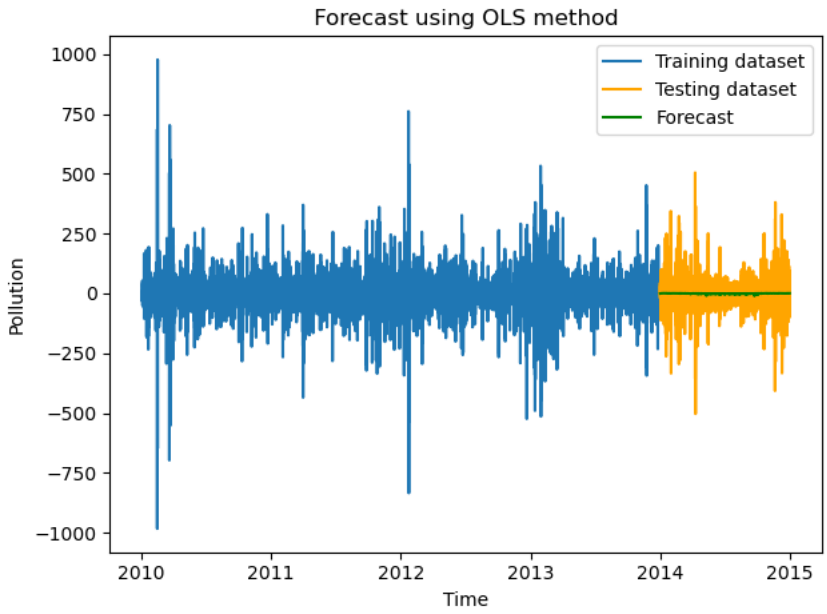


Figure 7.1. Multiple Linear Regression model forecast

The Multiple Linear Regression model forecasts well on the test data by looking at the graph briefly. The MSE & variance of predicted and forecasted data are good. The mean residual error is approximately zero which is good. The Q-value is also less. The ratio of variance of forecast errors versus variance of residual errors is also good. The AIC, BIC, RMSE and Adjusted R-squared values are not the best but considering the type of real-life data that is being considered, this model is performing well.

```
MSE Prediction data:  1581.01
MSE Forecasted data:  1167.68
Variance Prediction data:  1581.01
Variance Forecasted data:  1167.67
mean_res_train:  0.0
Q-value:  9200.03
var(forecast errors)/var(Residual errors): 0.74
```

Figure 7.2. Multiple Linear Regression model errors

OLS Regression Results

Dep. Variable:	y	R-squared:	0.001
Model:	OLS	Adj. R-squared:	0.000
Method:	Least Squares	F-statistic:	9.227
Date:	Tue, 09 May 2023	Prob (F-statistic):	9.86e-05
Time:	11:44:22	Log-Likelihood:	-1.7867e+05
No. Observations:	35020	AIC:	3.573e+05
Df Residuals:	35017	BIC:	3.574e+05
Df Model:	2		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
const	0.0003	0.212	0.001	0.999	-0.416	0.417
press	0.5676	0.213	2.662	0.008	0.150	0.986
rain	-0.6694	0.213	-3.139	0.002	-1.087	-0.251

Omnibus:	12661.847	Durbin-Watson:	2.073
Prob(Omnibus):	0.000	Jarque-Bera (JB):	6980925.459
Skew:	-0.110	Prob(JB):	0.00
Kurtosis:	72.167	Cond. No.	1.09

Figure 7.3. Multiple Linear Regression Summary

We can also see that ACF plot of residual errors tends to zero i.e. white noise except the lags at seasonal periods=24.

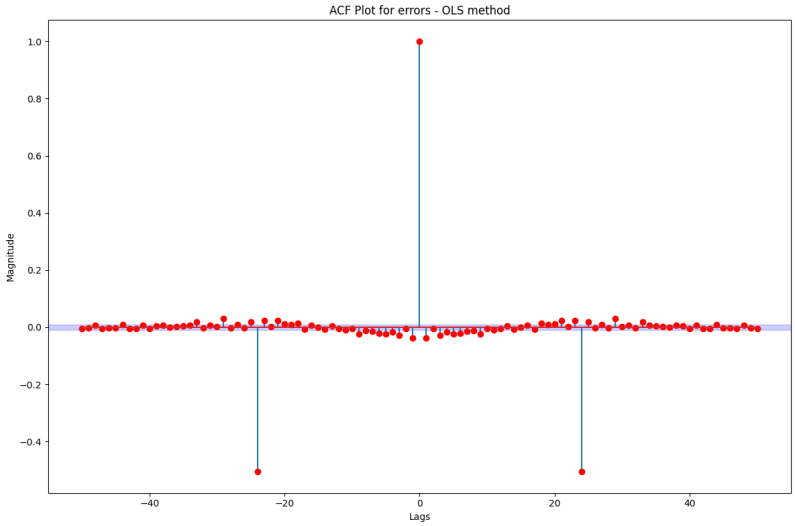


Figure 7.4. ACF of residuals

```
T-Test
const    0.999035
press    0.007768
rain     0.001694
dtype: float64

F-Test
9.863301880350688e-05
```

Figure 7.5. T-Test & F-Test

The t-test is used to determine if there is a significant difference between the means of two groups, helping to assess the significance of a relationship or the effectiveness of a treatment. The f-test compares the variances of two or more groups, allowing for the evaluation of differences in means among multiple groups or the overall significance of a regression model.

Thus, as the p-values for all the features from the T-test are lower than 0.05, we can say that these features are statistically significant. Also, the p-value for F-test is lower than 0.05 (conf interval=95%), we can say that the final model that we created is better than the base model.

8. SARIMA model

8.1. GPAC & SARIMA model comparison

GPAC (Generalized Partial Autocorrelation) is used to estimate the orders of autoregressive (AR) and moving average (MA) components in time series analysis, providing insights into the relationship between variables and aiding in model selection.

The GPAC in fig 8.1.1. is too small thus a zoomed version is provided in fig 8.1.2.

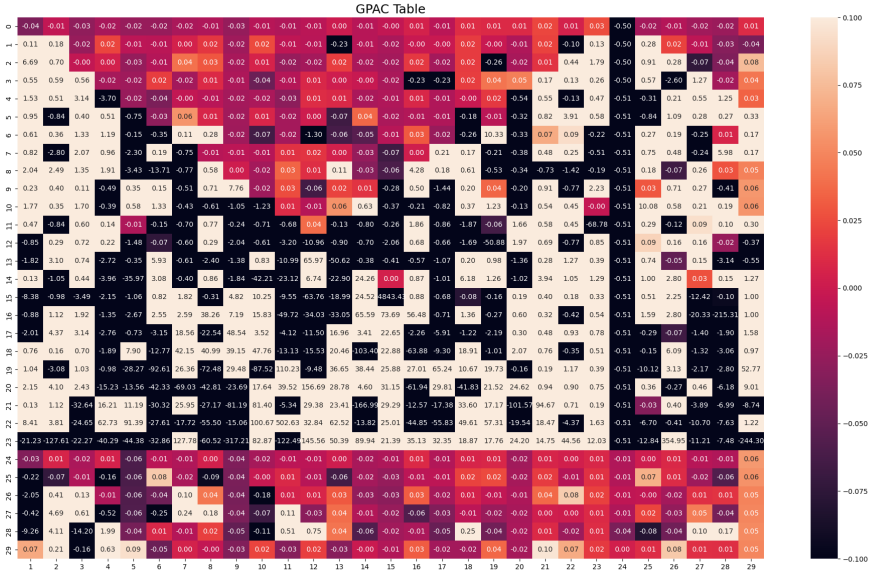


Figure 8.1.1. GPAC

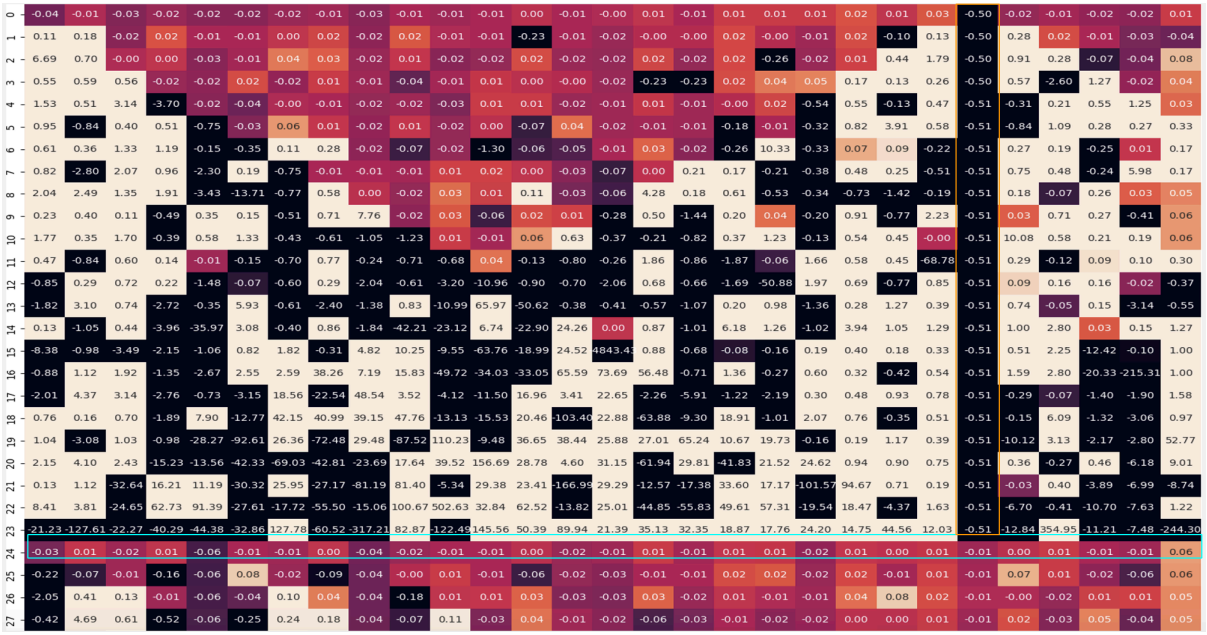


Figure 8.1.2. GPAC (zoomed)

Here, we can clearly observe that there can be multiple order combinations of AR & MA using the GPAC above. For this project, two orders were considered: one with MA=1 and AR=0 and one with AR=1 and MA=0.

