TIME SERIES ANALYSIS AND FORECASTING FINAL PROJECT

Dr. Reza Jafari

Final Project Report

Sharmin Kantharia

(December 16, 2020)

TABLE OF CONTENTS

Abstract

Time series analysis and forecasting of stock market data is a difficult process, due to the unpredictable nature of the stocks. Stock market moves are ultimately an aggregate of all the decisions of all its participants (multiple variables), which makes it difficult to find reliable patterns. The goal of this project is to understand stock market data, identify the various components of the time series such as trend or seasonality, make prediction using different types of models, ranging from simple base models to more complex ones such as the ARIMA model. Using the various statistics as results of the modeling process, a final model is selected using the Mean Squared Error (MSE) of the prediction and the Q values of the models. The objective is to understand the implementation of all concepts learnt during the course and create a base for advancement.

Introduction

Stock market prediction aims to determine the future movement of the stock value of a financial exchange. Many factors such as changes in interest rates, politics, and economic growth, makes the prediction process volatile and difficult. A major advantage of stock market prediction is that it provides business firms, trade agencies and even individual investors, the opportunity to improve investments and make profits.

This project explores the Reliance Stock Market Data. Reliance Industries Limited (RIL) is an Indian multinational conglomerate company headquartered in India. The dataset has many important features such as the 'Close', 'Open', 'High', 'Low', 'Date', 'Volume', and so on. These features consider the pricing history of a stock and the trading volumes. The objective of this project is to forecast the 'Close' prices of a stock based using different modeling methods.

The project involves data pre-processing, visualization, and time series decomposition, followed by modeling using base models, Holt-Winter's methods, multiple linear regression, ARMA process and the ARIMA process. Data pre-processing and visualization provides information on the stationarity of the data and the correlation between the features. Observing this information helps in understanding what kind of data can be provided to the different models.

Time series decomposition allows to find the strengths of the various components in the time series, which helps in understanding which models need to be considered for prediction. The final statistics are provided at the end of the report, using which, the final or best model will be chosen.

Dataset Description

The dataset is the Reliance Stock Market Data of the Nifty-50 index and consists of the price history and trading volumes from the National Stock Exchange (NSE) in India. The time series spans from January 1, 2000 to July 31, 2020 and is collected daily. The data is taken from Kaggle and can be found here.

The dataset provides the following information:

- The dataset is provided in a .csv format and has 12 columns and 5184 instances.
- The 'Symbol' and 'Series' columns provide text information, so they are excluded from the main coding in the project.

- The 'Trades', 'Deliverable Volume' and the '%Deliverable' columns have a lot of null values and hence, they are removed as well during pre-processing.
- The target variable is 'Close' which describes the closing price of a stock on that day.
- The time series is recorded daily, excluding weekends or holidays.

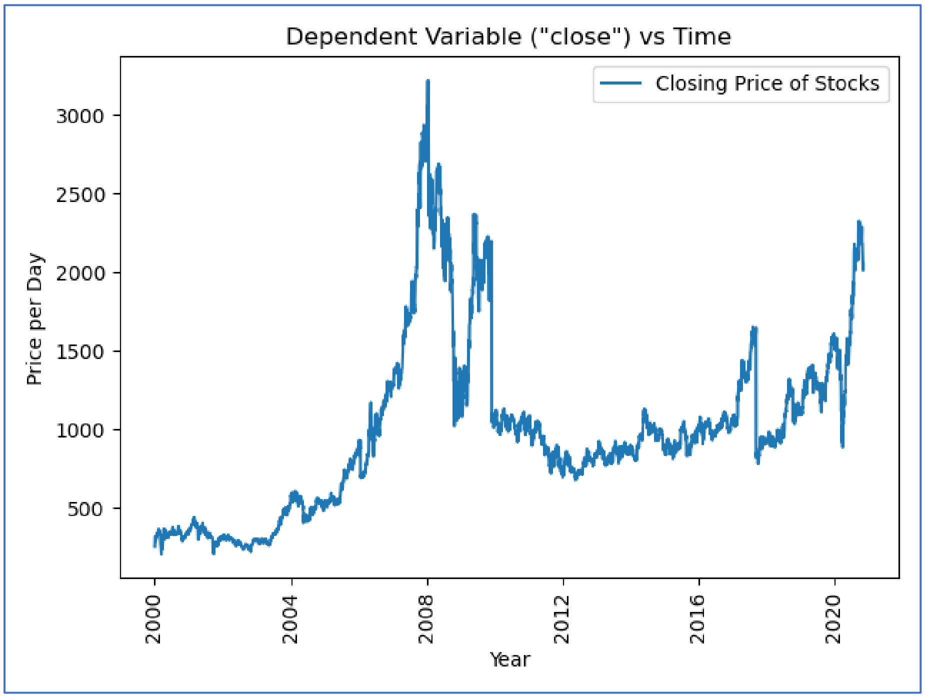
	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
1	Date	Symbol	Series	Prev Close	Open	High	Low	Last	Close	VWAP	Volume	Turnover	Trades	Deliverable Volume	%Deliverble
2	1/3/2000	RELIANCE	EQ	233.05	237.5	251.7	237.5	251.7	251.7	249.37	4456424	1.11E+14			
3	1/4/2000	RELIANCE	EQ	251.7	258.4	271.85	251.3	271.85	271.85	263.52	9487878	2.50E+14			
4	1/5/2000	RELIANCE	EQ	271.85	256.65	287.9	256.65	286.75	282.5	274.79	26833684	7.37E+14			
5	1/6/2000	RELIANCE	EQ	282.5	289	300.7	289	293.5	294.35	295.45	15682286	4.63E+14			
6	1/7/2000	RELIANCE	EQ	294.35	295	317.9	293	314.5	314.55	308.91	19870977	6.14E+14			
7	1/10/2000	RELIANCE	EQ	314.55	317.4	318.7	305.3	306.65	308.5	312.35	13417057	4.19E+14			
8	1/11/2000	RELIANCE	EQ	308.5	307.95	310.95	283.85	288.5	288.5	296.4	12544322	3.72E+14			
9	1/12/2000	RELIANCE	EQ	288.5	289	305	282.15	304.7	301.7	294.57	12109507	3.57E+14			
10	1/13/2000	RELIANCE	EQ	301.7	306	316.4	304.1	309.75	311.85	311.79	17076042	5.32E+14			
11	1/14/2000	RELIANCE	EQ	311.85	309.5	321.65	309.5	317	316.3	316.17	13460592	4.26E+14			
12	1/17/2000	RELIANCE	EQ	316.3	318.25	322.9	307.05	307.5	308.75	315.77	10729180	3.39E+14			
13	1/18/2000	RELIANCE	EQ	308.75	309	318.95	305.55	318.25	314.2	312.14	10083321	3.15E+14			
14	1/19/2000	RELIANCE	EQ	314.2	322.85	324.9	316.2	318.8	319.6	321.17	11481392	3.69E+14			

```
RangeIndex: 5184 entries, 0 to 5183
Data columns (total 15 columns):
 #   Column              Non-Null Count  Dtype
---  ------              --------------  -----
 0   Date                5184 non-null   object
 1   Symbol              5184 non-null   object
 2   Series              5184 non-null   object
 3   Prev Close          5184 non-null   float64
 4   Open                5184 non-null   float64
 5   High                5184 non-null   float64
 6   Low                 5184 non-null   float64
 7   Last                5184 non-null   float64
 8   Close               5184 non-null   float64
 9   VWAP                5184 non-null   float64
 10  Volume              5184 non-null   int64
 11  Turnover            5184 non-null   float64
 12  Trades              2334 non-null   float64
 13  Deliverable Volume  4670 non-null   float64
 14  %Deliverble         4670 non-null   float64
dtypes: float64(11), int64(1), object(3)
memory usage: 546.8+ KB
```

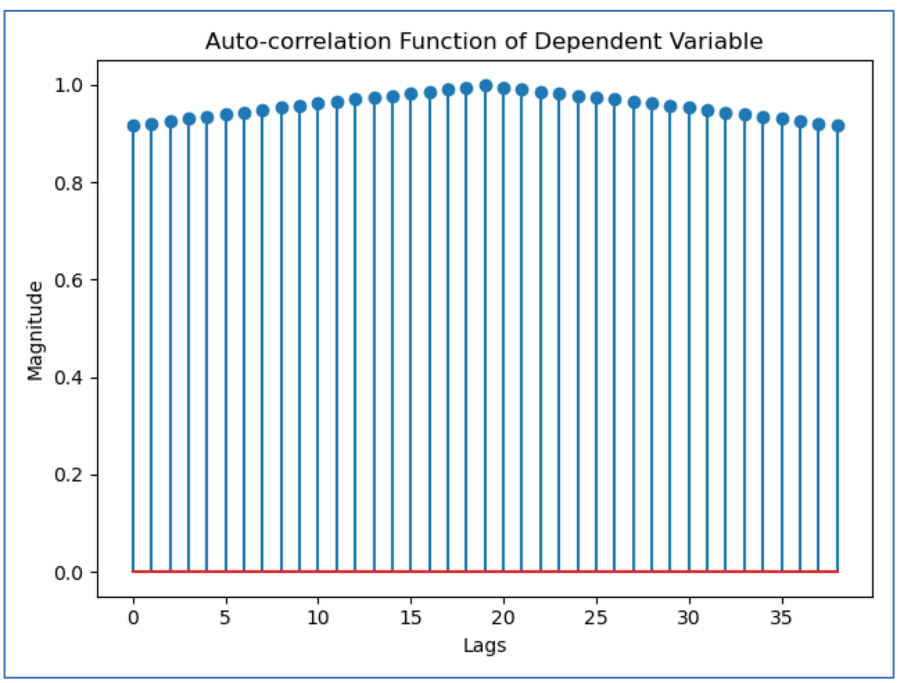
Data Pre-processing

In data pre-processing, the time series was loaded, the data was copied to be used in different sections of the project, the date was formatted, the check for stationarity was performed and the ACF and heatmap of the correlation matrix was plotted. The details are provided below.

- The dependent variable was taken as 'Close'. The initial plot of the dependent variable vs time is shown below. It suggests that initially, there is an increasing trend in prices from 2000 to 2008, after which there is a sudden drop in the prices. This lowered price is maintained until about 2017, after which there is again an increasing trend till 2020.



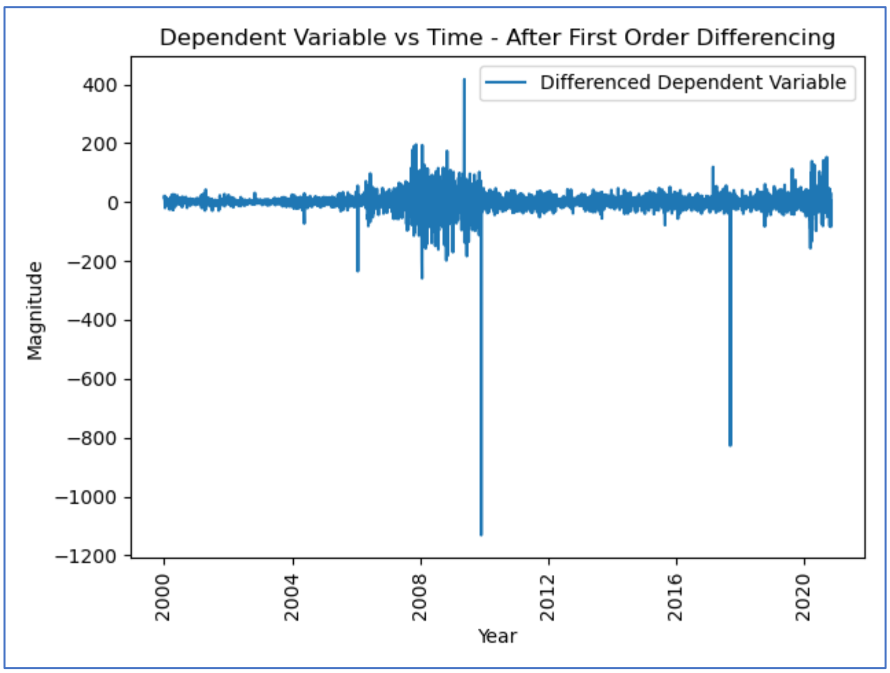
- The ACF plot of the original data suggests that there it is not stationary, as the ACF values do not drop to 0, but are gradually decreasing to 0. 20 lags were used to calculate the ACF.



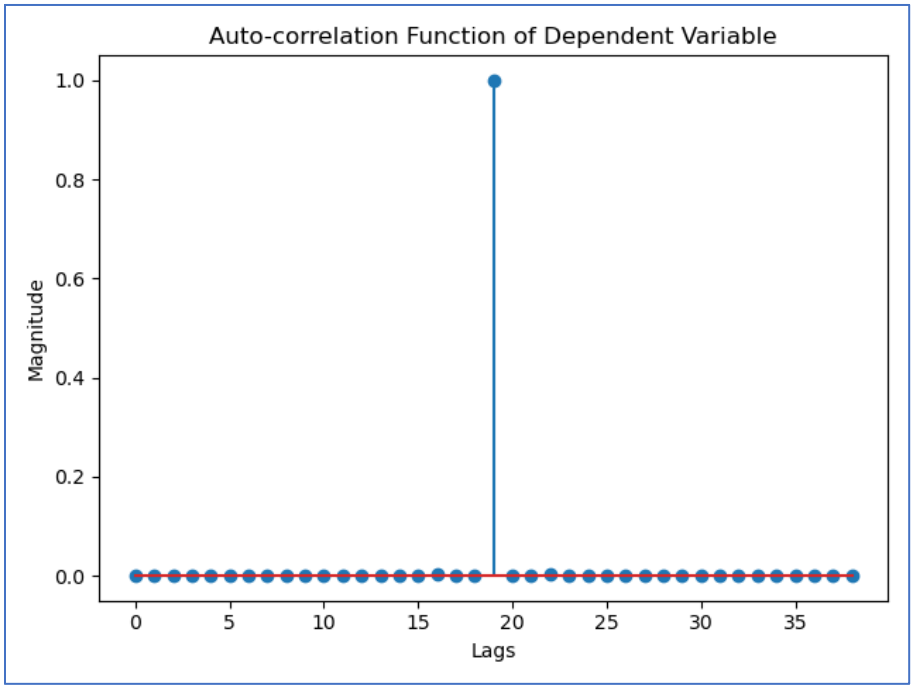
- Looking at the ADF-test conducted on the original data, the p-value is 0.379, which is much greater than 0.05. Also, the ADF Statistic is -1.802. This indicated clearly that the dataset is not stationary with over 95% confidence and needs to be transformed.

```
ADF-test on original dependent variable:
ADF Statistic: -1.802876
p-value: 0.379035
Critical Values:
    1%: -3.432
    5%: -2.862
    10%: -2.567
```