SARIMAX Results

Dep. Variable:	diff_order_1	No. Observations:	35020
Model:	ARIMA(1, 0, 0, 24)	Log Likelihood	-173528.770
Date:	Tue, 09 May 2023	AIC	347063.539
Time:	15:31:30	BIC	347088.930
Sample:	01-03-2010 - 01-01-2014	HQIC	347071.627
Covariance Type:	opg		

	coef	std err	z	P>|z|	[0.025	0.975]
const	0.0003	0.122	0.002	0.998	-0.240	0.240
ar.S.L24	-0.5046	0.001	-529.103	0.000	-0.506	-0.503
sigma2	1179.2172	1.533	769.044	0.000	1176.212	1182.223

Ljung-Box (L1) (Q):	38.60	Jarque-Bera (JB):	6394422.69
Prob(Q):	0.00	Prob(JB):	0.00
Heteroskedasticity (H):	0.85	Skew:	-0.67
Prob(H) (two-sided):	0.00	Kurtosis:	69.19

Figure 8.1.3. SARIMA with AR=1, MA=0 and seasonal_period=24

SARIMAX Results

Dep. Variable:	diff_order_1	No. Observations:	35020
Model:	ARIMA(0, 0, [1], 24)	Log Likelihood	-166647.826
Date:	Tue, 09 May 2023	AIC	333301.652
Time:	12:34:15	BIC	333327.043
Sample:	01-03-2010	HQIC	333309.740
	- 01-01-2014		
Covariance Type:	opg		

	coef	std err	z	P>|z|	[0.025	0.975]
const	-5.535e-06	0.000	-0.011	0.991	-0.001	0.001
ma.S.L24	-0.9981	0.001	-1697.124	0.000	-0.999	-0.997
sigma2	792.7778	0.862	920.209	0.000	791.089	794.466

Ljung-Box (L1) (Q):	19.58	Jarque-Bera (JB):	22953640.55
Prob(Q):	0.00	Prob(JB):	0.00
Heteroskedasticity (H):	0.83	Skew:	-1.27
Prob(H) (two-sided):	0.00	Kurtosis:	128.40

Figure 8.1.4. SARIMA with AR=0, MA=1 and seasonal_period=24

Looking at the results both the SARIMA models, we can say that ARIMA(0,0,[1],24) performs better with lower AIC, BIC values. The ACF for both models indicate that the residual errors generated are white. This indicates that the derived model is not biased. Thus, this model has been considered for the further analysis.

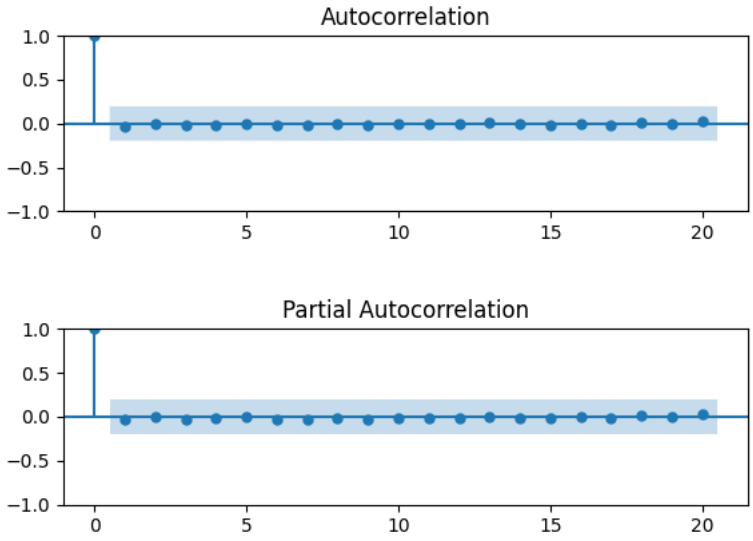


Figure 8.1.5. ACF/PACF plot of SARIMA with AR=1, MA=0 and seasonal_period=24

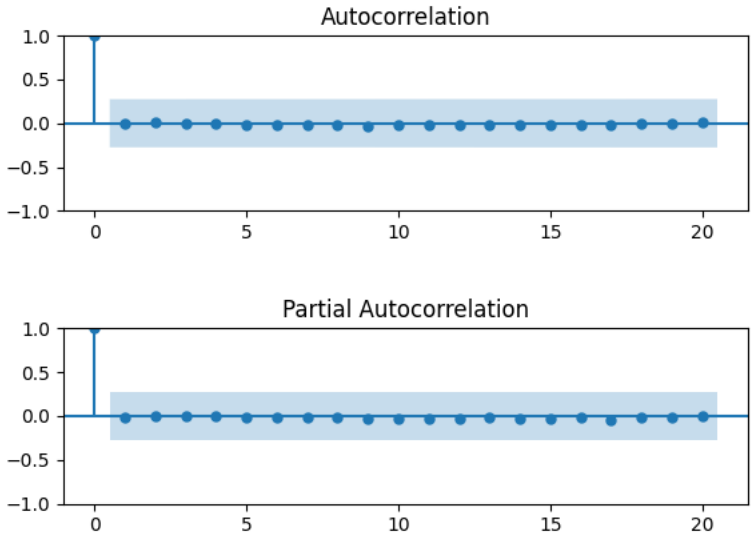


Figure 8.1.6. ACF/PACF plot of SARIMA with AR=0, MA=1 and seasonal_period=24

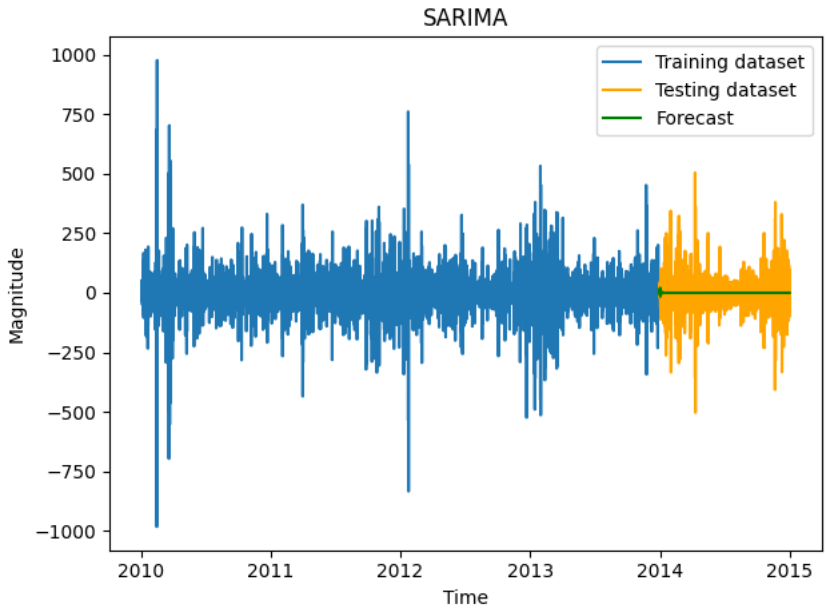


Figure 8.1.7. SARIMA Forecast

The selected SARIMA model forecasts the best on the test data by looking at the graph briefly. The MSE & variance of predicted and forecasted data are best among the models seen. The mean residual error is approximately zero which is good. The Q-value is also the least. The ratio of variance of forecast errors versus variance of residual errors is also good.

```
MSE Prediction data:  796.49
MSE Forecasted data:  1168.42
Variance Prediction data:  796.49
Variance Forecasted data:  1168.42
mean_res_train:  0.01
Q-value:  398.43
var(forecast errors)/var(Residual errors): 1.47
```

Figure 8.1.8. SARIMA model errors

8.2. LM Algorithm - Parameter Estimation

The Levenberg-Marquardt (LM) algorithm is an optimization technique used for parameter estimation in nonlinear regression models. It combines the efficiency of the Gauss-Newton method with the robustness of the gradient descent algorithm, iteratively adjusting the model parameters to minimize the difference between the predicted and observed values.

```
Convergence Occured in 1 iterations
The MA coefficient 1 is: -0.037
Confidence Interval for the estimated parameter(s)
MA Coefficient 1: (-0.047, -0.026)
Estimated Covariance Matrix of estimated parameters:
 [[0.]]
Estimated variance of error: 1579.798
[0.03659315]
[0.]
Roots of numerator: [0.037]
Roots of denominator: [0.]
```

Figure 8.2.1. Parameter Estimation results

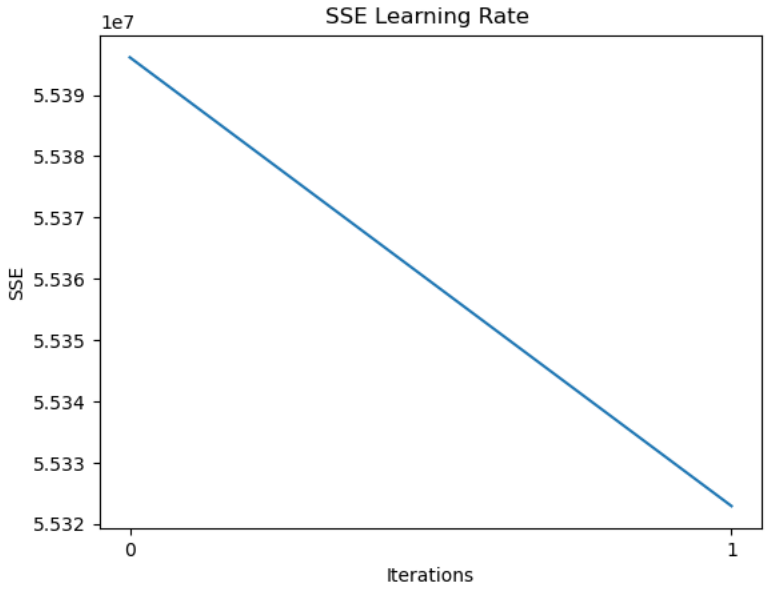


Figure 8.2.2. LM Algorithm Convergence SSE over iterations

	lb_stat	lb_pvalue	bp_stat	bp_pvalue
20	299.054024	3.125952e-52	298.960794	3.266510e-52

Figure 8.2.3. lb_pvalue & bp_value

```
As 398.4348745355847 (Q-value) > 66.3386488629688 (Chi-sq test) => The residual is NOT white
```

Figure 8.2.4. Chi-Square test significance

Looking at the results of LM algorithm, we can see that the convergence occurred in 1 iteration, indicating that the optimization process successfully found a solution. The estimated MA coefficient 1 is -0.037, suggesting a negative relationship between the moving average component and the dependent variable. The confidence interval for the estimated MA coefficient 1 is (-0.05, -0.03), providing a range within which the true value of the coefficient is likely to fall. It does not include zero which means that the model coefficients are significant. The estimated covariance matrix of the parameters shows a variance of 0 for the MA coefficient 1, indicating a lack of variability in its estimate. The estimated variance of the error is 1579.8, representing the variability or dispersion of the residuals in the model. The roots of the numerator and denominator in the transfer function suggest the presence of a single root at 0.04, indicating a potential impact on the dynamics of the system being modeled. Thus, there is no zero/pole cancellation.

9. LSTM

LSTM (Long Short-Term Memory) is a type of recurrent neural network (RNN) that excels at capturing long-term dependencies in sequential data by using memory cells and gates.

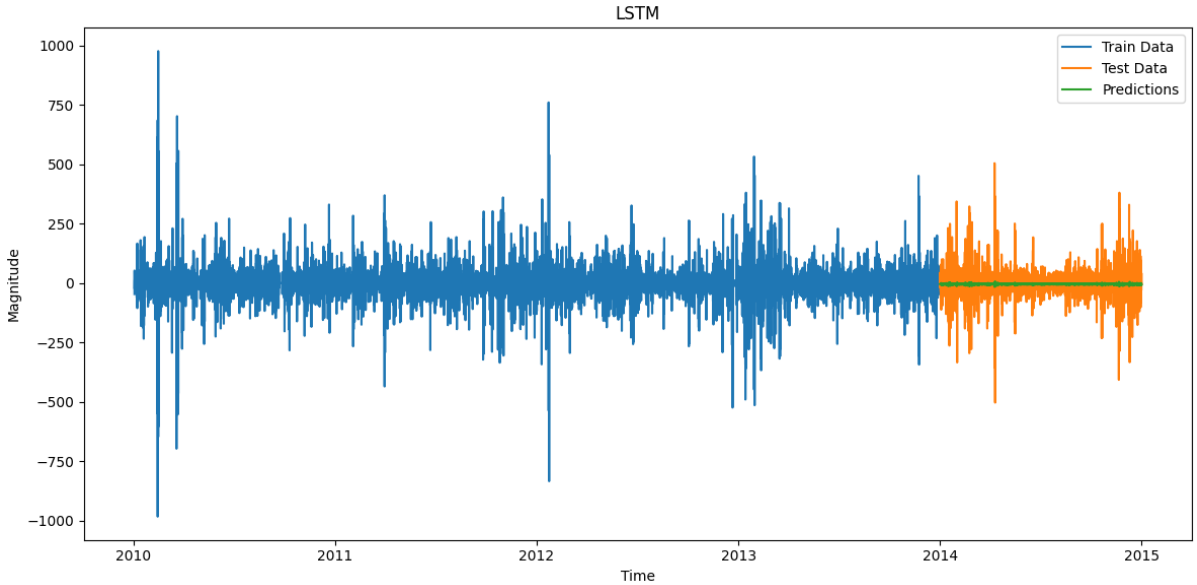


Figure 9.1. LSTM Forecast

The LSTM model consists of an LSTM layer with 4 units and a dense output layer. Early stopping is used as a callback to prevent overfitting. The LSTM model forecasts well on the test data by looking at the graph briefly. The MSE & variance of predicted and forecasted data are good. The mean residual error is higher than other models. The Q-value is not as good as SARIMA model. The ratio of variance of forecast errors versus variance of residual errors is good.

```
MSE Prediction data:  1598.71
MSE Forecasted data:  1191.69
Variance Prediction data:  1578.96
Variance Forecasted data:  1171.96
mean_res_train:  4.44
Q-value:  9111.28
var(forecast errors)/var(Residual errors): 0.74
```

Figure 9.2. LSTM model errors

10. Summary and Conclusion

	method_name	MSE_train	MSE_test	VAR_train	VAR_test	mean_res_train	Q-value	Var_test vs Var_train
0	Average	1581.647536	1170.432433	1581.647535	1170.432082	9.658077e-04	9202.876370	0.74
1	Naive	3276.485045	2195.340813	3276.485043	1170.432082	1.170158e-03	23392.041338	0.36
2	Drift	3277.638627	2943.722675	3277.638448	1201.681987	-1.336925e-02	23392.539307	0.37
3	SES	2160.224148	1826.430687	2160.224139	1170.432082	3.030896e-03	13736.181700	0.54
4	Holt-Winters	1627.691348	1171.268593	1627.691204	1170.739722	-1.199480e-02	9188.840342	0.72
5	Multi-linear Regression	1581.009709	1167.675138	1581.009709	1167.673191	2.739911e-15	9200.034745	0.74
6	SARIMA	796.490709	1168.424308	796.490647	1168.424308	7.901494e-03	398.434875	1.47
7	LSTM	1598.714784	1191.692833	1578.959023	1171.958864	4.444745e+00	9111.282413	0.74

Figure 10. Model Comparison

The table shows the results of different forecasting methods on a dataset, including MSE (Mean Squared Error) for training and testing, VAR (Variance) for training and testing, mean residual values, Q-value, and the ratio of testing variance to training variance. The lower the MSE values, the better the model performance. The Q-value represents the sum of squared residuals, and a lower value indicates a better fit. The ratio of testing variance to training variance measures the difference in variability between the training and testing datasets, with a value close to 1 indicating similar variability. The SARIMA model achieved the lowest MSE for testing, while the LSTM model had comparable performance.

Thus, in conclusion, we can state the following:

- The dataset was originally non-stationary. A seasonal differencing with period=24 and a non-seasonal differencing of order one was performed to make it stationary.
- After evaluating performance of different models, the SARIMA with AR=0 and MA=1 with seasonal period=24 is giving the best performance.

11. Future Scope

There can be a number of adjustments that can be made for the models built in this project. Holt-Winters model can be improved by fine-tuning the smoothing parameters and considering more advanced variations of the model, such as the multiplicative Holt-Winters method or including additional seasonal components. Multi-linear Regression model can be improved by exploring different feature selection techniques or incorporating more advanced regression methods such as polynomial regression or regularization techniques like Ridge or Lasso regression. LSTM (Long Short-Term Memory): This model can be improved by adjusting the architecture and hyperparameters of the LSTM network, such as the number of layers, hidden units, and dropout rate. SARIMA model can be improved by exploring different combinations of autoregressive (AR), moving average (MA), and seasonal components. Adjusting the seasonal period and considering higher-order seasonal patterns may also enhance performance.

12. DASH Application

A DASH application was also built to interact with the data. It has 4 tabs in which user can see the target variable in raw, 1st transformation and stationary form:

1. Data Distribution: The user can see the target variable vs time plot
2. ACF/PACF: The user can see the ACF/PACF plot with a facility to select the number of lags.
3. Stationarity: The user can see the rolling mean and rolling variance graph of the target.
4. Stationarity-ADF/KPSS: The user can see the results of the target's ADF and KPSS test.

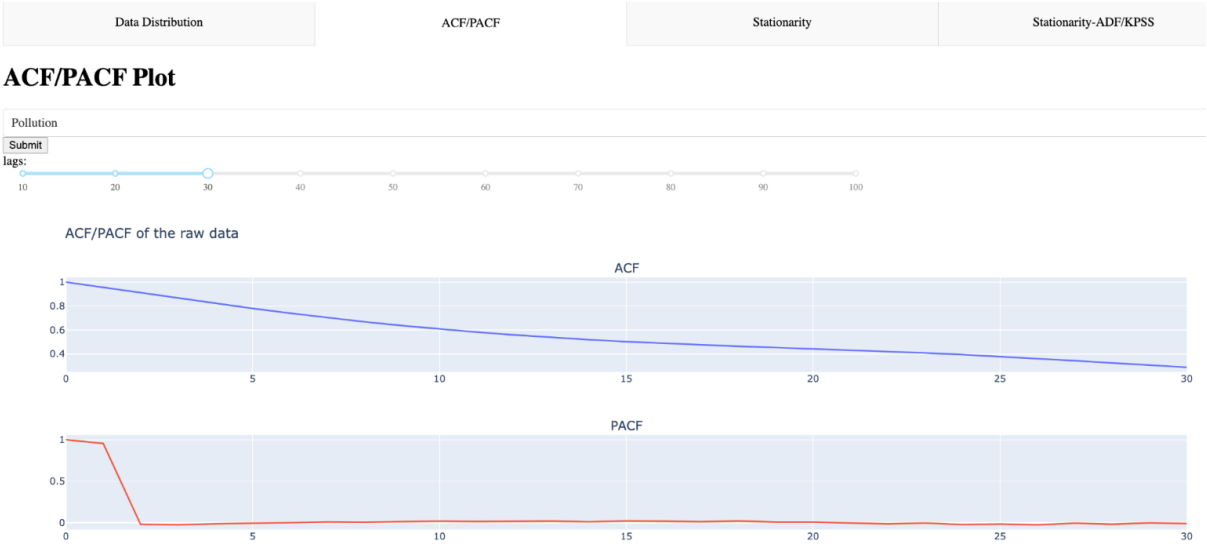


Figure 12. DASH App Snapshot

Appendix

main.py

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
import statsmodels.api as sm
import scipy.stats as stats
from sklearn.preprocessing import StandardScaler
from statsmodels.stats.outliers_influence import variance_inflation_factor

import toolkit

# %%
# to display all the columns
pd.set_option('display.max_columns', None)
#pd.set_option('display.max_rows', None)
np.random.seed(6313)

url = 'https://raw.githubusercontent.com/tanmayk26/Air-Pollution-
Forecasting/main/LSTM-Multivariate_pollution.csv'
df = pd.read_csv(url, index_col='date')
date = pd.date_range(start='1/2/2010',
                     periods=len(df),
                     freq='H')
df.index = date

# Prints head and tail of the dataframe
print(df)

# Describe the data
print(f'Data basic statistics: \n{df.describe()}')

# Check NA value
print(f'NA value: \n{df.isna().sum()}')

# As we can see there are no null values


def wind_encode(s):
    if s == "SE":
        return int(1)
    elif s == "NE":
        return int(2)
    elif s == "NW":
        return int(3)
    else:
        return int(4)


df["wnd_dir"] = df["wnd_dir"].apply(wind_encode)
print(df.info())

# Correlation Matrix
corr = df.corr()
plt.figure(figsize=(16, 8))
```

```
sns.heatmap(corr, annot=True, cmap='BrBG')
plt.title('Correlation Heatmap', fontsize=18)
plt.show()

# Plotting dependent variable vs time
toolkit.plot_graph(x_value=df.index.values, y_value=df['pollution'],
xlabel='Time', ylabel='Pollution', title='Pollution over Time')

# ACF/PACF plot raw data
toolkit.ACF_PACF_Plot(df['pollution'], lags=60)

# Stationarity Tests on raw data
print('ADF test on pollution:-')
toolkit.ADF_Cal(df['pollution'])
print('KPSS test on pollution:-')
toolkit.kpss_test(df['pollution'])
toolkit.CalRollingMeanVarGraph(df, 'pollution')

# STL Decomposition
toolkit.STL_decomposition(df, 'pollution')

# Seasonal Differencing
s = 24
print(f'Performing Seasonal Differencing with interval={s}')

df['seasonal_d_o_1'] = toolkit.seasonal_differencing(df['pollution'],
seasons=s)

# Plotting dependent variable vs time
toolkit.plot_graph(x_value=df.index.values, y_value=df['seasonal_d_o_1'],
xlabel='Time', ylabel='seasonal_d_o_1', title='Pollution over Time')

# ACF/PACF plot seasonaly differenced data
toolkit.ACF_PACF_Plot(df['seasonal_d_o_1'][s:], lags=60)

# Stationarity on seasonaly differenced data
print('ADF test on seasonal_d_o_1:-')
toolkit.ADF_Cal(df['seasonal_d_o_1'][s:])
print('KPSS test on seasonal_d_o_1:-')
toolkit.kpss_test(df['seasonal_d_o_1'][s:])
toolkit.CalRollingMeanVarGraph(df[s:], 'seasonal_d_o_1')

# STL Decomposition

toolkit.STL_decomposition(df[s:], 'seasonal_d_o_1')

# Doing a non-seasonal differencing after the seasonal differrencing
# Transforming data to make it stationary
print('Performing Non-Seasonal Differencing with interval=1')
df['diff_order_1'] = toolkit.differencing(df['seasonal_d_o_1'], s)

# Plotting dependent variable vs time
toolkit.plot_graph(x_value=df.index.values, y_value=df['diff_order_1'],
xlabel='Time', ylabel='diff_order_1', title='Pollution over Time')

# ACF/PACF plot transformed data
toolkit.ACF_PACF_Plot(df['diff_order_1'][s+1:], lags=60)

# Stationarity Tests on transformed data
toolkit.CalRollingMeanVarGraph(df[s+1:], 'diff_order_1')
print('ADF test on diff_order_1:-')
```