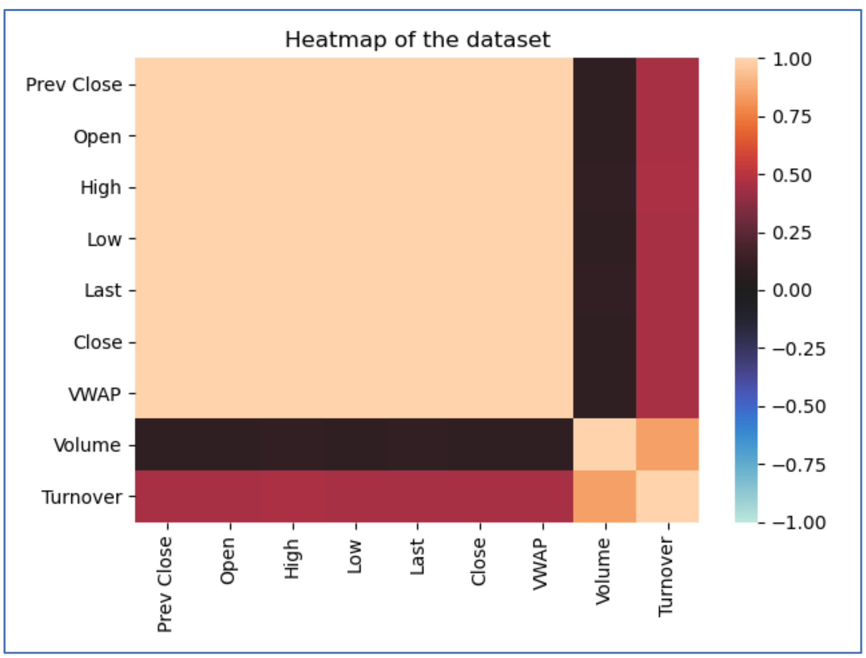
- After performing First Order Differencing, the dataset looks stationary. The ACF plot of the differenced dependent variable represents white noise, indicating that the data is now stationary. The ADF-test conducted on the differenced data shows the p-value of 0.00 and the ADF Statistic is -16.309, further proving with over 95% confidence that the data is now stationary.




```
ADF-test on differenced dependent variable:
ADF Statistic: -16.309782
p-value: 0.000000
Critical Values:
    1%: -3.432
    5%: -2.862
    10%: -2.567
```



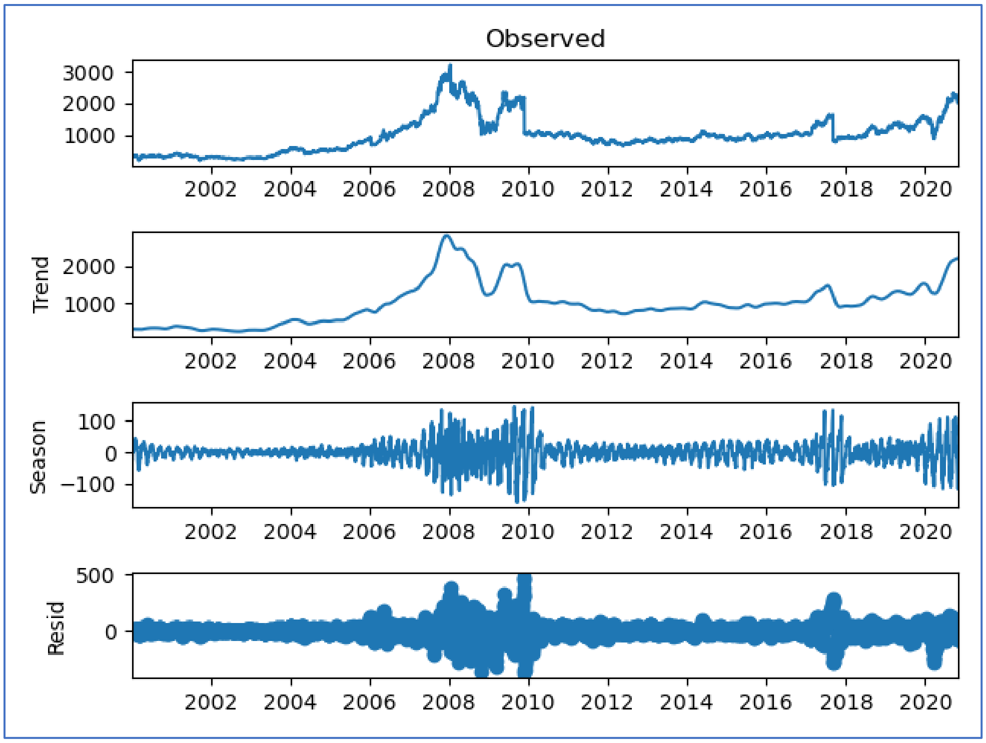
- The heatmap of the correlation matrix (using Pearson's Correlation Coefficient) as shown below suggests that all variable, except 'Volume' and 'Turnover' are highly and positively correlated with the target variable 'Close'.



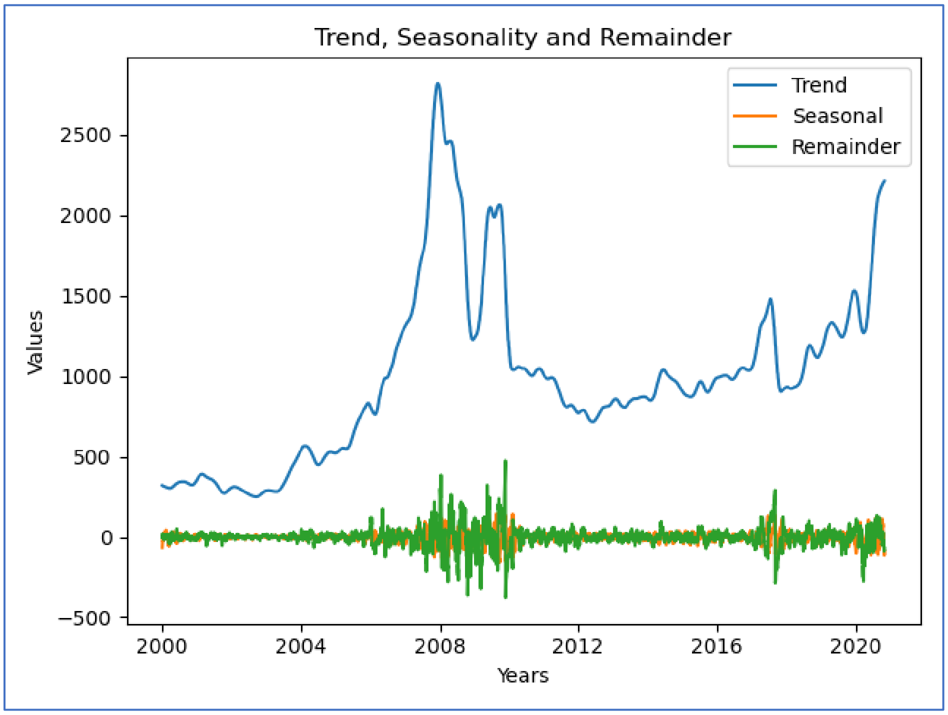
Time Series Decomposition

Time series decomposition is a process which allows us to find the different components of the time series such as the Trend, Seasonality and Cyclicity. The STL decomposition method stands for Seasonal-Trend decomposition using LOESS and is used in this project. Here, the strength of the trend and seasonality helps to understand which component is more dominant in this time series, using which further modeling can be performed.

- The plot formed after performing STL decomposition is divided into 3 main components.
  - Trend – The values of the trend component are very large (greater that 2000), which when compared to the values of the seasonal component suggest that the data is highly trended.
  - Seasonality – The seasonality of the data can be observed as irregular and infrequent. The values of the seasonal component are comparatively lower that the trend (ranging from -100 to slightly above 100).
  - Residual – this is the remainder component, which again has lower values compared to the trend component. This plot also shows that there is little to no cyclicity in the time series. This can be argued to be so, since the data is of stock prices, which can be highly unpredictable.



- The plot below segregates the components and plots them versus the time. The strength of the trend is 0.987 while that of seasonality is 0.246, clearly indicating that the time series is highly trended and has very little seasonality. This observation will allow for better modeling decisions in the further stages.



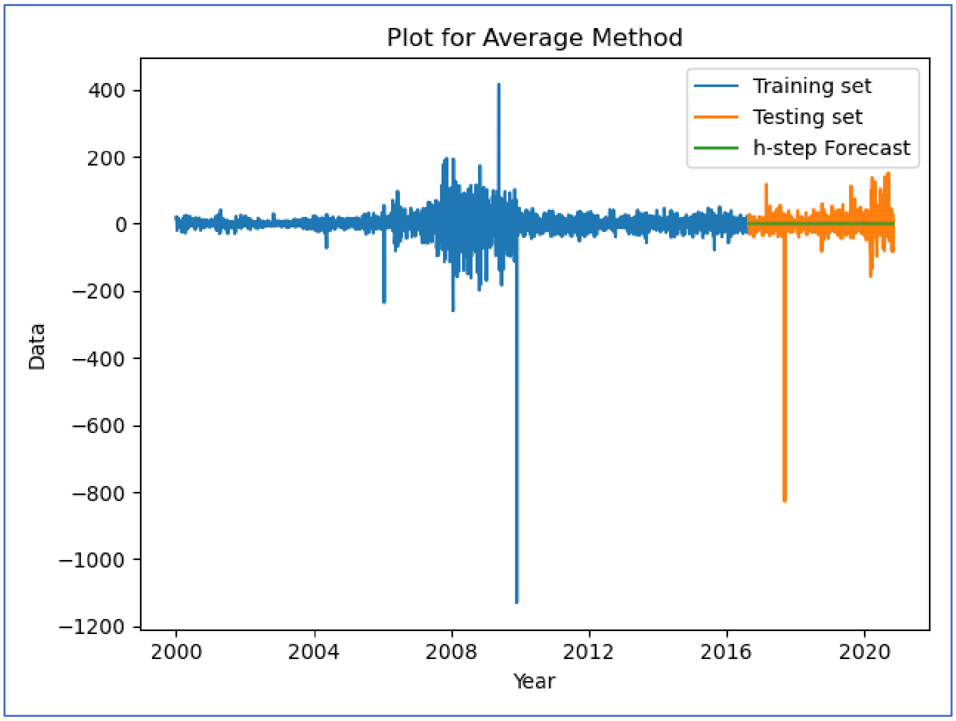

```
The strength of Trend for data is: 0.9877395204177081
The strength of Seasonality for data is: 0.2463688126295941
```

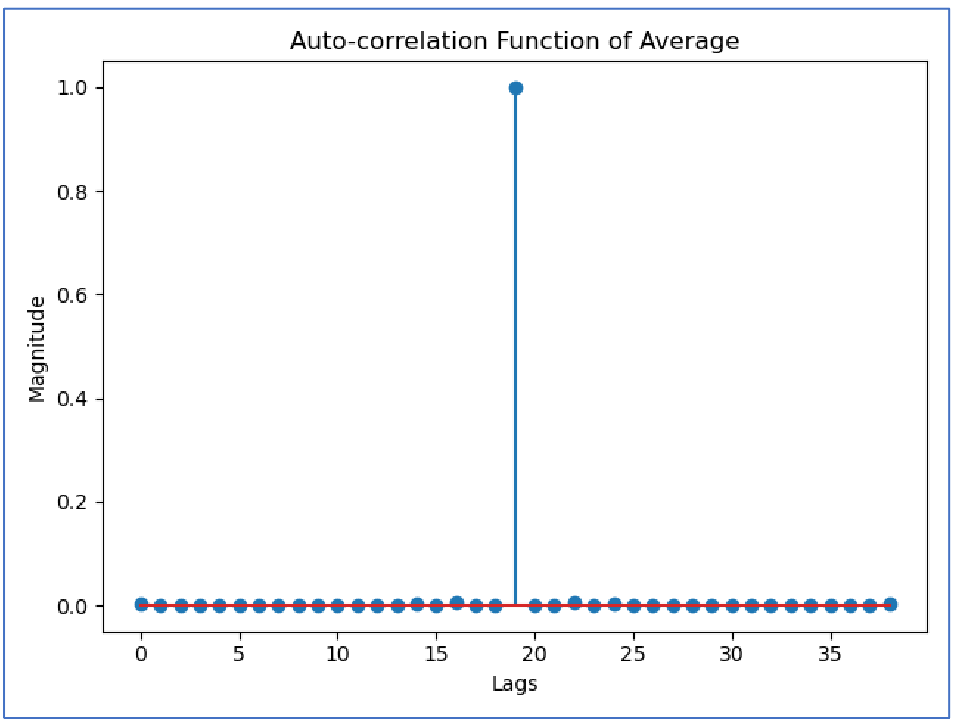
Base Models

There are 4 simple forecasting methods which help develop the base models. By looking at the residuals (prediction errors), forecast errors, Q values, etc. the best model can be selected. The base models act as guides to help improve the more complex models and better the forecasting process. To perform modeling using the base methods, stationary data was used as it gave much better results compared to the non-stationary data.

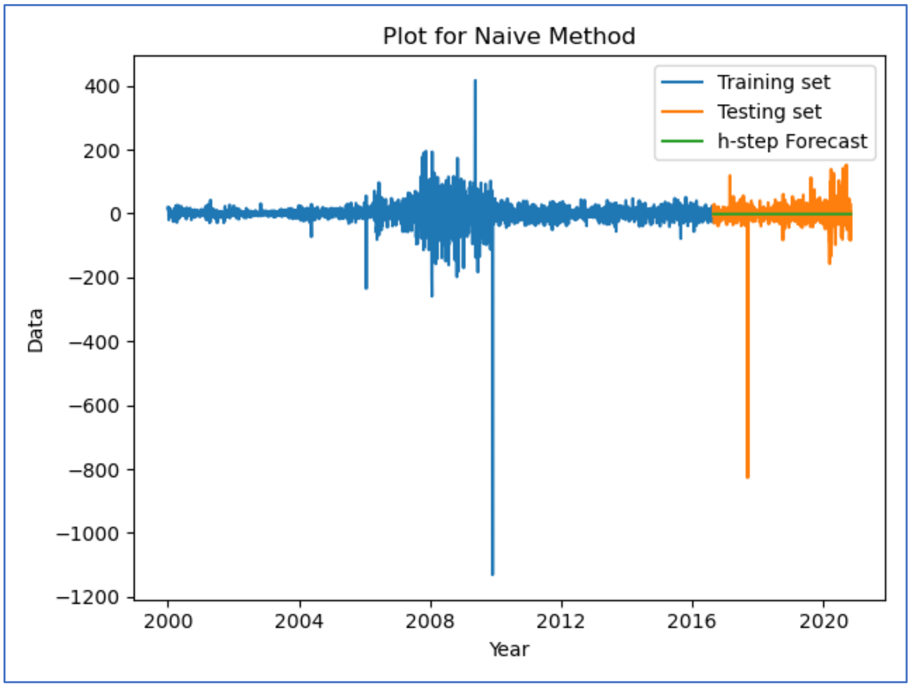
1. <u>Average Method</u> – The forecast of all future values is equal to the average (mean) of the historical data. The average method assumes that all observations are of equal importance and gives them equal weights when generating forecasts.

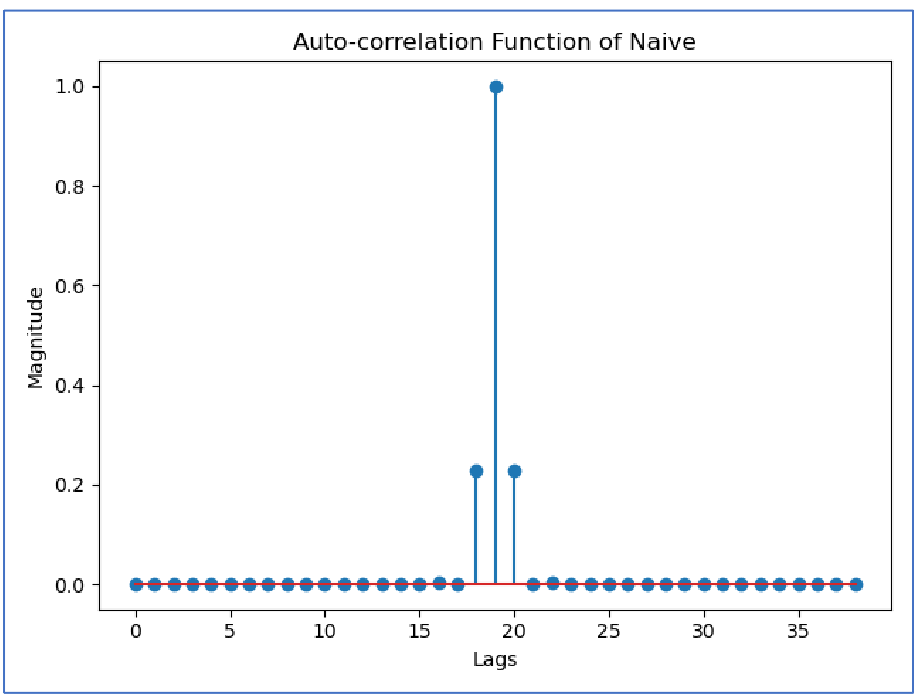
- The plot for h-step forecast of the average method is shown below. The ACF plot of the residuals or prediction errors of the represents white noise or impulse. This indicates that for stationary data most of the underlying trends or patterns were captured by the model.