```
toolkit.ADF_Cal(df['diff_order_1'][s+1:])
print('KPSS test on diff_order_1:-')
toolkit.kpss_test(df['diff_order_1'][s+1:])

# STL Decomposition
toolkit.STL_decomposition(df[s+1:], 'diff_order_1')


# %%

# Train-Test Split
df.index.freq = 'H'
index_df = df.index
new_index_df = index_df[s+1:]
df_train, df_test = train_test_split(df, shuffle=False, test_size=0.20)
print(f'Training set size: {len(df_train)} rows and
{len(df_train.columns)+1} columns')
print(f'Testing set size: {len(df_test)} rows and {len(df_test.columns)+1}
columns')

# Base Models

df_err = toolkit.base_method('Average', df['diff_order_1'][s+1:],
len(df_train), index_df[s+1:])
naive = toolkit.base_method('Naive', df['diff_order_1'][s+1:],
len(df_train), index_df[s+1:])
drift = toolkit.base_method('Drift', df['diff_order_1'][s+1:],
len(df_train), index_df[s+1:])
ses_point5 = toolkit.base_method('SES', df['diff_order_1'][s+1:],
len(df_train), index_df[s+1:], 0.5)
holt_winters = toolkit.base_method('Holt-Winters',
df['diff_order_1'][s+1:], len(df_train), index_df[s+1:])

df_err = pd.concat([df_err, naive], ignore_index=True)
df_err = pd.concat([df_err, drift], ignore_index=True)
df_err = pd.concat([df_err, ses_point5], ignore_index=True)
df_err = pd.concat([df_err, holt_winters], ignore_index=True)

print('===Model Comparison===\n', df_err)


# %%
df_2 = df[s+1:].drop(['pollution', 'seasonal_d_o_1'], axis=1)

X = df_2.drop(['diff_order_1'], axis=1)
y = df_2['diff_order_1']

X_train, X_test, y_train, y_test = train_test_split(X, y, shuffle=False,
test_size=0.2)
col_list = X_train.columns.to_list()

print(f'Training set size: {len(X_train)} rows and {len(X_train.columns)+1}
columns')
print(f'Testing set size: {len(X_test)} rows and {len(X_test.columns)+1}
columns')


scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

```
X_train = pd.DataFrame(X_train, columns=col_list)
X_test = pd.DataFrame(X_test, columns=col_list)

X1 = X_train.to_numpy()
H = np.dot(X1.T, X1)
u, s, vh = np.linalg.svd(H)
print('Singular values =', s)
print('Condition number is', round(np.linalg.cond(X1), 2))


#calculate VIF for each explanatory variable
vif = pd.DataFrame()
vif['VIF'] = [variance_inflation_factor(X_train.values, i) for i in
range(X_train.shape[1])]
vif['variable'] = X_train.columns

#view VIF for each explanatory variable
print(vif)


# %%
############### Feature selection
X_train_t = sm.add_constant(X_train, prepend=True)
X_test_t = sm.add_constant(X_test, prepend=True)

model_ols = sm.OLS(list(y_train), X_train_t).fit()
print(model_ols.summary())

# %%

X_train_t = X_train_t.drop(['temp'], axis=1)
model_ols = sm.OLS(list(y_train), X_train_t).fit()
print(model_ols.summary())

# %%
X_train_t = X_train_t.drop(['snow'], axis=1)
model_ols = sm.OLS(list(y_train), X_train_t).fit()
print(model_ols.summary())

# %%
X_train_t = X_train_t.drop(['wnd_spd'], axis=1)
model_ols = sm.OLS(list(y_train), X_train_t).fit()
print(model_ols.summary())

# %%
X_train_t = X_train_t.drop(['wnd_dir'], axis=1)
model_ols = sm.OLS(list(y_train), X_train_t).fit()
print(model_ols.summary())

# %%
X_train_t = X_train_t.drop(['dew'], axis=1)
model_ols = sm.OLS(list(y_train), X_train_t).fit()
print(model_ols.summary())


# Final Model
# %%

model_ols = sm.OLS(list(y_train), X_train_t).fit()
print(model_ols.summary())
y_pred = model_ols.predict(X_train_t)
```

```
X_test_t = X_test_t[X_train_t.columns.to_list()]
y_forecast = model_ols.predict(X_test_t)

df_final = pd.DataFrame(list(zip(pd.concat([y_train, y_test], axis=0),
pd.concat([y_pred, y_forecast], axis=0))), columns=['y', 'y_pred'])
toolkit.plot_forecast(df_final, len(y_train), new_index_df,
title_body='Forecast using OLS method', xlabel='Time', ylabel='Pollution')
e, e_sq, MSE_train, VAR_train, MSE_test, VAR_test, mean_res_train =
toolkit.cal_errors(df_final['y'].to_list(), df_final['y_pred'].to_list(),
len(y_train), 0)

lags=50

method_name = 'Multi-linear Regression'
q_value = sm.stats.acorr_ljungbox(e[:len(y_train)], lags=[50],
boxpierce=True, return_df=True)['bp_stat'].values[0]
print('Error values using {} method'.format(method_name))
print('MSE Prediction data: ', round(MSE_train, 2))
print('MSE Forecasted data: ', round(MSE_test, 2))
print('Variance Prediction data: ', round(VAR_train, 2))
print('Variance Forecasted data: ', round(VAR_test, 2))
print('mean_res_train: ', round(mean_res_train, 2))
print('Q-value: ', round(q_value, 2))
var_f_vs_r = round(VAR_test / VAR_train, 2)
print(f'var(forecast errors)/var(Residual errors): {var_f_vs_r:.2f}')

l_err = [[method_name, MSE_train, MSE_test, VAR_train, VAR_test,
mean_res_train, q_value, var_f_vs_r]]
print(l_err)
df_err2 = pd.DataFrame(l_err, columns=['method_name', 'MSE_train',
'MSE_test', 'VAR_train', 'VAR_test', 'mean_res_train', 'Q-value', 'Var_test
vs Var_train'])
df_err = pd.concat([df_err, df_err2], ignore_index=True)

title='ACF Plot for errors - OLS method'
toolkit.Cal_autocorr_plot(model_ols.resid, lags, title)

print('T-Test')
print(model_ols.pvalues)
print('\nF-Test')
print(model_ols.f_pvalue)

print(df_err)

# %%
lags = 100
round_off = 2
ry = sm.tsa.stattools.acf(y_train, nlags=lags)
toolkit.gpac(ry, j_max=30, k_max=30, round_off=round_off)

# %%

model = sm.tsa.ARIMA(y_train, order=(0,0,0), seasonal_order=(1,0,0,24))
model_fit = model.fit()
print(model_fit.summary())
y_result_hat = model_fit.predict()
y_result_h_t = model_fit.forecast(steps=len(y_test))
res_e = y_train - y_result_hat
fore_error = y_test - y_result_h_t

var_f_vs_r = round(np.var(fore_error)/np.var(res_e), 2)
```

```
print(f'variance of forecast errors is = {np.var(fore_error):.2f}')
print(f'variance of residual errors is = {np.var(res_e):.2f}')
print(f'var(forecast errors)/var(Residual errors): {var_f_vs_r:.2f}')

re = []
for lag in range(0, lags + 1):
    re.append(toolkit.Cal_autocorr(model_fit.resid, lag))
toolkit.ACF_PACF_Plot(re, lags=20)


# %%

model = sm.tsa.ARIMA(y_train, order=(0,0,0), seasonal_order=(0,0,1,24))
model_fit = model.fit()
print(model_fit.summary())
y_result_hat = model_fit.predict()
y_result_h_t = model_fit.forecast(steps=len(y_test))
res_e = y_train - y_result_hat
fore_error = y_test - y_result_h_t

var_f_vs_r = round(np.var(fore_error)/np.var(res_e), 2)

print(f'variance of forecast errors is = {np.var(fore_error):.2f}')
print(f'variance of residual errors is = {np.var(res_e):.2f}')
print(f'var(forecast errors)/var(Residual errors): {var_f_vs_r:.2f}')

df_final = pd.DataFrame(list(zip(pd.concat([y_train, y_test], axis=0),
pd.concat([y_result_hat, y_result_h_t], axis=0))), columns=['y', 'y_pred'])
toolkit.plot_forecast(df_final, len(y_train), new_index_df,
title_body='Forecast using SARIMA', xlabel='Time', ylabel='Pollution')
e, e_sq, MSE_train, VAR_train, MSE_test, VAR_test, mean_res_train =
toolkit.cal_errors(df_final['y'].to_list(), df_final['y_pred'].to_list(),
len(y_train), 0)

lags=50

method_name = 'SARIMA'
q_value = sm.stats.acorr_ljungbox(e[:len(y_train)], lags=[50],
boxpierce=True, return_df=True)['bp_stat'].values[0]
print('Error values using {} method'.format(method_name))
print('MSE Prediction data: ', round(MSE_train, 2))
print('MSE Forecasted data: ', round(MSE_test, 2))
print('Variance Prediction data: ', round(VAR_train, 2))
print('Variance Forecasted data: ', round(VAR_test, 2))
print('mean_res_train: ', round(mean_res_train, 2))
print('Q-value: ', round(q_value, 2))
var_f_vs_r = round(VAR_test / VAR_train, 2)
print(f'var(forecast errors)/var(Residual errors): {var_f_vs_r:.2f}')

l_err = [[method_name, MSE_train, MSE_test, VAR_train, VAR_test,
mean_res_train, q_value, var_f_vs_r]]
df_err2 = pd.DataFrame(l_err, columns=['method_name', 'MSE_train',
'MSE_test', 'VAR_train', 'VAR_test', 'mean_res_train', 'Q-value', 'Var_test
vs Var_train'])
df_err = pd.concat([df_err, df_err2], ignore_index=True)

l_err = [['LSTM', 1598.714784, 1191.692833, 1578.959023, 1171.958864,
4.444745, 9111.282413, 0.74]]
df_err2 = pd.DataFrame(l_err, columns=['method_name', 'MSE_train',
'MSE_test', 'VAR_train', 'VAR_test', 'mean_res_train', 'Q-value', 'Var_test
```

```
vs Var_train'])
df_err = pd.concat([df_err, df_err2], ignore_index=True)

print(df_err)

# %%
na = 0
nb = 1
model_name = f'ARMA({na}, {nb})'

theta, sse, var_error, covariance_theta_hat, sse_list =
toolkit.lm_step3(y_train, na, nb)

theta2 = np.array(theta).reshape(-1)
for i in range(na + nb):
    if i < na:
        print('The AR coefficient {} is: {:.3f}'.format(i + 1,
np.round(theta2[i], 3)))
    else:
        print('The MA coefficient {} is: {:.3f}'.format(i + 1 - na,
np.round(theta2[i], 3)))
toolkit.lm_confidence_interval(theta, covariance_theta_hat, na, nb,
round_off=round_off)
print("Estimated Covariance Matrix of estimated parameters:\n",
np.round(covariance_theta_hat, decimals=round_off))
print("Estimated variance of error:", round(var_error, round_off))
print(toolkit.lm_find_roots(theta, na, round_off=round_off))
toolkit.plot_sse(sse_list, '')

re = []
for lag in range(0, lags + 1):
    re.append(toolkit.Cal_autocorr(model_fit.resid, lag))
toolkit.ACF_PACF_Plot(re, lags=20)
Q = sm.stats.acorr_ljungbox(model_fit.resid, lags=[50], boxpierce=True,
return_df=True, model_df=1)['bp_stat'].values[0]
print(f"Q-Value for training set Method) : ", np.round(Q, 2))

DOF = lags - 1
alfa = 0.05
chi_critical = stats.chi2.ppf(1-alfa, DOF)
if Q < chi_critical:
    print(f"As {Q} (Q-value) < {chi_critical} (Chi-sq test) => The residual
is white")
else:
    print(f"As {Q} (Q-value) > {chi_critical} (Chi-sq test) => The residual
is NOT white")

results = sm.stats.diagnostic.acorr_ljungbox(model_fit.resid, model_df=1,
boxpierce=True, lags=[20])
print(results)

plt.plot(list(y_train.index.values + 1), y_train, label='Training dataset')
plt.plot(list(y_test.index.values + 1), y_test, label='Testing dataset',
color='orange')
plt.plot(list(y_test.index.values + 1), y_result_h_t, label='Forecast',
color='green')
plt.xlabel('Time')
plt.ylabel('Magnitude')
plt.title('SARIMA')
plt.legend()
```

```
plt.tight_layout()
plt.show()
```

toolkit.py

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

from statsmodels.tsa.stattools import adfuller
from statsmodels.tsa.stattools import kpss
from statsmodels.tsa.holtwinters import ExponentialSmoothing
import statsmodels.api as sm
import copy

from scipy import signal
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf

from statsmodels.tsa.seasonal import STL


# np.abs(np.roots(ar))
# This process in non-stationary as some roots lie on 1. It should be less
than 1

np.random.seed(6313)