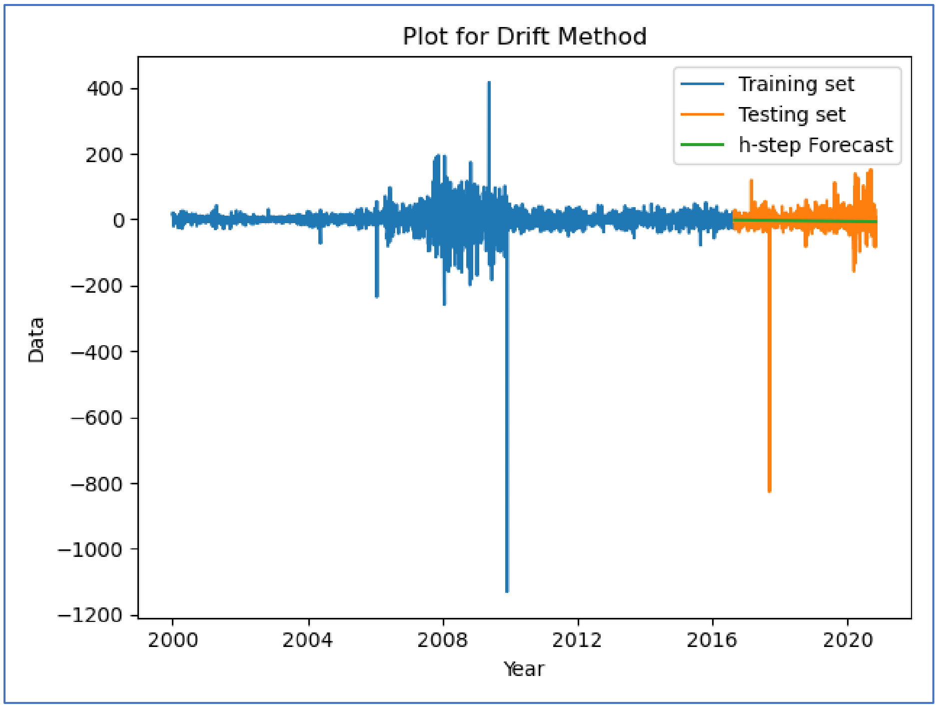


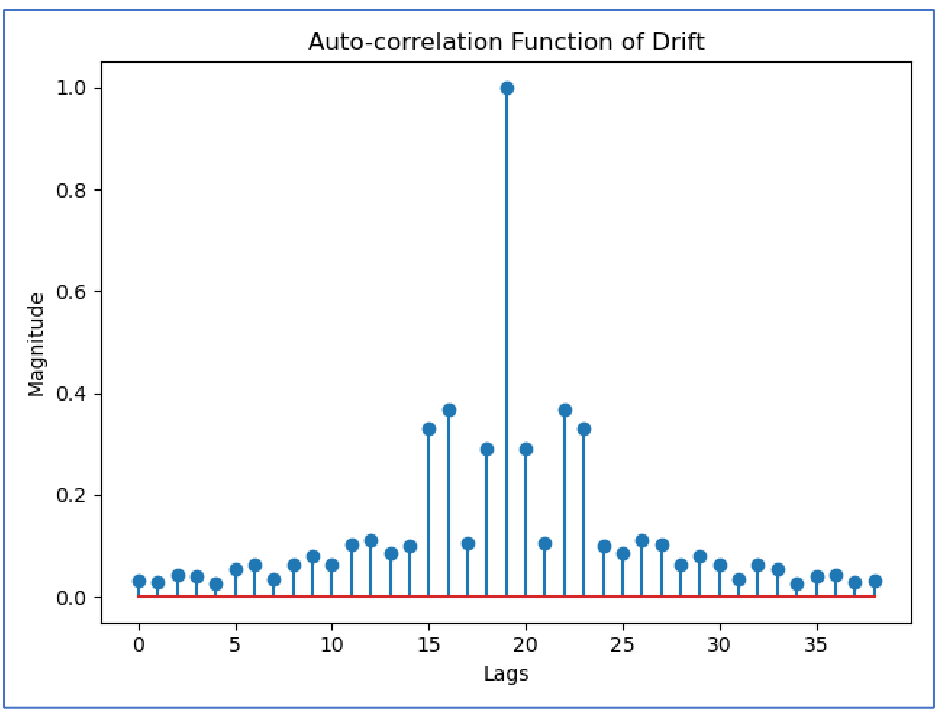
2. Naive Method – The forecast of all future values is equal to the value of the last observation. The naïve method assumes, that the most observation is the only important one, and all previous observations provide no information for the future. They are also called random walk forecasts.
   - The plot for h-step forecast of the naive method is shown below. The ACF plot of the residuals does not represent impulse. This indicates that for stationary data most of the underlying trends or patterns were captured by the model, however, some residual trend or seasonality was not captured.



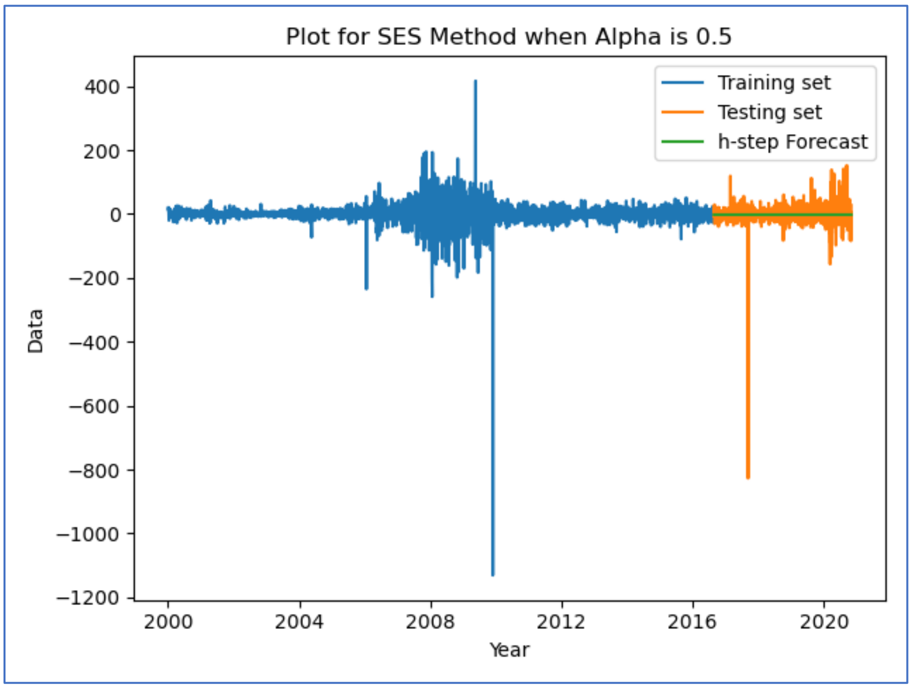


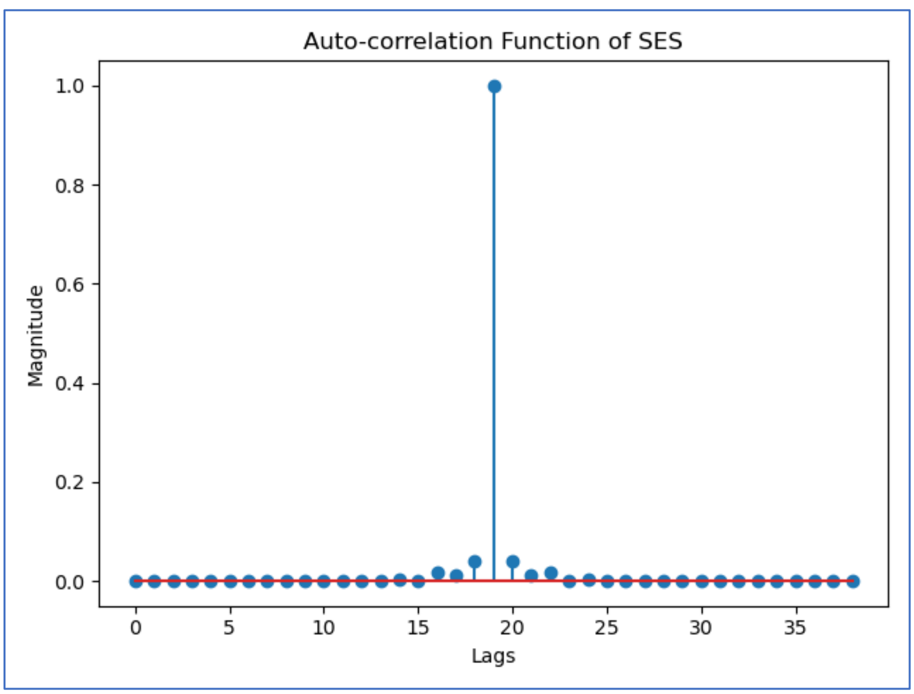
3. <u>Drift Method</u> – The variation on the naïve method is to allow the forecast to increase or decrease over time, where the amount of change over time (called the drift) is set to be the average change seen in the historical data. It is equivalent to drawing a line between the first and last observations and extrapolating it into the future.
    - The plot for h-step forecast of the drift method is shown below. The ACF plot of the residuals does not represent impulse. This indicates that a lot of underlying trends or patterns were not captured by the model, i.e., residual trends or seasonality was not captured.





4. Simple Exponential Smoothing Method – SES method is calculated using weighted averages where the weights decrease exponentially as observations come from further in the past, the smallest weights are associated with the oldest observations
    - The plot for h-step forecast of the SES method is shown below. The ACF plot of the residuals does not represent impulse accurately, however, it indicates that a very few underlying trends or patterns were not captured by the model.

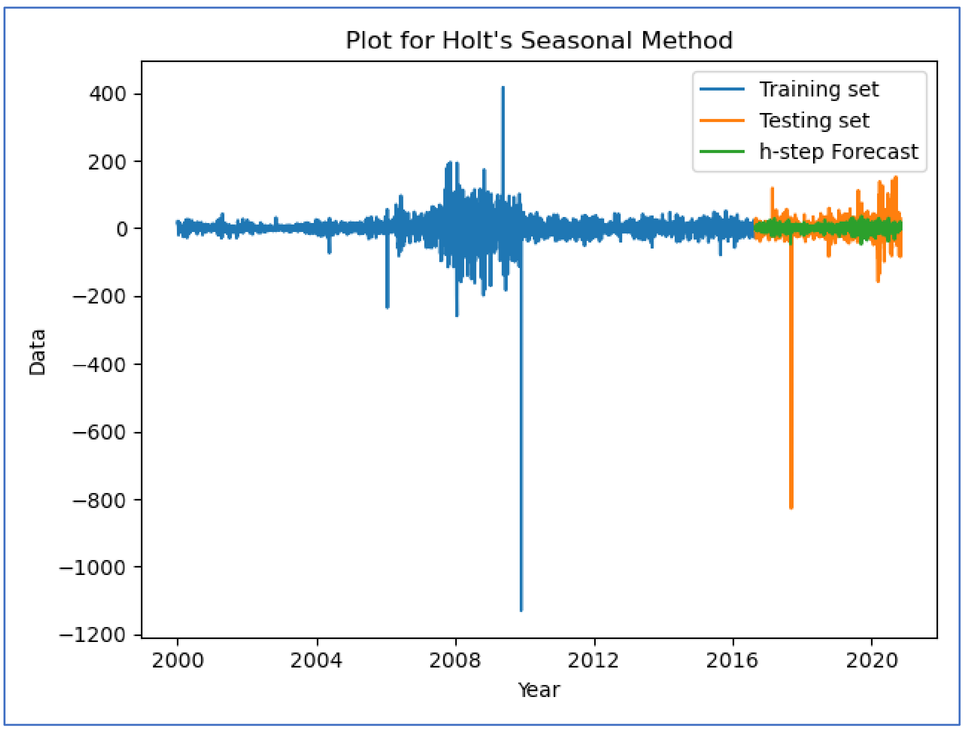


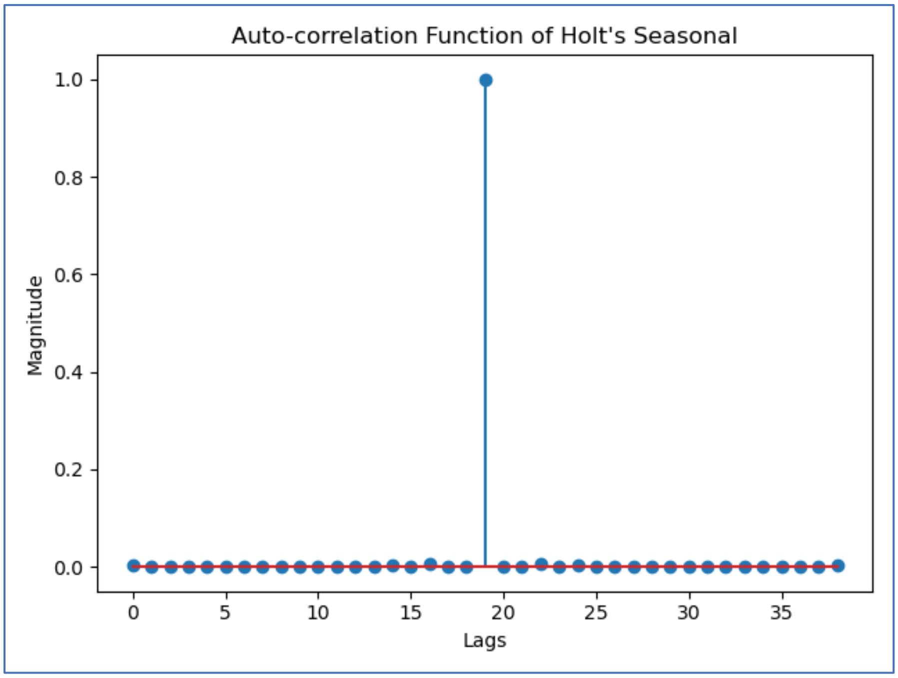


Holt-Winter's Method

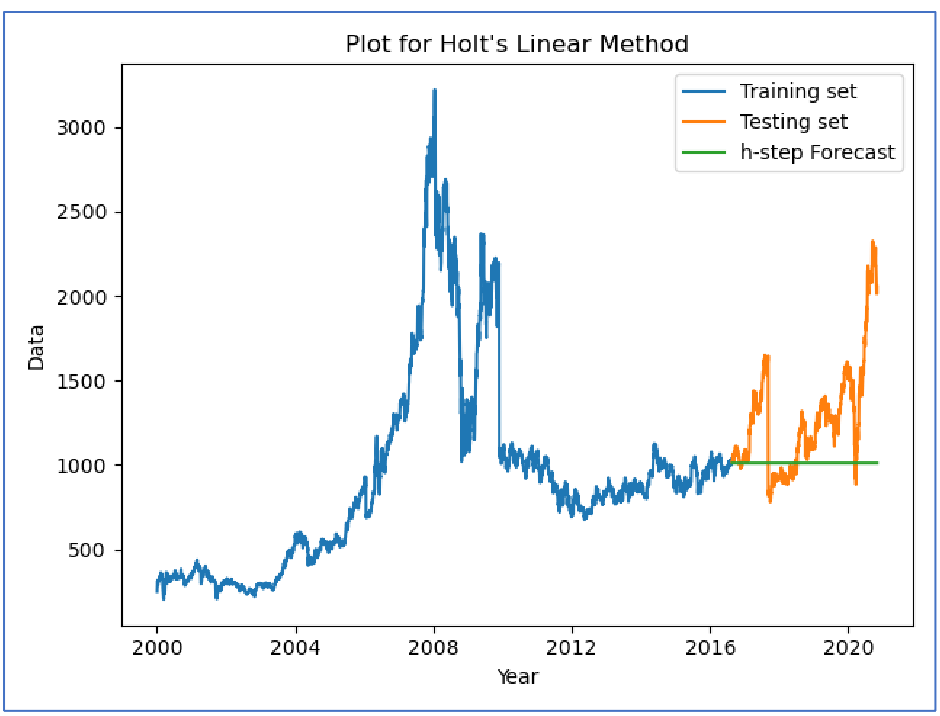
There 2 main methods used for forecasting. They each focus on the different components of the time series and allow for forecasting. By looking at the time series decomposition in the previous sections, it was observed that the data is highly trended and has very less seasonality. Hence, Holt's Linear Trend method is mainly used for forecasting. The Holt-Winter's Seasonal method is included for observation and practice.

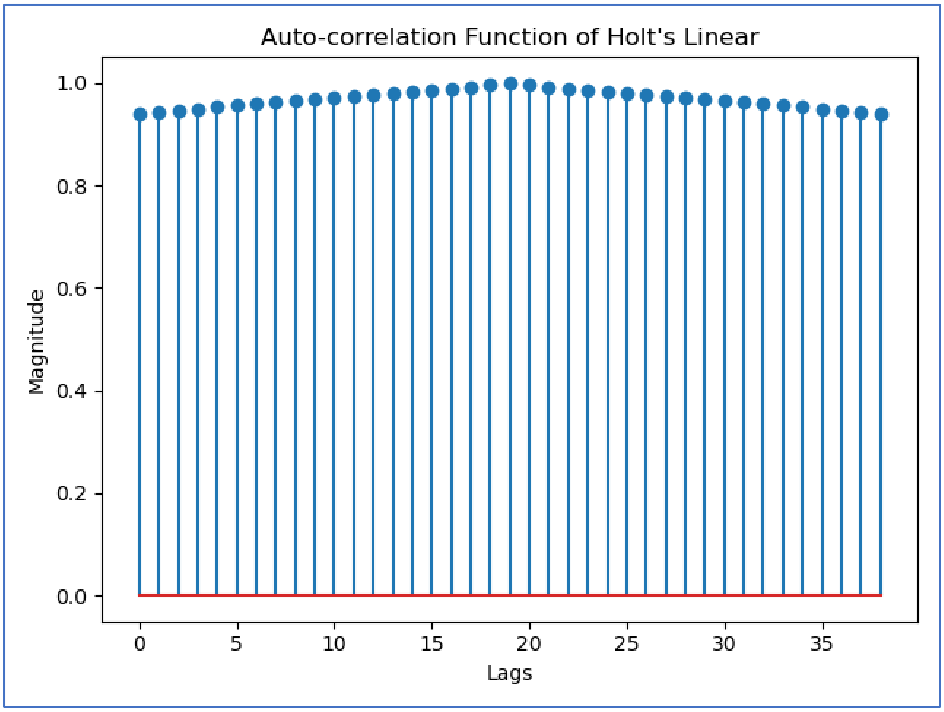
1. <u>Holt-Winter's Seasonal Method</u> – Holt (1975) and Winter (1960) extended Holt's method allows to capture the seasonality. Holt-Winter seasonal method comprises the forecast equation and three smoothing equations: Level (l_t), Trend (b_t) and Seasonal (s_t).
   - For this method, stationary data was used, as it gave better results in terms of the ACF. Since the data is collected daily, the trend and seasonality used is 'additive'. The seasonal period is taken to be 1440. According to the article mentioned in Reference 3, data having daily seasonality have a frequency of 24*60=1440.
   - The plot for h-step forecast of the seasonal method is shown below. The ACF plot of the residuals represents impulse accurately. This indicates that most underlying trends or patterns were captured by the model.



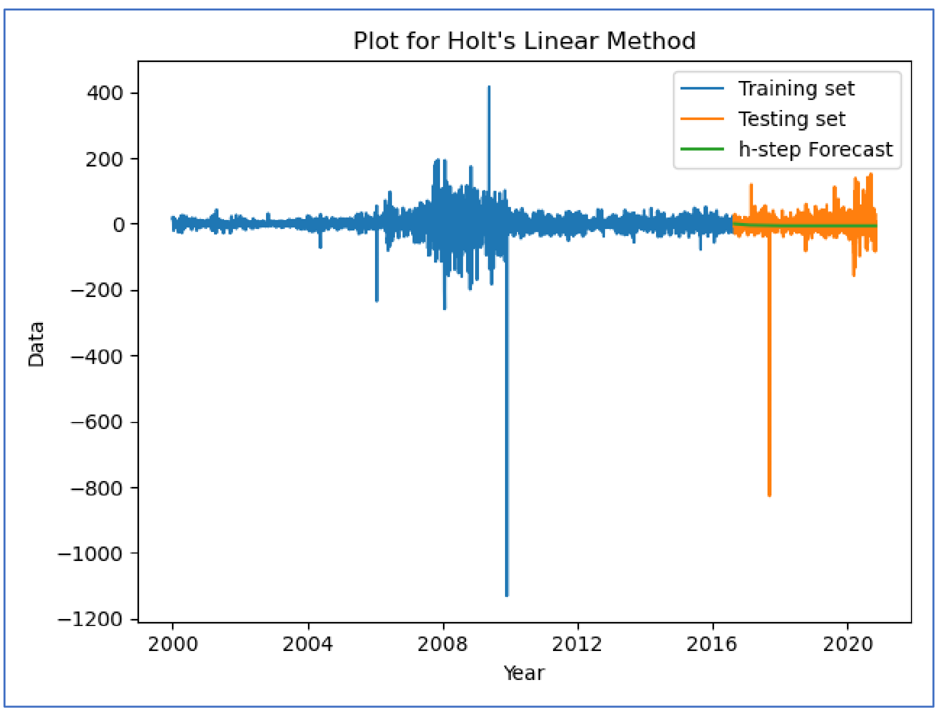


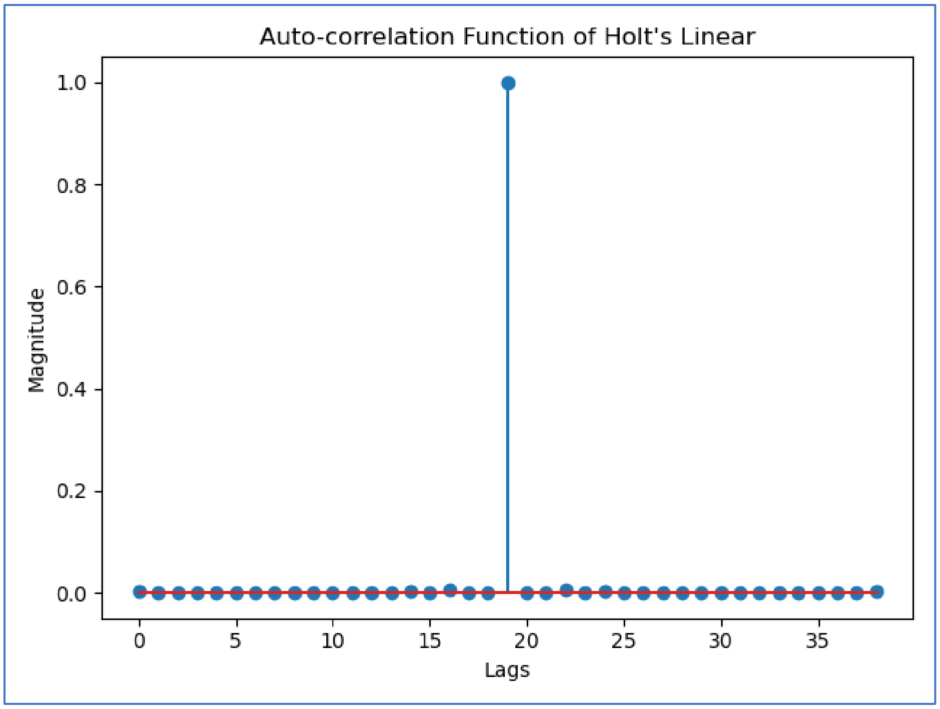
2. <u>Holt's Linear Trend Method</u> – Holt's (1957) extended simple exponential smoothing allows the forecasting of data with trend. This method involves a forecast equation and two smoothing equations (one for level and one for the trend). Since is not a requirement for this method, for observation purposes, modeling was done with both stationary and non-stationary data.
    - <u>Using non-stationary data</u> – The plot for h-step forecast of the linear method is shown below. The ACF plot of the residuals does not represent impulse. This indicates that most underlying trends or patterns were not captured by the model. The Q values and other statistics, however, show lower values compared to the model from the differenced data.





- Using stationary data – The plot for h-step forecast of the linear method is shown below. The ACF plot of the residuals represents impulse. This indicates that most underlying trends or patterns were captured by the model. The Q values and other statistics, however, show extremely high values compared to the model from the non-stationary data.





Feature Selection & Multiple Linear Regression

Feature selection is performed using the Least Squares Estimation method after which, the OLS package is used to perform linear regression. Two types of feature selection methods are used. They are explained as follows.

1. Backward Stepwise Regression – In this method, modeling starts with the model containing all potential predictors and we remove one predictor at a time. Using the AIC, BIC and Adjusted R-squared values, we keep the model that improves the measure of predictive accuracy and iterate until no further improvement.
    - While performing this regression, 4 models were developed, and their statistics are shown below. The aim is to find the model that gives the highest value of the Adjusted R-squared and the lowest AIC and BIC values. From the table below, it is observed that Model 3 performs the best.

USING BACKWARD STEPWISE REGRESSION
The statistics of the 4 models are as follows:

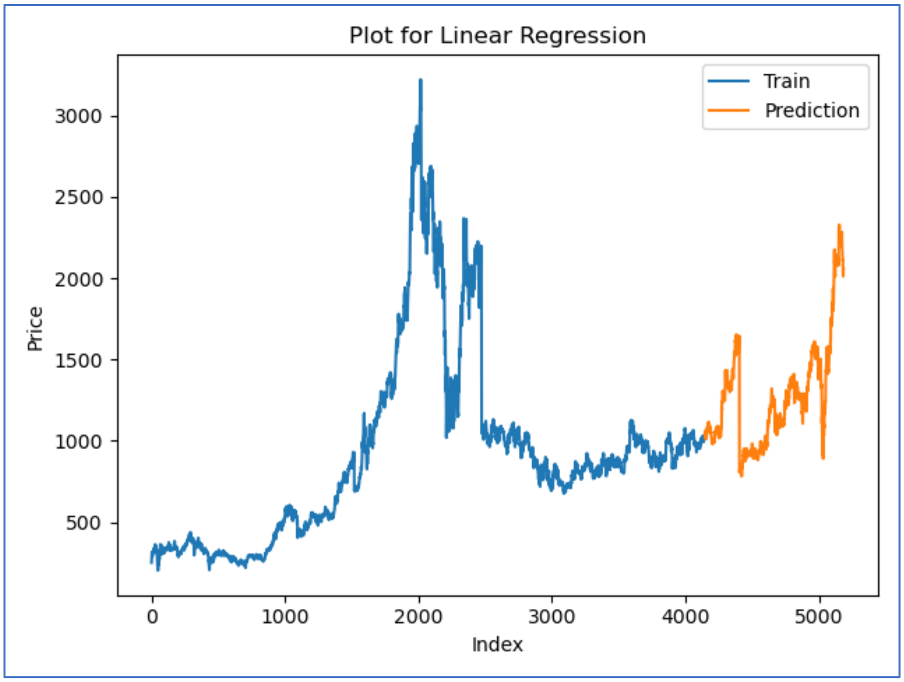
	AIC	BIC	Adj. R-squared
Model 1	21287.784704	21338.425827	0.99997
Model 2	21283.934868	21328.245851	0.99997
Model 3	21283.409106	21321.389949	0.99997
Model 4	21289.499221	21327.480064	0.99997

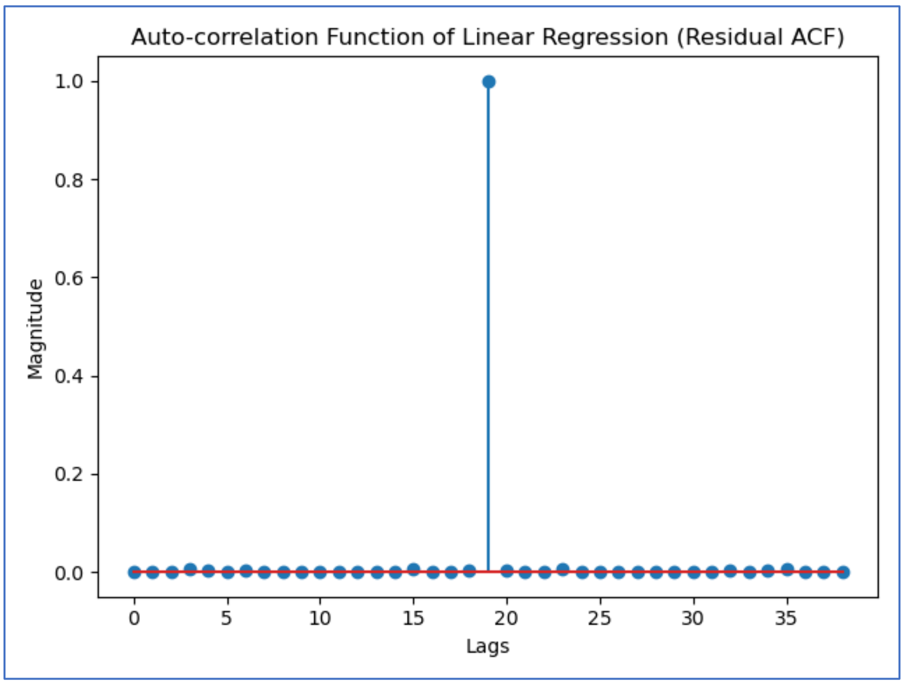
- The summary of Model 3 shows that there are 6 important parameters that are used for multiple linear regression. All the corresponding p-values are less than 0.05, indicating that the coefficients are not 0 (reject the H_0).
- By looking at the F-test results, the p-value is 0.0 (less than the significance value of 0.05). Hence, the model is a better one than the base models.