# The following function calculates the Rolling Mean and Rolling Variance
def CalRollingMeanVar(dataset, column_name):
    for i in range(1, len(dataset)):
        return np.mean(dataset[column_name].head(i)),
np.var(dataset[column_name].head(i))


# The following function calculates the Rolling Mean and Rolling Variance
and subsequently plots the graph
def CalRollingMeanVarGraph(dataset, column_name):
    df_plot = pd.DataFrame(columns=['Samples', 'Mean', 'Variance'])
    for i in range(1, len(dataset)):
        df_plot.loc[i] = [i, np.mean(dataset[column_name].head(i)),
np.var(dataset[column_name].head(i))]
    plt.subplot(2, 1, 1)
    plt.plot(df_plot['Samples'], df_plot['Mean'], label='Rolling Mean')
    plt.xlabel('Samples')
    plt.ylabel('Magnitude')
    plt.title('Rolling Mean - {}'.format(column_name))
    plt.subplot(2, 1, 2)
    plt.plot(df_plot['Samples'], df_plot['Variance'], label='Rolling
Variance')
    plt.xlabel('Samples')
    plt.ylabel('Magnitude')
    plt.title('Rolling Variance - {}'.format(column_name))
    plt.tight_layout()
    plt.show()
```

```
# create a differenced series
def differencing(series, order=1):
    diff = []
    for i in range(order):
        diff.append(np.nan)
    for i in range(order, len(series)):
        diff.append(series[i] - series[i - 1])
    return diff


def rev_differencing(series, first_obs, order=1):
    series.reset_index(inplace=True, drop=True)
    rev_diff = []
    for i in range(order-1):
        rev_diff.append(np.nan)
    rev_diff.append(first_obs)
    for i in range(order, len(series)+1):
        rev_diff.append(series[i-1] + rev_diff[i-1])
    return rev_diff


# create a differenced series
def seasonal_differencing(series, seasons=1):
    diff = []
    for i in range(seasons):
        diff.append(np.nan)
    for i in range(seasons, len(series)):
        diff.append(series[i] - series[i - seasons])
    return diff


def rev_seasonal_differencing(series, first_obs, seasons=1):
    series.reset_index(inplace=True, drop=True)
    rev_diff = list(first_obs)
    for i in range(len(series)):
        rev_diff.append(series[i] + rev_diff[i])
    return rev_diff


# # df['rev_seasonal_d_o_1'] = [np.nan] * s +
toolkit.rev_differencing(df['diff_order_1'][s+1:], df['seasonal_d_o_1'][s],
order=1)
# # print(df.head(50))
# # print('-----')
# # print(df.tail(50))
# # df['rev_pollution'] =
toolkit.rev_seasonal_differencing(df['rev_seasonal_d_o_1'][s:],
df['pollution'][:s], seasons=s)
# # print(df.head(50))



# Augmented Dickey-Fuller test (for stationarity)
# For this test, we state the following Null hypothesis (H0) and
alternative hypothesis (H1):
# H0: The time-series has a unit root, meaning it is non-stationary.
# H1: The time-series does not have a unit root, meaning it is stationary.
# When the test statistic is lower than the critical value shown, you
reject the null hypothesis and infer that the time series is stationary.
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))


# KPSS Test (for stationarity)
# For this test, we state the following Null hypothesis (H0) and
alternative hypothesis (H1):
# H0: The process is trend stationary.
# H1: The series has a unit root (series is not stationary).
# the test statistic should be lesser than the provided critical values to
not reject null hypothesis.
# regression options
# "c" : The data is stationary around a constant (default).
# "ct" : The data is stationary around a trend.
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)


# Auto Correlation Function
def Cal_autocorr(y, lag):
    mean = np.mean(y)
    numerator = 0
    denominator = 0
    for t in range(0, len(y)):
        denominator += (y[t] - mean) ** 2
    for t in range(lag, len(y)):
        numerator += (y[t] - mean)*(y[t-lag] - mean)
    return numerator/denominator


# Auto Correlation Function graph
def Cal_autocorr_plot(y, lags, title='ACF Plot', plot_show='Yes'):
    ryy = []
    ryy_final = []
    lags_final = []
    for lag in range(0, lags+1):
        ryy.append(Cal_autocorr(y, lag))
    ryy_final.extend(ryy[:0:-1])
    ryy_final.extend(ryy)
    lags = list(range(0, lags+1, 1))
    lags_final.extend(lags[:0:-1])
    lags_final = [value*(-1) for value in lags_final]
    lags_final.extend(lags)
    plt.figure(figsize=(12, 8))
    markers, stemlines, baseline = plt.stem(lags_final, ryy_final)
    plt.setp(markers, color='red', marker='o')
    plt.axhspan((-1.96 / np.sqrt(len(y))), (1.96 / np.sqrt(len(y))),
alpha=0.2, color='blue')
    plt.xlabel('Lags')
    plt.ylabel('Magnitude')
    plt.title(title)
    plt.tight_layout()
```

```
    if plot_show == 'Yes':
        plt.show()


def base_method(method_name, y, train_n, index_df, alpha=0.5):
    print(f'\n=== {method_name} Method ===\n')
    if method_name == 'Average':
        y_pred = average(y, train_n)
    elif method_name == 'Naive':
        y_pred = naive(y, train_n)
    elif method_name == 'Drift':
        y_pred = drift(y, train_n)
    elif method_name == 'SES':
        y_pred = SES(y, train_n, alpha)
    elif method_name == 'Holt-Winters':
        model = ExponentialSmoothing(y[:train_n], seasonal='add',
seasonal_periods=24).fit()
        y_pred = model.predict(start=0, end=(train_n - 1))
        y_pred = pd.concat([y_pred, model.forecast(steps=len(y)-train_n-
1)])
        print(y_pred)
    if method_name == 'Holt-Winters':
        e = y - y_pred
        e_sq = e ** 2
        MSE_train = np.nanmean(e_sq[:train_n])
        VAR_train = np.nanvar(e[:train_n])
        MSE_test = np.nanmean(e_sq[train_n:])
        VAR_test = np.nanvar(e[train_n:])
        mean_res_train = np.nanmean(e[:train_n])
    else:
        e, e_sq, MSE_train, VAR_train, MSE_test, VAR_test, mean_res_train =
cal_errors(y, y_pred, train_n, 2)
    df = pd.DataFrame(list(zip(y, y_pred, e, e_sq)), columns=['y',
'y_pred', 'e', 'e^2'])
    print(df)
    if method_name == 'SES':
        title_suffix = 'with alpha={}'.format(alpha)
        plot_forecast(df, train_n, index_df, method_name, 'Yes',
title_suffix)
    elif method_name == 'Holt-Winters':
        holt_winter_plot(y[:train_n], y[train_n:], y_pred[train_n:])
    else:
        plot_forecast(df, train_n, index_df, method_name)
    lags = 50
    q_value = sm.stats.acorr_ljungbox(e[2:train_n], lags=[50],
boxpierce=True, return_df=True)['bp_stat'].values[0]
    #q_value = Cal_q_value(e[:train_n], lags, train_n, 2)
    print(sm.stats.acorr_ljungbox(e[2:train_n], lags=[50], boxpierce=True,
return_df=True))
    print('Error values using {} method'.format(method_name))
    print('MSE Prediction data: ', round(MSE_train, 2))
    print('MSE Forecasted data: ', round(MSE_test, 2))
    print('Variance Prediction data: ', round(VAR_train, 2))
    print('Variance Forecasted data: ', round(VAR_test, 2))
    print('mean_res_train: ', round(mean_res_train, 2))
    print('Q-value: ', round(q_value, 2))
    var_f_vs_r = round(VAR_test / VAR_train, 2)
    print(f'var(forecast errors)/var(Residual errors): {var_f_vs_r:.2f}')
    # title = f'ACF Plot for errors - {method_name}'
    # Cal_autocorr_plot(e[:train_n], lags, title)
    l_err = [[method_name, MSE_train, MSE_test, VAR_train, VAR_test,
```

```
mean_res_train, q_value, var_f_vs_r]]
    print(l_err)
    df_err = pd.DataFrame(l_err, columns=['method_name', 'MSE_train',
'MSE_test', 'VAR_train', 'VAR_test', 'mean_res_train', 'Q-value', 'Var_test
vs Var_train'])
    return df_err


def holt_winter_plot(train, test, test_predictions):
    train.plot(legend = True, label = 'Training dataset')
    test.plot(legend = True, label = 'Testing dataset', color='orange')
    test_predictions.plot(legend = True, label = 'Forecast',
color='green')
    plt.xlabel('Time')
    plt.ylabel('Magnitude')
    plt.title('Holt Winters method & forecast')
    plt.tight_layout()
    plt.show()


def average(y, n):
    y_pred = list(np.nan for i in range(0, len(y)))
    for i in range(1, n):
        y_pred[i] = np.mean(y[:i])
    for i in range(n, len(y)):
        y_pred[i] = np.mean(y[:n])
    return y_pred


def naive(y, n):
    y_pred = list(np.nan for i in range(0, len(y)))
    for i in range(1, n):
        y_pred[i] = y[i-1]
    for i in range(n, len(y)):
        y_pred[i] = y[n - 1]
    return y_pred


def drift(y, n):
    y_pred = list(np.nan for i in range(0, len(y)))
    for i in range(2, n):
        y_pred[i] = y[i-1] + ((y[i-1]-y[0]))/(i-1)
    for i in range(n, len(y)):
        y_pred[i] = y[n-1] + (i+1-n)*(y[n-1]-y[0])/(n-1)
    return y_pred


def SES(y, n, alpha):
    y_pred = list(np.nan for i in range(0, len(y)))
    l0 = y[0]
    y_pred[1] = alpha * l0 + (1 - alpha) * l0
    for i in range(2, n):
        y_pred[i] = alpha * y[i-1] + (1-alpha) * y_pred[i-1]
    for i in range(n, len(y)):
        y_pred[i] = alpha * y[n-1] + (1-alpha) * y_pred[n-1]
    return y_pred
```