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                  Close   R-squared:                       1.000
Model:                            OLS   Adj. R-squared:                  1.000
Method:                 Least Squares   F-statistic:                 2.801e+07
Date:                Wed, 09 Dec 2020   Prob (F-statistic):               0.00
Time:                        14:01:24   Log-Likelihood:                -10636.
No. Observations:                4147   AIC:                         2.128e+04
Df Residuals:                    4141   BIC:                         2.132e+04
Df Model:                           5
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const         -0.1216      0.096     -1.264      0.206      -0.310       0.067
Prev Close    -0.0133      0.002     -6.173      0.000      -0.017      -0.009
Open          -0.0300      0.003     -8.601      0.000      -0.037      -0.023
High           0.0177      0.004      4.323      0.000       0.010       0.026
Last           0.8066      0.004    185.413      0.000       0.798       0.815
VWAP           0.2192      0.007     32.753      0.000       0.206       0.232
Turnover   -5.247e-16   2.24e-16     -2.344      0.019   -9.63e-16   -8.58e-17
==============================================================================
Omnibus:                      946.037   Durbin-Watson:                   1.892
Prob(Omnibus):                  0.000   Jarque-Bera (JB):            44403.313
Skew:                          -0.151   Prob(JB):                         0.00
Kurtosis:                      19.028   Cond. No.                     9.72e+14
==============================================================================
```

```
The F-test values are:
F-value: 28012838.222564943 F_p-value: 0.0
```

- The plot for 1-step prediction and train set is shown below. The ACF of the residuals represents white noise, suggesting that the model is a good one and that most underlying information is captured by the model.





2. <u>Forward Stepwise Regression</u> – In this method, modeling starts by adding all the predictors one after the other. Using the AIC, BIC and Adjusted R-squared values, we keep the model that improves the measure of predictive accuracy and iterate until no further improvement.
   - Initially, while performing the regression, all predictors were added in the model including the predictor 'Low'. The model that performed the best was the Model 4, with 'Low' included. However, this model gave p-values much greater than

0.05. Hence, it was removed from the list of predictors and modeling was continued without it.

- Forward regression gave 7 final models, out of which Model 6 was the best one. It had the highest Adjusted R-squared values and the lowest AIC and BIC values.

```
The statistics of the 7 models are as follows:
                   AIC           BIC  Adj. R-squared
Model 1  40643.511024  40656.171305        0.996844
Model 2  38952.401314  38971.391735        0.997901
Model 3  36387.391225  36412.711787        0.998870
Model 5  22298.269707  22329.920409        0.999962
Model 6  21289.499221  21327.480064        0.999970
Model 7  21290.984872  21335.295856        0.999970
Model 8  21286.798857  21331.109840        0.999970
```

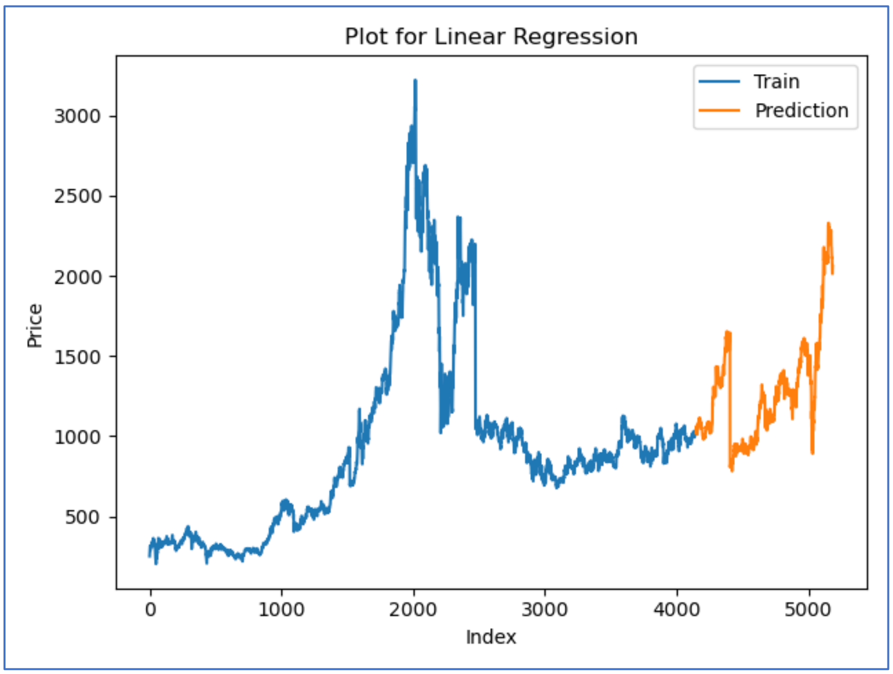
- The summary of Model 6 shows that there are 5 important parameters that are used for multiple linear regression. All the corresponding p-values are less than 0.05, indicating that the coefficients are not 0 (reject the H_0).

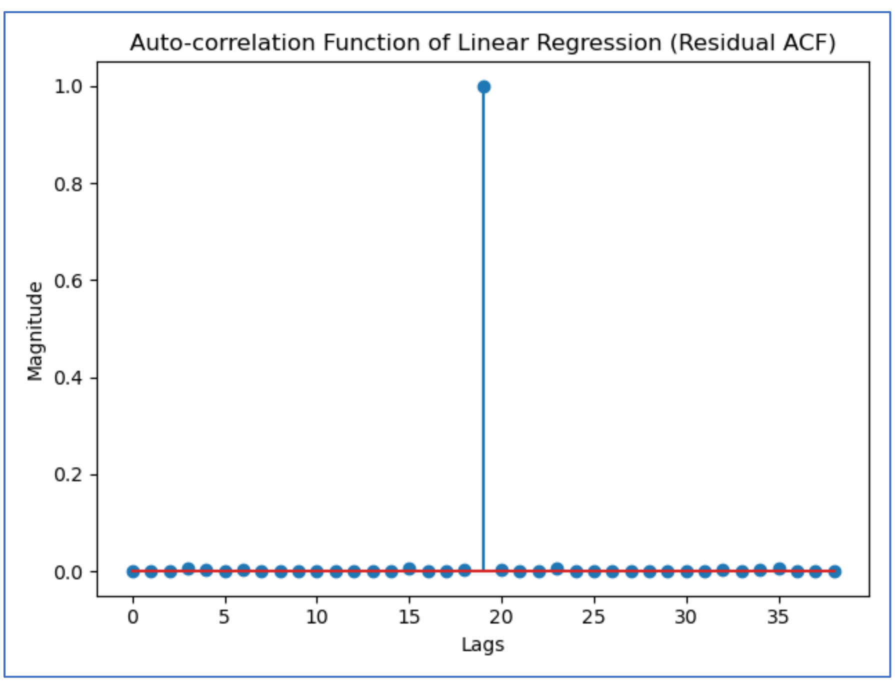
```
                            OLS Regression Results
==============================================================================
Dep. Variable:                  Close   R-squared:                       1.000
Model:                            OLS   Adj. R-squared:                  1.000
Method:                 Least Squares   F-statistic:                 2.797e+07
Date:                Wed, 09 Dec 2020   Prob (F-statistic):               0.00
Time:                        14:10:58   Log-Likelihood:                -10639.
No. Observations:                4147   AIC:                         2.129e+04
Df Residuals:                    4141   BIC:                         2.133e+04
Df Model:                           5
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const         -0.1823      0.093     -1.961      0.050      -0.365   -5.73e-05
Prev Close    -0.0140      0.002     -6.549      0.000      -0.018      -0.010
Open          -0.0282      0.003     -8.260      0.000      -0.035      -0.022
High           0.0132      0.004      3.591      0.000       0.006       0.020
Last           0.8068      0.004    185.359      0.000       0.798       0.815
VWAP           0.2224      0.007     33.807      0.000       0.209       0.235
==============================================================================
Omnibus:                      955.274   Durbin-Watson:                   1.892
Prob(Omnibus):                  0.000   Jarque-Bera (JB):            46883.735
Skew:                          -0.139   Prob(JB):                         0.00
Kurtosis:                      19.470   Cond. No.                     4.62e+03
==============================================================================
```

- By looking at the F-test results, the p-value is 0.0 (less than the significance value of 0.05). Hence, the model is a better one than the base models.

```
The F-test values are:
F-value: 27971728.686665963 F_p-value: 0.0
```

- The plot for 1-step prediction and train set is shown below. The ACF of the residuals represents white noise, suggesting that the model is a good one and that most underlying information is captured by the model.





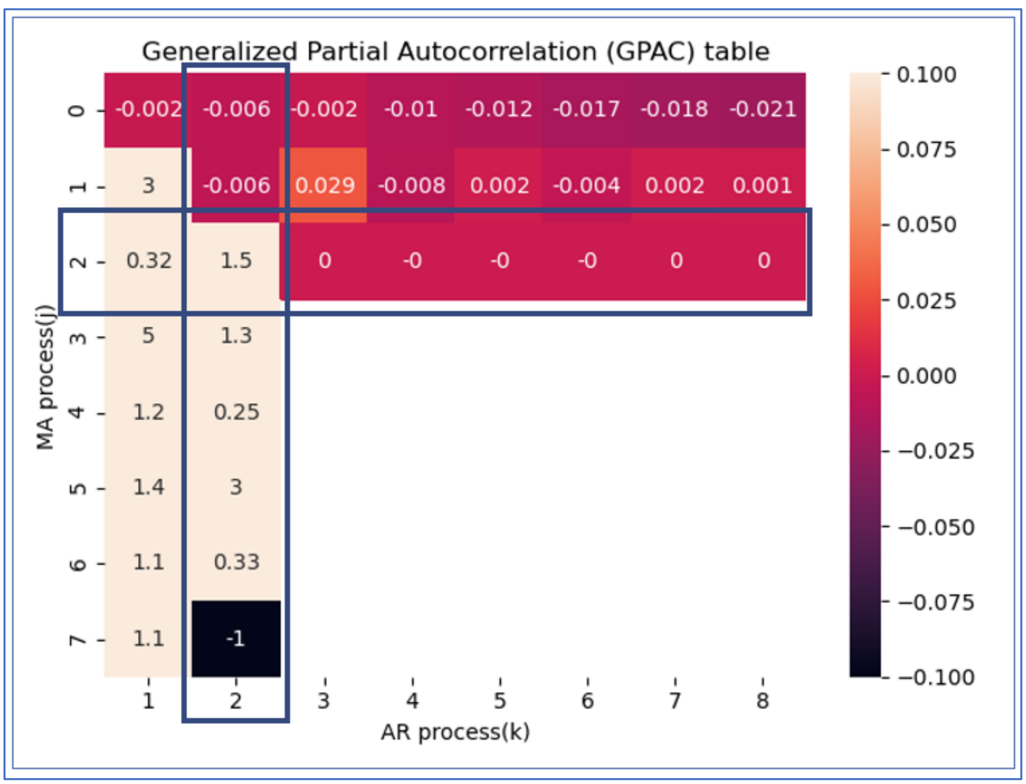
ARMA Models

Autoregressive moving average (ARMA(na, nb)) models are the combination of AR(na) and MA(nb) models. ARMA models provide the most effective linear model of stationary time series since they can model the unknown process with the minimum number of parameters. ARMA used in studying stationary stochastic processes and it is suited for a large class of practical problems.

We can use PACF to find either the na or nb. When both are not equal to 0, we need to use the Generalized Partial Autocorrelation Function or GPAC. It is used to estimate the order of the ARMA(na, nb) process.

The Levenberg Marquardt Algorithm solves nonlinear LSE problems by minimizing the sum of squares of the errors (SSE). It also combines two minimization methods: the gradient decent method and the Gauss-Newton method. It is used for parameter estimation in ARMA process.

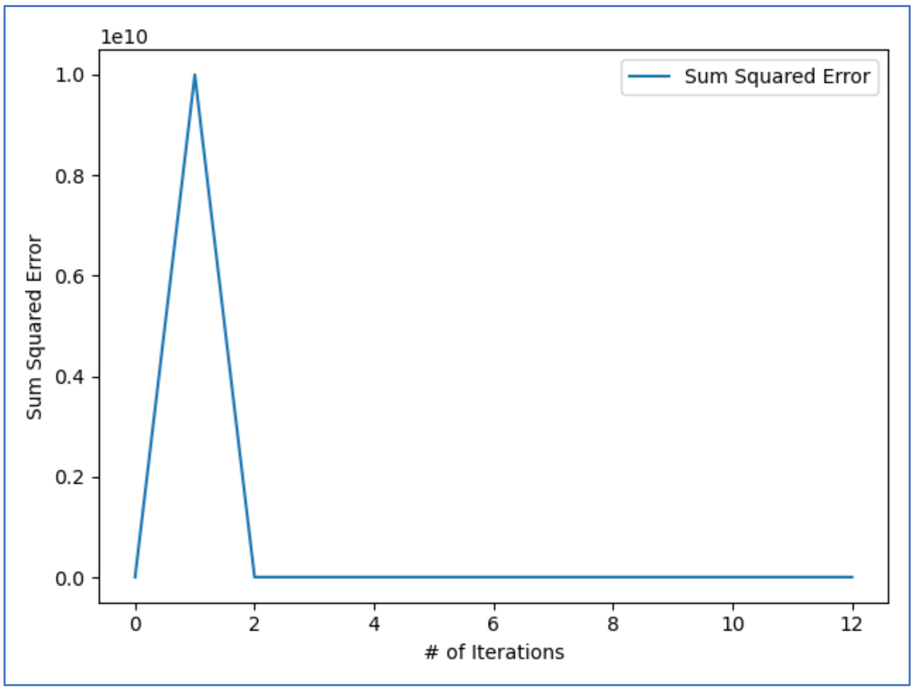
1. <u>Order Estimation</u> – Using the GPAC table we can easily estimate the order(s) of the ARMA process. From the GPAC table below, we can see 2 possible orders of the ARMA process – ARMA(2,2) and ARMA(2,0).



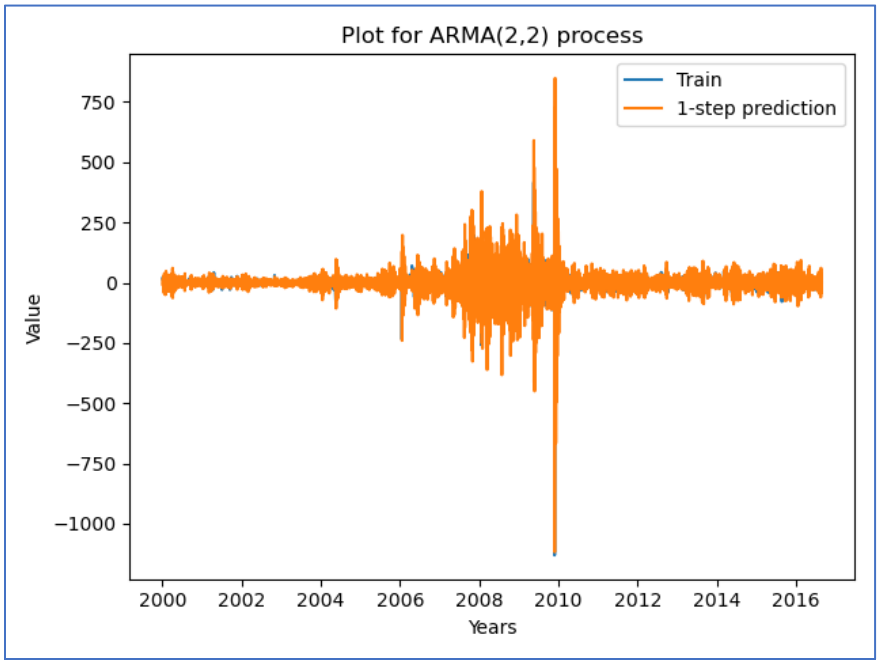
2. <u>ARMA(2,2)</u>

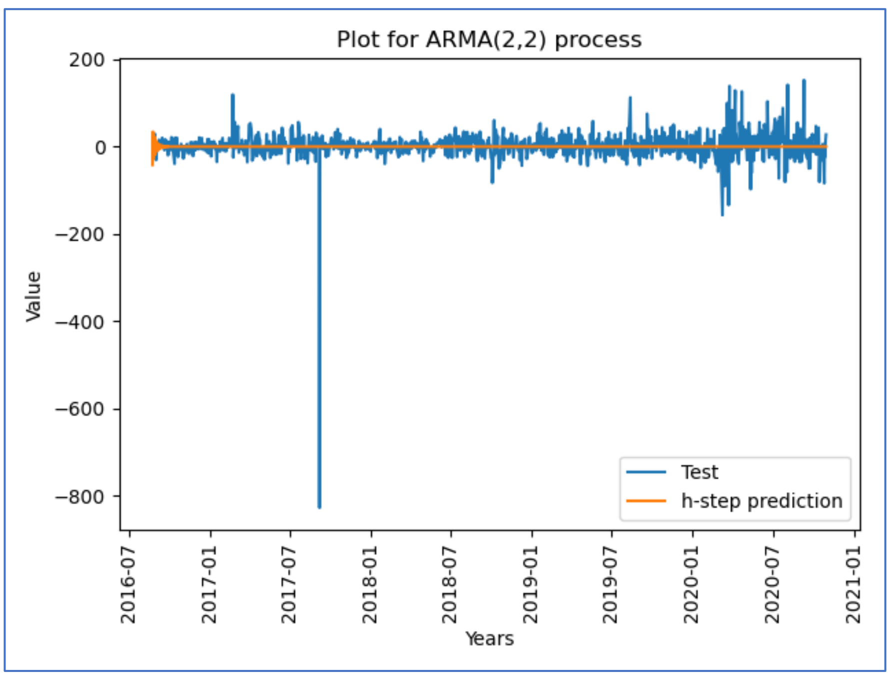
- <u>Parameter estimation</u> – The algorithm successfully converges after about 2 iterations. Looking at the confidence interval of the parameters a1, a2, b1 and b2, we can see that 0 is not included in them, suggesting that the parameters are statistically significant.

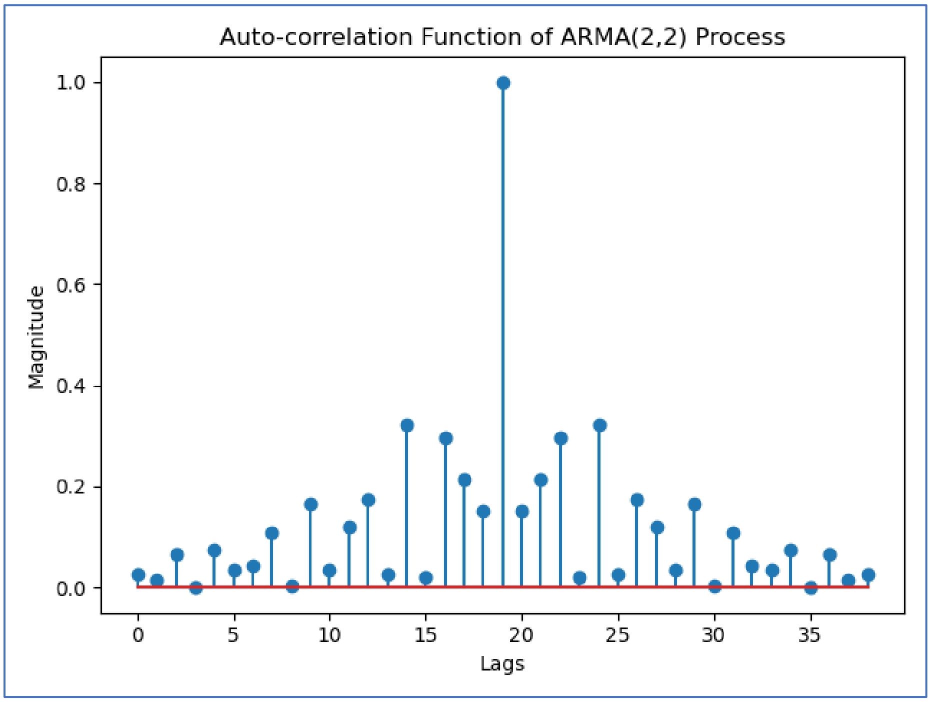
```
 **** Algorithm Converged ****