```
def cal_errors(y, y_pred, n, skip_first_n_obs=0):
    e = []
    e_sq = []
    for i in range(0, len(y)):
        if y_pred[i] != np.nan:
            e.append(y[i] - y_pred[i])
            e_sq.append((y[i] - y_pred[i]) ** 2)
        else:
            e.append(np.nan)
            e_sq.append(np.nan)
    MSE_train = np.nanmean(e_sq[skip_first_n_obs:n])
    VAR_train = np.nanvar(e[skip_first_n_obs:n])
    MSE_test = np.nanmean(e_sq[n:])
    VAR_test = np.nanvar(e[n:])
    mean_res_train = np.nanmean(e[skip_first_n_obs:n])
    return e, e_sq, MSE_train, VAR_train, MSE_test, VAR_test,
mean_res_train


def plot_forecast(df, n, index_df, method_name='', plot_show='Yes',
title_body='method & forecast', title_suffix='', xlabel='Time',
ylabel='Magnitude'):
    plt.plot(list(index_df[:n].values + 1), df['y'][:n], label='Training
dataset')
    plt.plot(list(index_df[n:].values + 1), df['y'][n:], label='Testing
dataset', color='orange')
    plt.plot(list(index_df[n:].values + 1), df['y_pred'][n:],
label='Forecast',  color='green')


    plt.xlabel(xlabel)
    plt.ylabel(ylabel)
    plt.title('{} {} {}'.format(method_name, title_body, title_suffix))
    plt.legend()
    plt.tight_layout()
    if plot_show == 'Yes':
        plt.show()

# exclude first two observations for hw2. add a skip_first_n_obs parameter?
def Cal_q_value(e, lags, train_n, skip_first_obs=0):
    acf = 0
    data = [x for x in e[skip_first_obs:] if np.isnan(x) == False]
    for lag in range(1, lags + 1):
        acf += Cal_autocorr(data, lag) ** 2
    q_value = len(data) * (acf)
    return q_value


# LSE Beta B = inv(X.T*X)*X.T*Y
def LSE_Beta(X_train, y_train):
    X_train = sm.add_constant(X_train, prepend=True)
    return np.dot(np.linalg.inv(np.dot(X_train.T, X_train)),
np.dot(X_train.T, y_train))


# Backward Step-wise Regression Function: looks at the p-value of each
feature. Remove the feature with the highest p-value till there is no
feature with p-value>0.05.
def backward_stepwise_regression(X_train, y_train):
    f_X_train = copy.deepcopy(X_train)
```

```
    f_y_train = copy.deepcopy(y_train)
    model_comp = {}
    feature_rmv = 'Overall'
    feature_rmv_list = []
    i = 1
    print('\n--- Overall ---\n')
    while i < len(f_X_train):
        model_ols = sm.OLS(f_y_train, f_X_train).fit()
        model_comp[feature_rmv] = [model_ols.aic, model_ols.bic,
model_ols.rsquared_adj]
        stop = len(f_X_train.columns) - 2
        while len(f_X_train.columns) > stop:
            current_aic = model_ols.aic
            current_bic = model_ols.bic
            current_adjr2 = model_ols.rsquared_adj
            results = []
            for variable in f_X_train.columns[1:]:
                temp_f_X_train = f_X_train.drop(variable, axis=1)
                temp_model = sm.OLS(f_y_train, temp_f_X_train).fit()
                temp_aic = temp_model.aic
                temp_bic = temp_model.bic
                temp_adjr2 = temp_model.rsquared_adj
                results.append((variable, temp_aic, temp_bic, temp_adjr2))
            results.sort(key=lambda x: (x[1], x[2], -x[3]))
            feature_rmv, best_aic, best_bic, best_adjr2 = results[0]
            if (best_aic < current_aic) and (best_bic < current_bic) and
(best_adjr2 > current_adjr2):
                feature_rmv_list.append(feature_rmv)
                print(model_ols.summary())
                f_X_train.drop(feature_rmv, axis=1, inplace=True)
                print(f'\n---Dropping {feature_rmv} ---\n')
            else:
                break
            break
        i += 1
    return model_comp, feature_rmv_list


def Moving_Average_raw(df_new, m, folding_order=0):
    df = copy.deepcopy(df_new)
    k = int((m - 1) / 2)
    even = (m+1)%2
    col_name = '{}-MA'.format(m)
    df[col_name] = np.nan
    for j in range(k, len(df)-k-even):
        for i in range(j-k, j+k+1+even):
            if np.isnan(df[col_name][j]):
                df[col_name][j] = df['Temp'][i]
            else:
                df[col_name][j] += df['Temp'][i]
    df[col_name] = round(df[col_name]/m, 3)
    if folding_order > 0:
        col_name_2 = '{}x{}-MA'.format(folding_order, m)
        df[col_name_2] = np.nan
        for j in range(k+1, len(df)-k-1):
            for i in range(j-1, j+1):
                if np.isnan(df[col_name_2][j]):
                    df[col_name_2][j] = df[col_name][i]
                else:
                    df[col_name_2][j] += df[col_name][i]
        df[col_name_2] = round(df[col_name_2] / folding_order, 3)
```

```
        df = df.loc[:, df.columns != col_name]
    return df



def Moving_Average(df_new, m, folding_order=0):
    df = copy.deepcopy(df_new)
    k = int((m - 1) / 2)
    even = (m+1)%2
    col_name = '{}-MA'.format(m)
    df[col_name] = np.nan
    for j in range(k, len(df)-k-even):
        df[col_name][j] = np.nansum(df['Temp'][j-k : j+k+1+even])
    df[col_name] = round(df[col_name]/m, 3)
    if folding_order > 0:
        col_name_2 = '{}x{}-MA'.format(folding_order, m)
        df[col_name_2] = np.nan
        for j in range(k+1, len(df)-k-1):
            df[col_name_2][j] = np.nansum(df[col_name][j - 1: j + 1])
        df[col_name_2] = round(df[col_name_2] / folding_order, 3)
        df = df.loc[:, df.columns != col_name]
    return df



def ar2_loop_method(N, coef, mean=0, std=1):
    np.random.seed(6313)
    e = np.random.normal(mean, std, N)
    y = np.zeros(len(e))
    for i in range(len(e)):
        if i == 0:
            y[0] = e[0]
        elif i == 1:
            y[i] = -coef[0]*y[i-1] + e[i]
        else:
            y[i] = -coef[0]*y[i-1] - coef[1]*y[i-2] + e[i]
    return y, e


def ar2_dslim_method(N, order, coef, mean=0, std=1):
    np.random.seed(6313)
    e = np.random.normal(mean, std, N)
    num = [1] + [0] * order
    den = [1] + coef
    system = (num, den, 1)
    t, y_dlsim = signal.dlsim(system, e)
    return y_dlsim.reshape(-1), e


def ar2_estimated_parameter_vals(y):
    y_t_1 = np.append(0, y[:-1])
    y_t_2 = np.append(0, y_t_1[:-1])
    X = np.vstack((y_t_1, y_t_2)).T
    return list(-np.array(LSE_Beta(X, y)))


def ma2_loop_method(N, coef, mean=0, std=1):
    np.random.seed(6313)
    e = np.random.normal(mean, std, N)
```

```
    y = np.zeros(len(e))
    for i in range(len(e)):
        if i == 0:
            y[0] = e[0]
        elif i == 1:
            y[i] = e[i] + coef[0]*e[i-1]
        else:
            y[i] = e[i] + coef[0]*e[i-1] + coef[1]*e[i-2]
    return y, e


def ma2_dslim_method(N, order, coef, mean=0, std=1):
    np.random.seed(6313)
    e = np.random.normal(mean, std, N)
    num = [1] + coef
    den = [1] + [0] * order
    system = (num, den, 1)
    t, y_dlsim = signal.dlsim(system, e)
    return y_dlsim.reshape(-1), e


def dslim_method(N, num, den, mean=0, std=1, seed=6313):
    np.random.seed(seed)
    e = np.random.normal(mean, std, N)
    system = (num, den, 1)
    t, y_dlsim = signal.dlsim(system, e)
    return y_dlsim.reshape(-1), e


def arma_input():
    N = int(input('Number of observations:'))
    mean_e = int(input('Enter the mean of white noise:'))
    var_e = int(input('Enter the variance of white noise:'))
    na = int(input('Enter AR order:'))
    nb = int(input('Enter MA order:'))
    den = []
    for i in range(1, na + 1):
        den.append(float(input(f'Enter the coefficient {i} of AR
process:')))
    num = []
    for i in range(1, nb + 1):
        num.append(float(input(f'Enter the coefficient {i} of MA
process:')))
    max_order = max(na, nb)
    ar_coef = [0] * (max_order)
    ma_coef = [0] * (max_order)
    for i in range(na):
        ar_coef[i] = den[i]
    for i in range(nb):
        ma_coef[i] = num[i]
    ar_params = np.array(ar_coef)
    ma_params = np.array(ma_coef)
    ar = np.r_[1, ar_params]
    ma = np.r_[1, ma_params]
    print('AR coef:', ar)
    print('MA coef:', ma)
    return N, na, nb, ar, ma, mean_e, var_e


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()
    plt.subplot(211)
    plt.title('ACF/PACF of the raw 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()
    return fig


def gpac(ry, show_heatmap='Yes', j_max=7, k_max=7, round_off=3, seed=6313):
    np.random.seed(seed)
    gpac_table = np.zeros((j_max, k_max-1))
    for j in range(0, j_max):
        for k in range(1, k_max):
            phi_num = np.zeros((k, k))
            phi_den = np.zeros((k, k))
            for x in range(0, k):
                for z in range(0, k):
                    phi_num[x][z] = ry[abs(j + 1 - z + x)]
                    phi_den[x][z] = ry[abs(j - z + x)]
            phi_num = np.roll(phi_num, -1, 1)
            det_num = np.linalg.det(phi_num)
            det_den = np.linalg.det(phi_den)
            if det_den != 0 and not np.isnan(det_den):
                phi_j_k = det_num / det_den
            else:
                phi_j_k = np.nan
            gpac_table[j][k - 1] = phi_j_k #np.linalg.det(phi_num) /
np.linalg.det(phi_den)
    if show_heatmap=='Yes':
        plt.figure(figsize=(24, 14))
        x_axis_labels = list(range(1, k_max))
        sns.heatmap(gpac_table, annot=True, xticklabels=x_axis_labels,
fmt=f'.{round_off}f', vmin=-0.1, vmax=0.1)#, cmap='BrBG'
        plt.title(f'GPAC Table', fontsize=18)
        plt.show()
    #print(gpac_table)
    return gpac_table



def arma_gpac_pacf(ar_order=0, ma_order=0, num=None, den=None, N=1000,
mean_e=0, var_e=1, lags=15, j_max=7, k_max=7, round_off=2, seed=6313,
user_ip='Yes'):
    N, na, nb, ar, ma, mean_e, var_e = arma_input()
    arma_process = sm.tsa.ArmaProcess(ar, ma)
    mean_y = mean_e*(1 + np.sum(ar))/(1 + np.sum(ma))
    y = arma_process.generate_sample(N, scale=np.sqrt(var_e)) + mean_y
    print('ARMA Process:', list(np.around(np.array(y[:15]), round_off)))
    ry = arma_process.acf(lags=lags)
    print('ACF:', list(np.around(np.array(ry[:15]),round_off)))
    gpac(ry, j_max=j_max, k_max=k_max, round_off=round_off)
    ACF_PACF_Plot(y, lags=20)