Estimated parameters :  [0.81311507 0.50060058 0.8592915  0.55665771]
Estimated Covariance matrix :  [[0.02998546 0.01150428 0.02876018 0.01168439]
 [0.01150428 0.0275947  0.01066254 0.02628518]
 [0.02876018 0.01066254 0.02775761 0.01094942]
 [0.01168439 0.02628518 0.01094942 0.02522542]]
Estimated variance of error :  1052.927698923516

 Confidence Interval for Estimated parameters
0.46678884505656726 < a1 < 1.1594413038724123
0.1683675075344147 < a2 < 0.8328336560893512
0.5260791833433514 < b1 < 1.1925038116597622
0.23900744818141784 < b2 < 0.8743079673978972

Zeros :  [-0.42964575+0.60996905j -0.42964575-0.60996905j]
Poles :  [-0.40655754+0.57906092j -0.40655754-0.57906092j]
```



- <u>Modeling</u> – The 1-step and h-step predictions of the ARMA(2,2) process were preformed manually (see appendix for code). From the plot for 1-step prediction it can be observed that the model predicts the next values quite accurately. It may suggest overfitting. The plot for h-step prediction is shown below as well. Looking at the ACF plot of the residuals, the plot does not represent white noise, hence, a lot underlying patterns or information can still be extracted by the model.







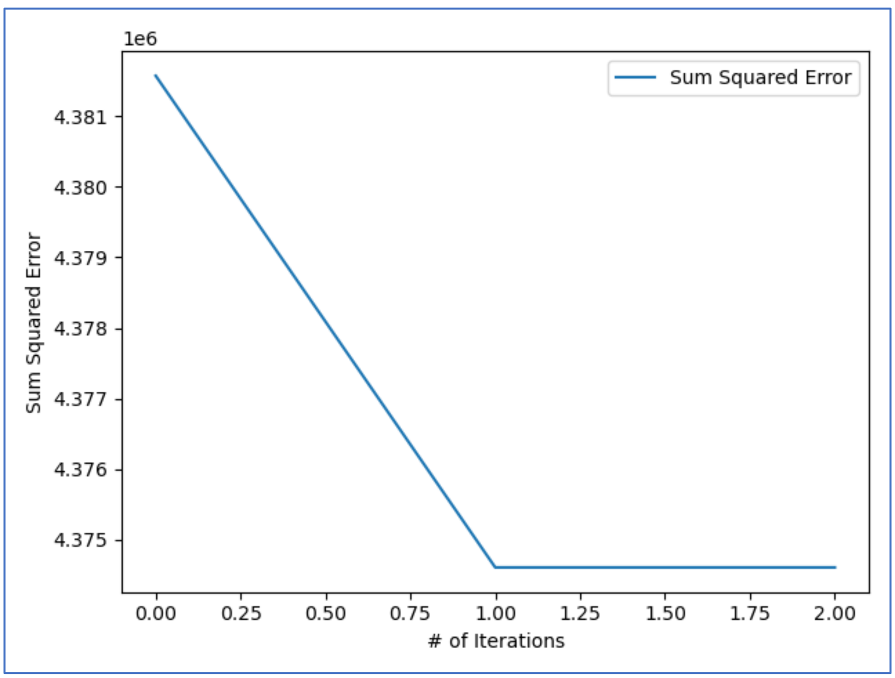
3. ARMA(2,0)
   - Parameter estimation – The algorithm successfully converges after about 1 iteration. Looking at the confidence interval of the parameters a1 and a2, we can see that 0 is included in the interval for a2 suggesting that the parameters are not statistically significant.
   - Hence, we reduce the na value from 2 to 1.

```
 **** Algorithm Converged ****

Estimated parameters :  [-0.0400301   0.00434235]
Estimated Covariance matrix :  [[ 2.41308281e-04 -9.61761613e-06]
 [-9.61761613e-06  2.41308794e-04]]
Estimated variance of error :  1055.6292654535364

 Confidence Interval for Estimated parameters
-0.0710982978563022 < a1 < -0.008961895477442392
-0.026725885766219374 < a2 < 0.03541058257365534

Zeros :  []
Poles :  [0.02001505+0.06278333j 0.02001505-0.06278333j]
```

4. <u>ARMA(1,0)</u>

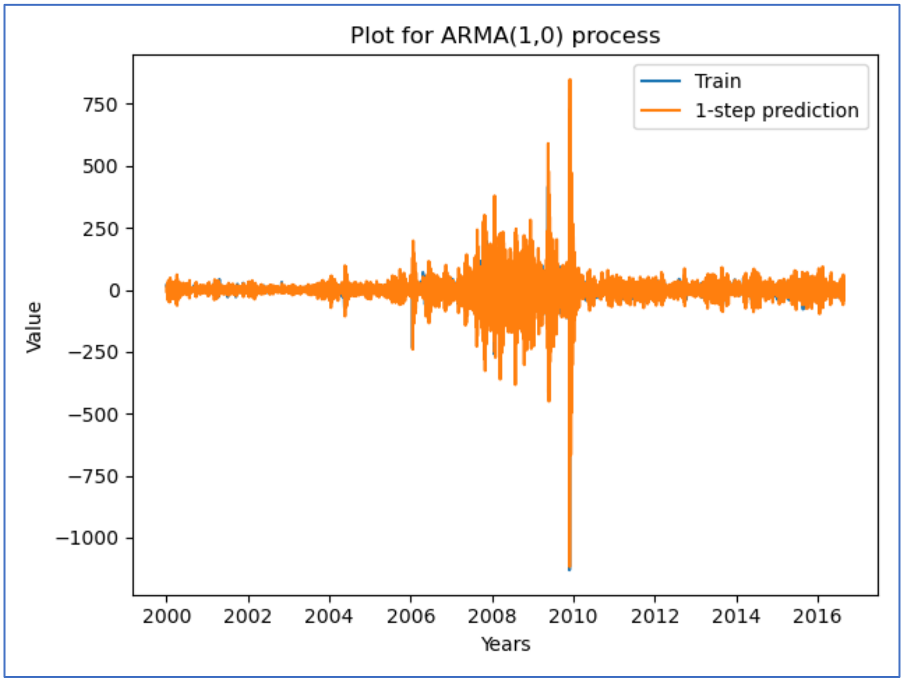
- <u>Parameter estimation</u> – The algorithm successfully converges after 1 iteration. Looking at the confidence interval of the parameter a1, we can see that 0 is not included in it, suggesting that the parameter is statistically significant.

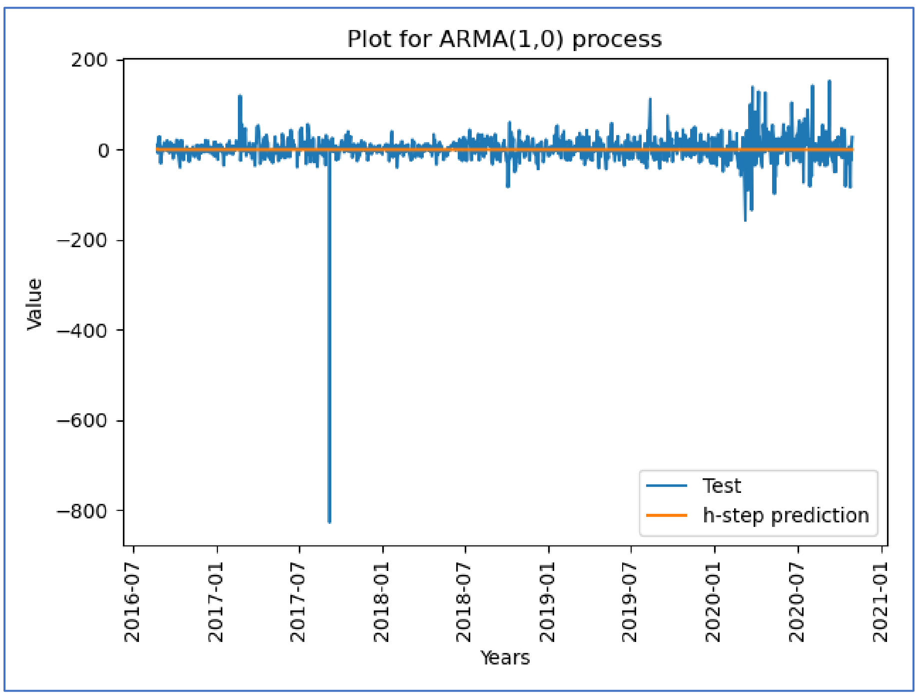
```
 **** Algorithm Converged ****

Estimated parameters :  [-0.03985703]
Estimated Covariance matrix :  [[0.00024087]]
Estimated variance of error :  1055.3944905756664

 Confidence Interval for Estimated parameters
-0.07089709086137862 < a1 < -0.008816964741860233