# LM algorithm

def lm_cal_e(y, na, theta, seed=6313):
```

```
    np.random.seed(seed)
    den = theta[:na]
    num = theta[na:]
    if len(den) > len(num):  # matching len of num and den
        for x in range(len(den) - len(num)):
            num = np.append(num, 0)
    elif len(num) > len(den):
        for x in range(len(num) - len(den)):
            den = np.append(den, 0)
    den = np.insert(den, 0, 1)
    num = np.insert(num, 0, 1)
    sys = (den, num, 1)
    _, e = signal.dlsim(sys, y)
    return e


def lm_step1(y, na, nb, delta, theta):
    n = na + nb
    e = lm_cal_e(y, na, theta)
    sse_old = np.dot(np.transpose(e), e)
    X = np.empty(shape=(len(y), n))
    for i in range(0, n):
        theta[i] = theta[i] + delta
        e_i = lm_cal_e(y, na, theta)
        x_i = (e - e_i) / delta
        X[:, i] = x_i[:, 0]
        theta[i] = theta[i] - delta
    A = np.dot(np.transpose(X), X)
    g = np.dot(np.transpose(X), e)
    return A, g, X, sse_old


def lm_step2(y, na, A, theta, mu, g):
    delta_theta = np.matmul(np.linalg.inv(A + (mu *
np.identity(A.shape[0]))), g)
    theta_new = theta + delta_theta
    e_new = lm_cal_e(y, na, theta_new)
    sse_new = np.dot(np.transpose(e_new), e_new)
    if np.isnan(sse_new):
        sse_new = 10 ** 10
    return sse_new, delta_theta, theta_new


def lm_step3(y, na, nb):
    N = len(y)
    n = na+nb
    mu = 0.01
    mu_max = 10 ** 20
    max_iterations = 500
    delta = 10 ** -6
    var_error = 0
    covariance_theta_hat = 0
    sse_list = []
    theta = np.zeros(shape=(n, 1))

    for iterations in range(max_iterations):
        A, g, X, sse_old = lm_step1(y, na, nb, delta, theta)
        sse_new, delta_theta, theta_new = lm_step2(y, na, A, theta, mu, g)
        sse_list.append(sse_old[0][0])
        if iterations < max_iterations:
            if sse_new < sse_old:
```

```
                if np.linalg.norm(np.array(delta_theta), 2) < 10 ** -3:
                    theta_hat = theta_new
                    var_error = sse_new / (N - n)
                    covariance_theta_hat = var_error * np.linalg.inv(A)
                    print(f"Convergence Occured in {iterations}
iterations")
                    break
                else:
                    theta = theta_new
                    mu = mu / 10
            while sse_new >= sse_old:
                mu = mu * 10
                if mu > mu_max:
                    print('No Convergence')
                    break
                sse_new, delta_theta, theta_new = lm_step2(y, na, A, theta,
mu, g)
        if iterations > max_iterations:
            print('Max Iterations Reached')
            break
        theta = theta_new
    return theta_new, sse_new, var_error[0][0], covariance_theta_hat,
sse_list


def lm_confidence_interval(theta, cov, na, nb, round_off=4):
    print("Confidence Interval for the estimated parameter(s)")
    lower_bound = []
    upper_bound = []
    for i in range(len(theta)):
        lower_bound.append(theta[i] - 2 * np.sqrt(cov[i, i]))
        upper_bound.append(theta[i] + 2 * np.sqrt(cov[i, i]))
    lower_bound = np.round(lower_bound, decimals=round_off)
    upper_bound = np.round(upper_bound, decimals=round_off)
    for i in range(na+nb):
        if i < na:
            print(f"AR Coefficient {i + 1}: ({lower_bound[i][0]},
{upper_bound[i][0]})")
        else:
            print(f"MA Coefficient {i + 1 - na}: ({lower_bound[i][0]},
{upper_bound[i][0]})")


def lm_find_roots(theta, na, round_off=4):
    den = theta[:na]
    num = theta[na:]
    if len(den) > len(num):
        for x in range(len(den) - len(num)):
            num = np.append(num, 0)
    elif len(num) > len(den):
        for x in range(len(num) - len(den)):
            den = np.append(den, 0)
    else:
        pass
    den = np.insert(den, 0, 1)
    num = np.insert(num, 0, 1)
    print(np.roots(num))
    print(np.roots(den))
    print("Roots of numerator:", np.round(np.roots(num),
decimals=round_off))
    print("Roots of denominator:", np.round(np.roots(den),
```

```
decimals=round_off))


def plot_sse(sse_list, model_name):
    plt.plot(sse_list)
    plt.xlabel('Iterations')
    plt.ylabel('SSE')
    plt.title(f'SSE Learning Rate {model_name}')
    plt.xticks(np.arange(0, len(sse_list), step=1))
    plt.show()


def STL_decomposition(data, column_name):
    series = pd.Series(data[column_name].values, index = data.index.values,
name = column_name)
    STL_tf = STL(series, period=24)
    res = STL_tf.fit()
    T = res.trend
    S = res.seasonal
    R = res.resid
    plt.figure(figsize=(16, 8))
    # This
    fig = res.plot()
    plt.show()
    str_trend = max(0, 1-(np.var(R)/np.var(T+R)))
    print(f'The strength of trend for {column_name} is {round(str_trend,
3)}.')
    str_seasonality = max(0, 1-(np.var(R)/np.var(S+R)))
    print(f'The strength of seasonality for {column_name} is
{round(str_seasonality, 3)}.')


def plot_graph(x_value, y_value, xlabel='Time', ylabel='Magnitude',
title='Samples over Time'):
    # Plotting dependent variable vs time
    plt.figure(figsize=(16, 8))
    plt.plot(list(x_value), y_value)
    plt.xlabel(xlabel)
    plt.ylabel(ylabel)
    plt.title(title)
    plt.legend()
    plt.tight_layout()
    plt.show()
```

lstm.py

```
# Import necessary modules
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import statsmodels.api as sm
from sklearn.preprocessing import MinMaxScaler
from keras.models import Sequential
from keras.layers import Dense, LSTM
from keras.callbacks import EarlyStopping

np.random.seed(6313)

url = 'https://raw.githubusercontent.com/tanmayk26/Air-Pollution-
```

```
Forecasting/main/LSTM-Multivariate_pollution.csv'
df = pd.read_csv(url, index_col='date')
date = pd.date_range(start='1/2/2010',
                     periods=len(df),
                     freq='H')
df.index = date

# Prints head and tail of the dataframe
print(df)

# Describe the data
print(f'Data basic statistics: \n{df.describe()}')

# Check NA value
print(f'NA value: \n{df.isna().sum()}')

# As we can see there are no null values

def wind_encode(s):
    if s == "SE":
        return int(1)
    elif s == "NE":
        return int(2)
    elif s == "NW":
        return int(3)
    else:
        return int(4)


df["wnd_dir"] = df["wnd_dir"].apply(wind_encode)
print(df.info())

def differencing(series, order=1):
    diff = []
    for i in range(order):
        diff.append(np.nan)
    for i in range(order, len(series)):
        diff.append(series[i] - series[i - 1])
    return diff

def seasonal_differencing(series, seasons=1):
    diff = []
    for i in range(seasons):
        diff.append(np.nan)
    for i in range(seasons, len(series)):
        diff.append(series[i] - series[i - seasons])
    return diff

s=24
df['seasonal_d_o_1'] = seasonal_differencing(df['pollution'], seasons=s)
df['diff_order_1'] = differencing(df['seasonal_d_o_1'], s)

def create_dataset(dataset, look_back=1):
    dataX, dataY = [], []
    for i in range(len(dataset) - look_back - 1):
        a = dataset[i:(i + look_back), 0]
        dataX.append(a)
        dataY.append(dataset[i + look_back, 0])
    return np.array(dataX), np.array(dataY)

target = 'diff_order_1'
```

```
scaler = MinMaxScaler(feature_range=(0, 1))
dataset = scaler.fit_transform(df[[target]][s+1:])
train_size = int(len(dataset) * 0.8)
test_size = len(dataset) - train_size
df_train, df_test = dataset[0:train_size, :],
dataset[train_size:len(dataset), :]

look_back = 3
X_train, y_train = create_dataset(df_train, look_back)
X_test, y_test = create_dataset(df_test, look_back)

X_train = np.reshape(X_train, (X_train.shape[0], 1, X_train.shape[1]))
X_test = np.reshape(X_test, (X_test.shape[0], 1, X_test.shape[1]))

# LSTM model
model = Sequential()
model.add(LSTM(4, input_shape=(1, look_back)))
model.add(Dense(1))
model.compile(loss='mean_squared_error', optimizer='adam')
early_stopping = EarlyStopping(monitor='val_loss', patience=10,
restore_best_weights=True)
model.fit(X_train, y_train, epochs=10, batch_size=1, verbose=2,
callbacks=[early_stopping], validation_split=0.1)

y_pred = model.predict(X_train)
y_fcst = model.predict(X_test)

y_pred = scaler.inverse_transform(y_pred)
y_train = scaler.inverse_transform([y_train])
y_fcst = scaler.inverse_transform(y_fcst)
y_test = scaler.inverse_transform([y_test])

plt.figure(figsize=(12, 6))
plt.plot(df[target][s+1:len(df_train)].index.values,
df[target][s+1:len(df_train)].values, label='df_train Data')
plt.plot(df[target][s+1+len(df_train):].index.values,
df[target][s+1+len(df_train):].values, label='df_test Data')
plt.plot(df[target][s+1+len(df_train)+4:].index.values, y_fcst,
label='Predictions')
plt.xlabel('Time')
plt.ylabel('Magnitude')
plt.title('LSTM')
plt.legend()
plt.tight_layout()
plt.show()

def cal_errors(y, y_pred, n, skip_first_n_obs=0):
    e = []
    e_sq = []
    for i in range(0, len(y)):
        if y_pred[i] != np.nan:
            e.append(y[i] - y_pred[i])
            e_sq.append((y[i] - y_pred[i]) ** 2)
        else:
            e.append(np.nan)
            e_sq.append(np.nan)
    MSE_train = np.nanmean(e_sq[skip_first_n_obs:n])
    VAR_train = np.nanvar(e[skip_first_n_obs:n])
    MSE_test = np.nanmean(e_sq[n:])
    VAR_test = np.nanvar(e[n:])
```