Zeros :  []
Poles :  [0.03985703]
```

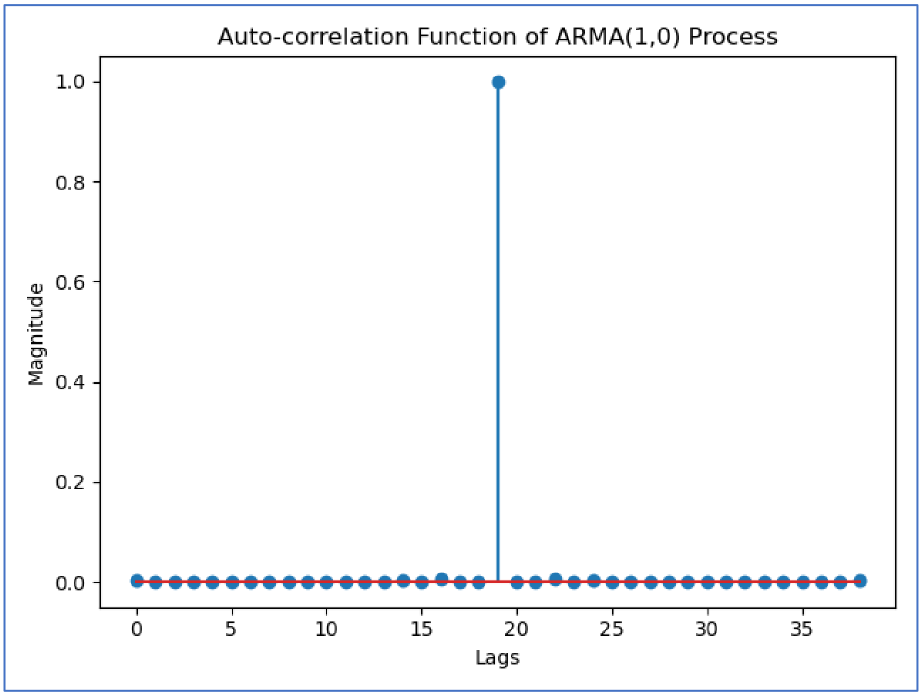


- <u>Modeling</u> – The 1-step and h-step predictions of the ARMA(1,0) process were preformed manually (see appendix for code). From the plot for 1-step prediction it can be observed that the model predicts the next values quite accurately. It may suggest overfitting. The plot for h-step prediction is shown below as well.

Looking at the ACF plot of the residuals, the plot represents white noise, suggesting the model is a good one and captures most underlying information in the data.



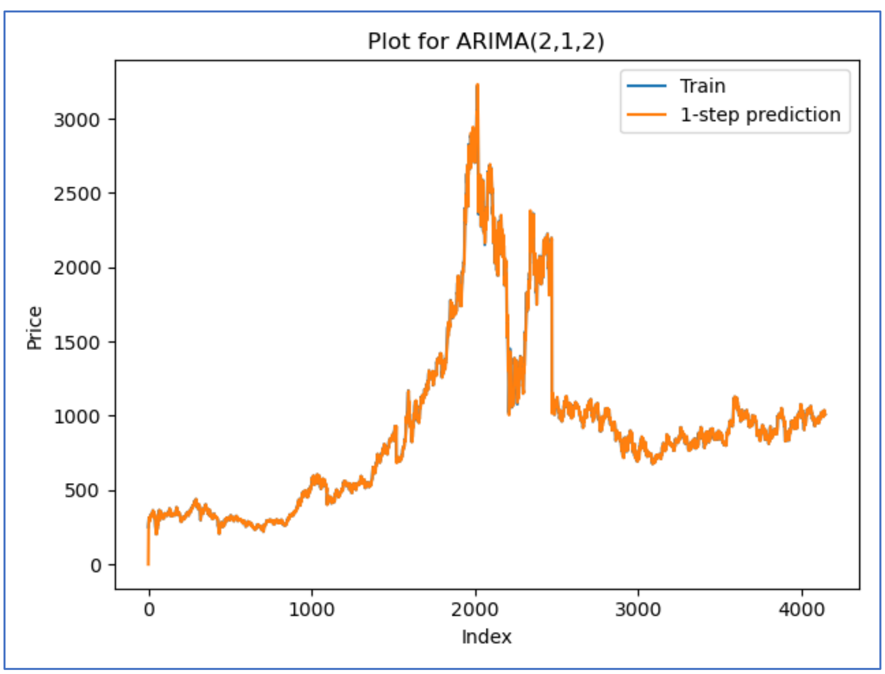


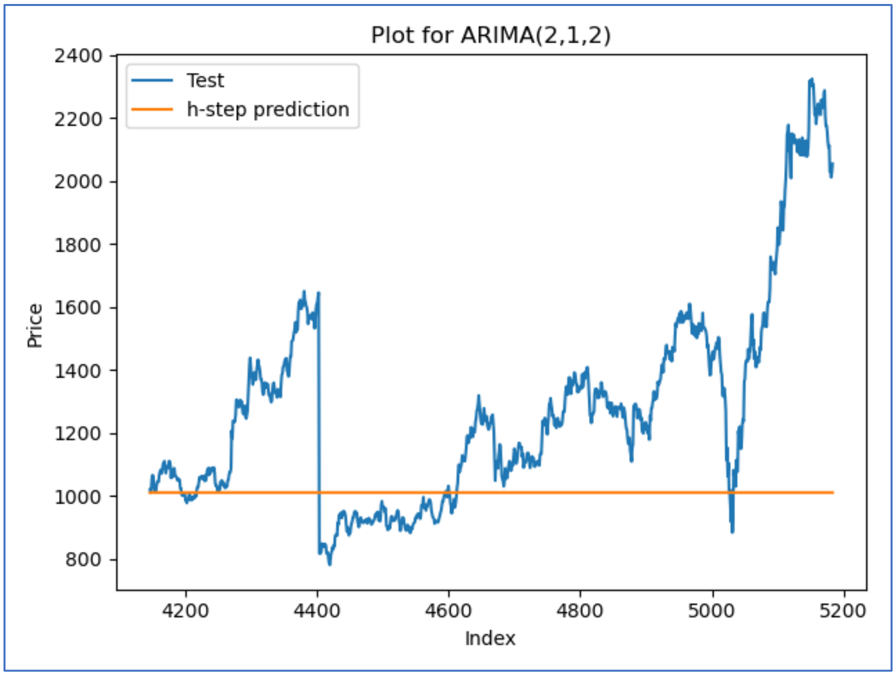


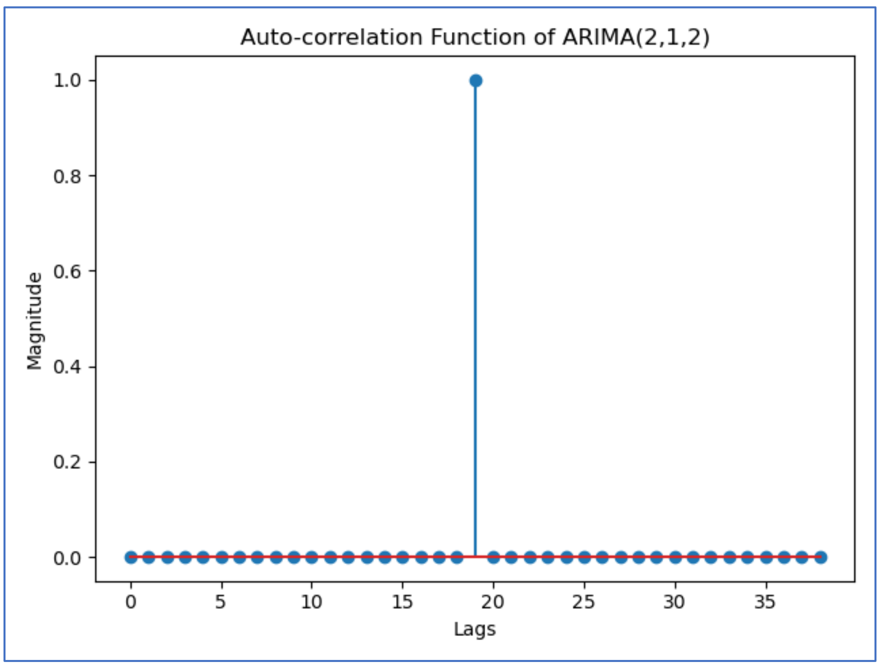
ARIMA Models

ARIMA stands for AutoRegressive Integrated Moving Average. It is a generalization of the simpler AutoRegressive Moving Average and adds the notion of integration. Following the same orders estimated using GPAC, the ARIMA models were created.

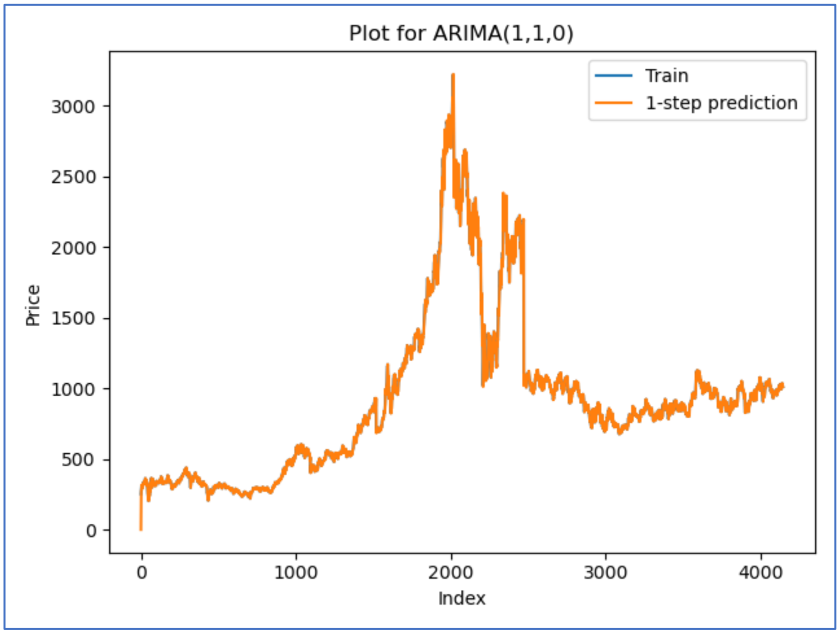
1. ARIMA(2,1,2) – The 1-step and h-step predictions for this model are plotted below. The 1-step prediction can suggest overfitting. Looking at the ACF plot of the residuals, it represents white noise, indicating that the model is a good one and that most of the underlying information is captured by the model.

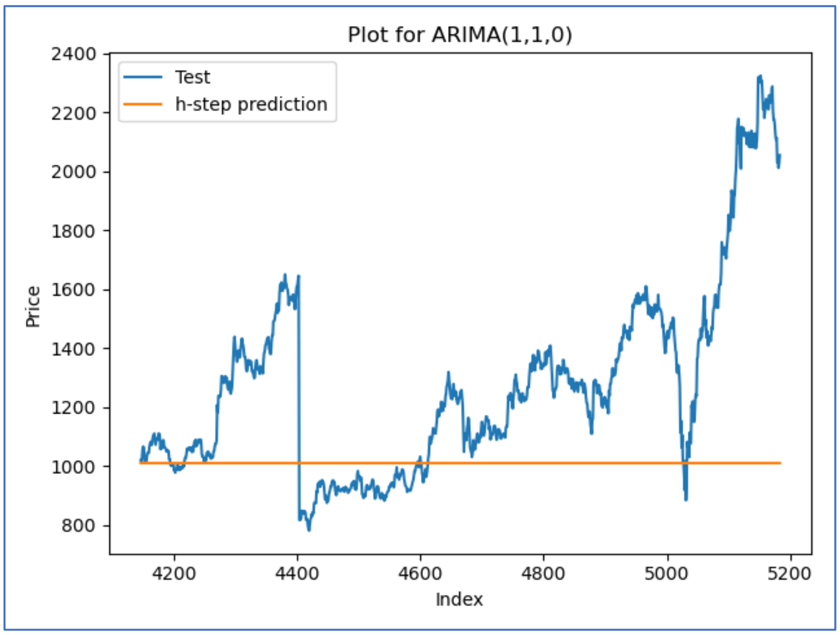


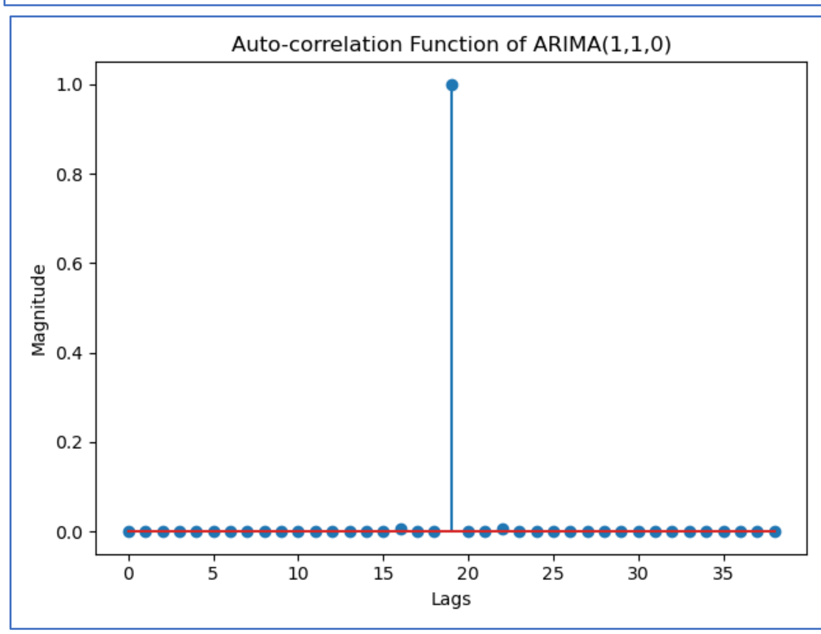




2. ARIMA(1,1,0) – In both models, the 1 is used as a differencing term, which makes the data stationary. The 1-step and h-step predictions for this model are plotted below. The 1-step prediction can suggest overfitting. Looking at the ACF plot of the residuals, it represents white noise, indicating that the model is a good one and that most of the underlying information is captured by the model.


Plot for ARIMA(1,1,0)
Train
1-step prediction
Price
Index
0
500
1000
1500
2000
2500
3000
0
1000
2000
3000
4000


Plot for ARIMA(1,1,0)
Test
h-step prediction
Price
Index
800
1000
1200
1400
1600
1800
2000
2200
2400
4200
4400
4600
4800
5000
5200


Auto-correlation Function of ARIMA(1,1,0)
Magnitude
Lags
0.0
0.2
0.4
0.6
0.8
1.0
0
5
10
15
20
25
30
35

Final Model Selection

The following table shows the various statistics of the different models implemented in the project. By looking at these statistics, we can see that the Average method, Holt's Linear model with non-stationary data, both the multiple linear regression models, ARMA(1,0) and both the ARIMA models have outperformed all the remaining models.

To select the final best model the Q values and MSE of the prediction will be considered. From the table, we can see that the ARIMA(2,1,2) model is the best performing one. This model has the least Q value (0.019) and MSE for prediction (1.055e+03) compared to all models.

STATISTICS

	MSE Prediction	MSE Forecast	Q Values
Average	1.057683e+03	1310.25	0.282761
Naive	2.029785e+03	1315.36	216.193719
Drift	6.689153e-02	1340.71	1702.780837
SES	3.503413e+02	1318.15	8.618003
Holt's Seasonal	1.056916e+03	1431.86	0.283676
Holt's Linear (Diff)	1.172093e+06	179541	73852.003989
Holt's Linear (Non-Diff)	1.056916e+03	1352.69	0.284587
Backward LR	9.891866e+00	N/A	0.459223
Forward LR	9.906404e+00	N/A	0.463528
ARMA(2,2)	5.638543e+03	1316.78	1501.892290
ARMA(1,0)	9.743991e+02	1310.56	0.284455
ARIMA(2,1,2)	1.043592e+03	179426	0.019898
ARIMA(1,1,0)	1.055140e+03	179715	0.245976

	Variance Residuals	Variance Forecast Error
Average	1057.639179	1309.57
Naive	2029.785426	1309.57
Drift	0.065473	1314.65
SES	350.341261	1309.57
Holt's Seasonal	1056.897310	1431.22
Holt's Linear (Diff)	334822.853487	108364
Holt's Linear (Non-Diff)	1056.888667	1310.98
Backward LR	3.148553	N/A
Forward LR	3.150866	N/A
ARMA(2,2)	5638.522352	1315.72
ARMA(1,0)	974.369780	1309.55
ARIMA(2,1,2)	1043.555973	108372
ARIMA(1,1,0)	1055.108961	108372

Summary & Conclusion

The goal of the project, to find the best model for the closing stock price prediction was achieved successfully. The ARIMA model with the order (2,1,2) was seen to be the best performing model. While there were many models that performed similarly, a more robust model such as the ARIMA was a better performing one maybe due to its accountability of the various components of the time series (i.e., Trend in this data).

The SARIMA model was not implemented in this project, mainly because the decomposition of the components suggested highly trended data. Since the SARIMA model caters to the seasonality present in the data, it would not have given desirable outputs. In conclusion, the project was completed successfully.

Future Work

The project covers a wide range of modeling and time series analysis techniques and concepts, however, for stock market given its unpredictability, it would be better to used more advanced models including neural networks such as the LSTM models. Prophet and Auto-ARIMA models are also used commonly today. Further work on the seasonality present in stock market data can be studied for better understanding of the time series.

Appendix

Final_Project-Sharmin_Kantharia.py

```
# IMPORT STATEMENTS
from FinalProjectFunctions import *
from sklearn.model_selection import train_test_split
from statsmodels.tsa.seasonal import STL
import seaborn as sns
import datetime as dt
import warnings
warnings.filterwarnings("ignore")
# ---------------------------------------------------------------------------------
---------

# SECTION 1 - DATA PRE-PROCESSING
# Load data
df = pd.read_csv('RELIANCE.csv', header=0)
# print(df.head())
```

```
# Data description
print('Data Description')
print(df.describe())
print(75*'-')
print(df.columns)
print(75*'-')
print(df.info())
print(75*'-')

# Copy data - for different parts of the project
data = df.copy()
data1 = df.copy()
data2 = df.copy()
data3 = df.copy()

# Format dates
new_date = []
for dat in df['Date']:
    d = dt.datetime.strptime(dat, "%m/%d/%Y")
    d = d.date()
    new_date.append(d.isoformat())
df.index = new_date
df = df.drop(['Date','Symbol','Series','Trades','Deliverable
Volume','%Deliverble'], axis=1)