```
    mean_res_train = np.nanmean(e[skip_first_n_obs:n])
    return e, e_sq, MSE_train, VAR_train, MSE_test, VAR_test,
mean_res_train


df_y_train = pd.Series(y_train[0])
df_y_test = pd.Series(y_test[0])
trainPredict2 = pd.Series(y_pred[:, 0])
testPredict2 = pd.Series(y_fcst[:, 0])

df_final = pd.DataFrame(list(zip(pd.concat([df_y_train, df_y_test],
axis=0), pd.concat([trainPredict2, testPredict2], axis=0))), columns=['y',
'y_pred'])
print(df_final)

e, e_sq, MSE_train, VAR_train, MSE_test, VAR_test, mean_res_train =
cal_errors(df_final['y'].to_list(), df_final['y_pred'].to_list(),
len(df_y_train), 0)

lags=50

method_name = 'LSTM'
q_value = sm.stats.acorr_ljungbox(e[:len(df_y_train)], lags=[50],
boxpierce=True, return_df=True)['bp_stat'].values[0]
print('Error values using {} method'.format(method_name))
print('MSE Prediction data: ', round(MSE_train, 2))
print('MSE Forecasted data: ', round(MSE_test, 2))
print('Variance Prediction data: ', round(VAR_train, 2))
print('Variance Forecasted data: ', round(VAR_test, 2))
print('mean_res_train: ', round(mean_res_train, 2))
print('Q-value: ', round(q_value, 2))
var_f_vs_r = round(VAR_test / VAR_train, 2)
print(f'var(forecast errors)/var(Residual errors): {var_f_vs_r:.2f}')

l_err = [[method_name, MSE_train, MSE_test, VAR_train, VAR_test,
mean_res_train, q_value, var_f_vs_r]]
df_err2 = pd.DataFrame(l_err, columns=['method_name', 'MSE_train',
'MSE_test', 'VAR_train', 'VAR_test', 'mean_res_train', 'Q-value', 'Var_test
vs Var_train'])
print(df_err2)
```

app.py

```
import dash
from dash import html
from dash import dcc
from dash.dependencies import Input, Output, State
import pandas as pd
import numpy as np
from statsmodels.tsa.stattools import acf, pacf, adfuller, kpss
import plotly.graph_objs as go
from plotly.subplots import make_subplots
import toolkit

np.random.seed(6313)

# load data
url = 'https://raw.githubusercontent.com/tanmayk26/Air-Pollution-
Forecasting/main/LSTM-Multivariate_pollution.csv'
```

```
df = pd.read_csv(url, index_col='date')
date = pd.date_range(start='1/2/2010',
                     periods=len(df),
                     freq='H')
df.index = date
df['Date'] = date
s=24


external_stylesheets = [
    {
        "href": (
            "https://fonts.googleapis.com/css2?"
            "family=Lato:wght@400;700&display=swap"
        ),
        "rel": "stylesheet",
    },
]
app = dash.Dash(__name__, external_stylesheets=external_stylesheets)
app.title = "Air Pollution Timeseries Forecasting"

tab1_display = html.Div([
    html.H1("Data Distribution"),
    html.Div([
        dcc.Dropdown(
            id='target-selector',
            options=[
                {'label': 'Pollution', 'value': 'pollution'},
                {'label': 'Seasonal Difference (s=24)', 'value':
'seasonal_diff'},
                {'label': 'Non-Seasonal Difference (order=1)', 'value':
'nonseasonal_diff'}
            ],
            value='pollution'
        ),

        html.Button('Submit', id='submit-button', n_clicks=0)
    ]),
    html.Div(
        dcc.Graph(id="pollution-chart",
                  style={'height': '500px'},
                  config={"displayModeBar": False})
    ),
])

tab2_display = html.Div([
    html.H1("ACF/PACF Plot"),
    html.Div([
        dcc.Dropdown(
            id='target-selector2',
            options=[
                {'label': 'Pollution', 'value': 'pollution'},
                {'label': 'Seasonal Difference (s=24)', 'value':
'seasonal_diff'},
                {'label': 'Non-Seasonal Difference (order=1)', 'value':
'nonseasonal_diff'}
            ],
            value='pollution'
        ),

        html.Button('Submit', id='submit-button2', n_clicks=0)
```

```
    ]),
    html.Div([
        html.Label("lags:"),
        dcc.Slider(
            id="lag-slider",
            min=10,
            max=100,
            step=10,
            value=20,
        ),
    ], style={'width': '70%', 'display': 'inline-block', 'vertical-align':
'middle'}),


    dcc.Graph(id="acf-pacf-graph", style={'height': '500px'}),
])

tab3_display = html.Div([
    html.H1("Stationarity - Rolling Mean/Variance"),
    html.Div([
        dcc.Dropdown(
            id='target-selector3',
            options=[
                {'label': 'Pollution', 'value': 'pollution'},
                {'label': 'Seasonal Difference (s=24)', 'value':
'seasonal_diff'},
                {'label': 'Non-Seasonal Difference (order=1)', 'value':
'nonseasonal_diff'}
            ],
            value='pollution'
        ),

        html.Button('Submit', id='submit-button3', n_clicks=0)
    ]),

    dcc.Graph(id="rolling-mean-var-graph", style={'height': '500px'}),

])

tab4_display = html.Div([
    html.H1("Stationarity - ADF/KPSS"),
    html.Div([
        dcc.Dropdown(
            id='target-selector4',
            options=[
                {'label': 'Pollution', 'value': 'pollution'},
                {'label': 'Seasonal Difference (s=24)', 'value':
'seasonal_diff'},
                {'label': 'Non-Seasonal Difference (order=1)', 'value':
'nonseasonal_diff'}
            ],
            value='pollution'
        ),

        html.Button('Submit', id='submit-button4', n_clicks=0)
    ]),

    html.Div(id="adf-test-output"),
    html.Div(id="kpss-test-result"),

])
```

```
@app.callback(
    Output("pollution-chart", "figure"),
    Input('submit-button', 'n_clicks'),
    State('target-selector', 'value')
)
def update_charts(n_clicks, target):
    if target == 'seasonal_diff':
        s = 24
        filtered_data = toolkit.seasonal_differencing(df['pollution'],
seasons=s)
    elif target == 'nonseasonal_diff':
        s = 24
        filtered_data = toolkit.seasonal_differencing(df['pollution'],
seasons=s)
        filtered_data = toolkit.differencing(filtered_data, 24)
    else:
        filtered_data = df['pollution']
    figure = {
        "data": [
            {
                "x": df['Date'],
                "y": filtered_data,
                "type": "lines",
                "hovertemplate": "%{y:.2f}<extra></extra>",
                "name": target
            },
        ],
        "layout": {
            "title": {
                "text": "Data - " + target,
                "x": 0.05,
                "xanchor": "left",
            },
        },
    }
    return figure


@app.callback(
    Output("acf-pacf-graph", "figure"),
    Input('submit-button2', 'n_clicks'),
    Input("lag-slider", "value"),
    State('target-selector2', 'value')
)
def update_tab2(n_clicks, lags, target):
    if target == 'seasonal_diff':
        s = 24
        filtered_data = toolkit.seasonal_differencing(df['pollution'],
seasons=s)
        filtered_data = filtered_data[s:]
    elif target == 'nonseasonal_diff':
        s = 24
        filtered_data = toolkit.seasonal_differencing(df['pollution'],
seasons=s)
        filtered_data = toolkit.differencing(filtered_data, 24)
        filtered_data = filtered_data[s+1:]
    else:
        filtered_data = df['pollution']
    fig = make_subplots(rows=2, cols=1, subplot_titles=("ACF", "PACF"))
```

```
    fig.add_trace(go.Scatter(x=list(range(int(lags)+1)),
y=acf(filtered_data, nlags=int(lags)), name='ACF'), row=1, col=1)
    fig.add_trace(go.Scatter(x=list(range(int(lags)+1)),
y=pacf(filtered_data, nlags=int(lags)), name='PACF'), row=2, col=1)
    fig.update_layout(height=500, showlegend=False, title_text="ACF/PACF of
the raw data")
    return fig



@app.callback(
    Output("rolling-mean-var-graph", "figure"),
    Input('submit-button3', 'n_clicks'),
    State('target-selector3', 'value')
)
def update_tab3(n_clicks, target):
    if target == 'seasonal_diff':
        s = 24
        filtered_data = toolkit.seasonal_differencing(df['pollution'],
seasons=s)
        filtered_data = pd.Series(filtered_data[s:])
    elif target == 'nonseasonal_diff':
        s = 24
        filtered_data = toolkit.seasonal_differencing(df['pollution'],
seasons=s)
        filtered_data = toolkit.differencing(filtered_data, 24)
        filtered_data = pd.Series(filtered_data[s + 1:])
    else:
        filtered_data = pd.Series(df['pollution'])

    rolling_mean = filtered_data.rolling(window=7).mean()
    rolling_var = filtered_data.rolling(window=7).var()
    fig = make_subplots(rows=2, cols=1, shared_xaxes=True,
vertical_spacing=0.05)
    fig.add_trace(go.Scatter(x=rolling_mean.index, y=rolling_mean,
name='Rolling Mean'), row=1, col=1)
    fig.add_trace(go.Scatter(x=rolling_var.index, y=rolling_var,
name='Rolling Var'), row=2, col=1)
    fig.update_layout(title=f'Rolling Mean and Rolling Var of {target}',
height=600)
    return fig



@app.callback(
    [Output("adf-test-output", "children"),
     Output("kpss-test-result", "children")],
    Input('submit-button4', 'n_clicks'),
    State('target-selector4', 'value')
)
def update_tab3(n_clicks, target):
    if target == 'seasonal_diff':
        s = 24
        filtered_data = toolkit.seasonal_differencing(df['pollution'],
seasons=s)
        filtered_data = pd.Series(filtered_data[s:])
    elif target == 'nonseasonal_diff':
        s = 24
        filtered_data = toolkit.seasonal_differencing(df['pollution'],
seasons=s)
        filtered_data = toolkit.differencing(filtered_data, 24)
```

```
        filtered_data = pd.Series(filtered_data[s + 1:])
    else:
        filtered_data = pd.Series(df['pollution'])
    adf_result = adfuller(filtered_data)
    adf_output = f"ADF Test Results:\nADF Statistic:
{adf_result[0]:.4f}\np-value: {adf_result[1]:.4f}\nCritical Values:
{adf_result[4]}"
    kpss_result = kpss(filtered_data)
    kpss_output = f"KPSS test statistic: {kpss_result[0]:.4f}, p-value:
{kpss_result[1]:.4f}"

    return adf_output, kpss_output


app.layout = html.Div([
    dcc.Tabs(id='tabs', value='tab1', children=[
        dcc.Tab(label='Data Distribution', value='tab1',
children=[tab1_display]),
        dcc.Tab(label='ACF/PACF', value='tab2', children=[tab2_display]),
        dcc.Tab(label='Stationarity', value='tab3',
children=[tab3_display]),
        dcc.Tab(label='Stationarity-ADF/KPSS', value='tab4',
children=[tab4_display]),
    ]),
])

if __name__ == '__main__':
    app.run_server(debug=False)
```

References

https://www.kaggle.com/datasets/rupakroy/lstm-datasets-multivariate-univariate
https://bobrupakroy.medium.com/multi-variate-lstm-time-series-forecasting-1a736009f6d
https://www.kaggle.com/code/rupakroy/lstm-multivariate
https://otexts.com/fpp3/seasonal-arima.html
https://otexts.com/fpp3/stlfeatures.html
https://dash.plotly.com/tutorial
https://realpython.com/python-dash/
https://medium.com/analytics-vidhya/python-code-on-holt-winters-forecasting-3843808a9873
https://www.geeksforgeeks.org/detecting-multicollinearity-with-vif-python/