# Determine 'target' or 'dependent' variable
data['Date'] = pd.to_datetime(data['Date'])
data = data.set_index('Date')
data = data.drop(['Symbol','Series','Trades', 'Deliverable Volume','%Deliverble'],
axis=1)
dep_var = data['Close']

# Correlation matrix and Pearson's correlation coefficient
print('Correlation Matrix')
corr_matrix = df.corr()
print(corr_matrix)
print(75*'-')
corr_plot = sns.heatmap(corr_matrix,vmin=-1,vmax=1,center=0).set_title('Heatmap of
the dataset')
plt.show()

# Plot of dependent variable vs time - original data
plt.figure()
plt.plot(dep_var, label='Closing Price of Stocks')
plt.xlabel('Year')
plt.ylabel('Price per Day')
plt.legend()
plt.title('Dependent Variable ("Close") vs Time')
plt.xticks(rotation=90)
plt.show()

# ACF calculation and plot of original data
lags = np.arange(1,21)
orig_acf = acf_values_df(dep_var,lags)
acf_plot(orig_acf,a='Dependent Variable')

# ADF-test 1 - original data
```

```
print('ADF-test on original dependent variable:')
adf_cal(dep_var)
print(75*'-')

# First Order Differencing
y_diff = dep_var.diff().dropna(axis=0)
y_diff.reset_index()

# ADF-test 2 - differenced data
print('ADF-test on differenced dependent variable:')
adf_cal(y_diff)
print(75*'-')

# Plot of dependent variable vs time - after differencing
plt.figure()
plt.plot(y_diff, label='Differenced Dependent Variable')
plt.xlabel('Year')
plt.ylabel('Magnitude')
plt.title('Dependent Variable vs Time - After First Order Differencing')
plt.xticks(rotation=90)
plt.legend()
plt.show()

# ACF calculation and plot of the differenced data
y_acf = acf_values_df(y_diff,lags)
acf_plot(y_acf,a='Dependent Variable')

# Data splitting into 80% - train set and 20% - test set
data = data.drop(columns='Close')
data = data.join(y_diff).set_index(data.index)
data = data.dropna(axis=0)
y = y_diff.to_frame()
y = y.reset_index()  # resets index and sets 'Date' as a column
train, test = train_test_split(y, shuffle=False, test_size=0.2)
# --------------------------------------------------------------------------------
---------

# SECTION 2 - TIME SERIES DECOMPOSITION

Close = data1['Close'].to_frame()  # to_frame() converts this series to a dataframe

# Apply STL decomposition and plot
stl = STL(Close, period=52)
result = stl.fit()
fig = result.plot()
plt.show()

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

# Plot
plt.figure()
plt.plot(T, label = 'Trend')
plt.plot(S, label = 'Seasonal')
plt.plot(R, label = 'Remainder')
plt.legend()
```

```
plt.title('Trend, Seasonality and Remainder')
plt.xlabel('Years')
plt.ylabel('Values')
plt.show()


# FUNCTION TO CALCULATE STRENGTH
def strength_stats(trend, seasonal, resid):
    ft = np.maximum(0,1 -
(np.var(np.array(resid))/(np.var(np.array(trend+resid)))))
    fs = np.maximum(0,1 -
(np.var(np.array(resid))/(np.var(np.array(seasonal+resid)))))
    return ft, fs

# CALCULATING STRENGTH OF TREND AND SEASONALITY
F_T_mul, F_S_mul = strength_stats(T,S,R)
print('The strength of Trend for data is:', F_T_mul)
print('The strength of Seasonality for data is:', F_S_mul)
print(75*'-')
# ------------------------------------------------------------------------------
---------

# SECTION 3 - BASE MODELS
# Load data and format dates
data1['Date'] = pd.to_datetime(data1['Date'])

# Non-differenced data
new_data = data1[['Date', 'Close']]
train_nd, test_nd = train_test_split(new_data, shuffle=False, test_size=0.2)
# Differenced data - Already set above as train and test

# AVERAGE METHOD
avg_p, avg_f, res_avg, err_avg, mse_p_avg, mse_f_avg, var_p_avg, var_f_avg, \
res_acf_avg, Q_avg = call_avg(train, test, lags, a='Close', b='Date', c='Average')
avg_corr = correlation_coefficient_cal1(err_avg, test['Close'])

# NAIVE METHOD
nai_p, nai_f, res_nai, err_nai, mse_p_nai, mse_f_nai, var_p_nai, var_f_nai, \
res_acf_nai, Q_nai = call_naive(train, test, lags, a='Close', b='Date', c='Naive')
nai_corr = correlation_coefficient_cal1(err_nai, test['Close'])

# DRIFT METHOD
dri_p, dri_f, res_dri, err_dri, mse_p_dri, mse_f_dri, var_p_dri, var_f_dri, \
res_acf_dri, Q_dri = call_drift(train, test, lags, a='Close', b='Date', c='Drift')
dri_corr = correlation_coefficient_cal1(err_dri, test['Close'])

# SES METHOD
ses_p, ses_f, res_ses, err_ses, mse_p_ses, mse_f_ses, var_p_ses, var_f_ses, \
res_acf_ses, Q_ses = call_ses(train, test, 0.5, lags, a='Close', b='Date', c='SES')
ses_corr = correlation_coefficient_cal1(err_ses, test['Close'])
# ------------------------------------------------------------------------------
---------

# HOLT-WINTER'S METHOD
# HOLT'S SEASONAL METHOD
hs_p, hs_f, res_hs, err_hs, mse_p_hs, mse_f_hs, var_p_hs, var_f_hs, \
res_acf_hs, Q_hs = call_holt_s(train,test,lags,a='Close',b='Date',c="Holt's
```

```
Seasonal")
hs_corr = correlation_coefficient_cal1(err_hs, test['Close'])

# HOLT'S LINEAR METHOD - Non-Differenced Data
hl_p, hl_f, res_hl, err_hl, mse_p_hl, mse_f_hl, var_p_hl, var_f_hl, \
res_acf_hl, Q_hl = call_holt_l(train_nd,test_nd,lags,a='Close',b='Date',c="Holt's
Linear")
hl_corr = correlation_coefficient_cal1(err_hl, test_nd['Close'])

# HOLT'S LINEAR METHOD - Differenced Data
hl_p_d, hl_f_d, res_hl_d, err_hl_d, mse_p_hl_d, mse_f_hl_d, var_p_hl_d, var_f_hl_d,
\
res_acf_hl_d, Q_hl_d = call_holt_l(train,test,lags,a='Close',b='Date',c="Holt's
Linear")
hl_corr_d = correlation_coefficient_cal1(err_hl_d, test['Close'])
# ---------------------------------------------------------------------------------
---------

# MULTIPLE LINEAR REGRESSION
# Load data
data2 = data2.drop(['Date','Symbol','Series','Trades', 'Deliverable
Volume','%Deliverble'], axis=1)

# Select target
target = 'Close'
y = data2[target]
X = data2.drop(columns=target)

# Split data
X_train, X_test, y_train, y_test = train_test_split(X, y, shuffle=False,
test_size=0.2)

# Create correlation matrix to find important features
# Update X_train and X_test with important features
corr_mat = df.corr()
features = corr_mat.index
new_features = features.drop(target,1)
X_train = X_train[new_features]
X_test = X_test[new_features]
# print(X_train.head(), X_train.columns)  # (4147, 8)
# print(X_test.head(), X_test.columns)  # (1037, 8)

# Create arrays using the split sets
y_trainarr = np.array(y_train).reshape(len(y_train),1)
y_testarr = np.array(y_test).reshape(len(y_test),1)
X_trainarr = X_train.to_numpy()
X_testarr = X_test.to_numpy()

# Add constant and begin modeling
X = sm.add_constant(X_train)
model1 = sm.OLS(y_train,X).fit()
# print(model1.summary())

# Feature Selection - Backward Stepwise Regression
print('BACKWARD STEPWISE REGRESSION')
# Drop Volume - based on the summary of model 1 it was seen that the p val of
Volume is 0.769 > 0.01 or 0.05
```

```
model2, X_train2 = backward_reg(X_train,y_train,drop_feature='Volume')
# Drop Low - based on model 2 summary - the p valy (0.224) > 0.01 or 0.05
model3,X_train3 = backward_reg(X_train2,y_train,drop_feature='Low')
# Drop Turnover - p-val is 0.019 based on the model 3 summary
model4, X_train4 = backward_reg(X_train3,y_train,drop_feature='Turnover')

# Table with information
pd.set_option('display.max_rows', None,'display.max_columns', None)
stat_table = pd.DataFrame({'AIC': [model1.aic,model2.aic,model3.aic,model4.aic],
                           'BIC': [model1.bic,model2.bic,model3.bic,model4.bic],
                           'Adj. R-squared':
[model1.rsquared_adj,model2.rsquared_adj,model3.rsquared_adj,
                                              model4.rsquared_adj]},
                          index=['Model 1','Model 2','Model 3','Model 4'])
print('The statistics of the models are as follows:')
print(stat_table)
print(75*'-')

# Statistics of final model 3
print('The statistics of Model 3 are as follows:')
print(model3.summary())
print('The F-test values are:')
print('F-value:', model3.fvalue,'F_p-value:', model3.f_pvalue)
print(75*'-')

# Modeling
fitted_val = model3.fittedvalues  # to help calculate residuals
# print('The Coefficients of Regression are:')
# print(model3.params)
x_test_new = X_test.drop(['Low','Volume'], axis=1)
X_1 = sm.add_constant(x_test_new)
prediction = model3.predict(X_1)

# Plot for 1-step prediction
plt.figure()
plt.plot(y_train, label='Train')
# plt.plot(y_test, label='Test')
plt.plot(prediction, label='Prediction')
plt.legend()
plt.xlabel('Index')
plt.ylabel('Price')
plt.title('Plot for Linear Regression')
plt.show()

# Model statistics
pred_err_lr = residual(y_train,fitted_val)
mse_pred_lr = mse_mlr(pred_err_lr)
acf_val_predn_lr = acf_values(pred_err_lr,lags)
acf_plot(acf_val_predn_lr,a='Linear Regression (Residual ACF)')
pred_var_lr = var_linreg(pred_err_lr,len(new_features))
Q_back = Q_val(y_trainarr, acf_val_predn_lr)

# Feature Selection - Forward Stepwise Regression
# Create a list of all the columns
feature_list = []
for name in df[new_features].columns:
    feature_list.append(name)
```

```
print('FORWARD STEPWISE REGRESSION')
X_trainf = pd.DataFrame()
# add Prev Close
modelf1, X_trainf1 =
forward_reg(y_train,X_train,X_trainf,feature_list[0],name='Prev Close')
# add Open
modelf2, X_trainf2 =
forward_reg(y_train,X_train,X_trainf1,feature_list[1],name='Open')
# add High
modelf3, X_trainf3 =
forward_reg(y_train,X_train,X_trainf2,feature_list[2],name='High')
# add Low - this model was not taken into account because it gave p-values > 0.05
# modelf4, X_trainf4 =
forward_reg(y_train,X_train,X_trainf3,feature_list[3],name='Low')
# print(modelf4.summary())
# add Last
modelf5, X_trainf5 =
forward_reg(y_train,X_train,X_trainf3,feature_list[4],name='Last')
# add VWAP
modelf6, X_trainf6 =
forward_reg(y_train,X_train,X_trainf5,feature_list[5],name='VWAP')
# add Volume
modelf7, X_trainf7 =
forward_reg(y_train,X_train,X_trainf6,feature_list[6],name='Volume')
# add Turnover
modelf8, X_trainf8 =
forward_reg(y_train,X_train,X_trainf7,feature_list[7],name='Turnover')

# Table with information
pd.set_option('display.max_rows', None,'display.max_columns', None)
stat_table = pd.DataFrame({'AIC':
[modelf1.aic,modelf2.aic,modelf3.aic,modelf5.aic,modelf6.aic,modelf7.aic,modelf8.ai
c],
                           'BIC':
[modelf1.bic,modelf2.bic,modelf3.bic,modelf5.bic,modelf6.bic,modelf7.bic,modelf8.bi
c],
                           'Adj. R-squared':
[modelf1.rsquared_adj,modelf2.rsquared_adj,modelf3.rsquared_adj,modelf5.rsquared_ad
j,modelf6.rsquared_adj,modelf7.rsquared_adj,modelf8.rsquared_adj]},
                          index=['Model 1','Model 2','Model 3','Model 5','Model
6','Model 7','Model 8'])
print('The statistics of the models are as follows:')
print(stat_table)
print(75*'-')

# Statistics of final model 6
print('The statistics of Model 6 are as follows:')
print(modelf6.summary())
print('The F-test values are:')
print('F-value:', modelf6.fvalue,'F_p-value:', modelf6.f_pvalue)
print(75*'-')

# Modeling
fitted_val1 = modelf6.fittedvalues  # to help calculate residuals
x_test_new1 = X_test.drop(['Low','Volume','Turnover'], axis=1)
X_1 = sm.add_constant(x_test_new1)
```

```
prediction1 = modelf6.predict(X_1)

# Plot for 1-step prediction
plt.figure()
plt.plot(y_train, label='Train')
plt.plot(prediction1, label='Prediction')
plt.legend()
plt.xlabel('Index')
plt.ylabel('Price')
plt.title('Plot for Linear Regression')
plt.show()

# Model statistics
pred_err1_lr = residual(y_train,fitted_val1)
mse_pred1_lr = mse_mlr(pred_err1_lr)
acf_val_predn1_lr = acf_values(pred_err1_lr,lags)
acf_plot(acf_val_predn1_lr,a='Linear Regression (Residual ACF)')
pred_var1 = var_linreg(pred_err1_lr,len(new_features))
Q_forw = Q_val(y_trainarr, acf_val_predn1_lr)
# ---------------------------------------------------------------------------------
---------

# ARMA() PROCESS - INCLUDING GPAC AND LM ALGORITHM
# Load data
data3['Date'] = pd.to_datetime(data3['Date'])

# Pre-process data
data3 = data3.set_index('Date')
data3 = data3.drop(['Symbol','Series','Trades', 'Deliverable
Volume','%Deliverble'], axis=1)
dep_var = data3['Close']
y_diff = dep_var.diff().dropna(axis=0)
y_diff.reset_index()

data3 = data3.drop(columns='Close')
data3 = data3.join(y_diff).set_index(data3.index)
data3 = data3.dropna(axis=0)
train, test = train_test_split(y_diff, shuffle=False, test_size=0.2)

# ACF calculation
y_acf = acf_values_df(train,lags)

# GPAC
acf_list = list(sm.tsa.stattools.acf(y_acf))
gpac_cal(acf_list,8,8)

# LM Algorithm
def step_0(na,nb):
    theta = np.zeros(shape=(na+nb,1))
    return theta.flatten()

def white_noise_simulation(theta,na,y):
    num = [1] + list(theta[na:])
    den = [1] + list(theta[:na])
    while len(num) < len(den):
        num.append(0)
    while len(num) > len(den):
```

```
            den.append(0)
        system = (den, num, 1)
        tout, e = signal.dlsim(system, y)
        e = [a[0] for a in e]
        return np.array(e)

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

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

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


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

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

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

            if SSE_new < SSE_old:
                print('Norm of delta_theta :', np.linalg.norm(theta_d))
                if np.linalg.norm(theta_d) < 1e-3:
                    theta_hat = theta_new
                    e_var = SSE_new / (len(y) - A.shape[0])
                    cov = e_var * np.linalg.inv(A)
```

```
                print('\n **** Algorithm Converged **** \n')
                return SSE, theta_hat, cov, e_var
            else:
                theta = theta_new
                mu = mu / 10

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

        theta = theta_new

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


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


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

# ARMA(2,2)
# Estimating parameters using LM algorithm
na = 2
nb = 2
SSE, est_params, cov, e_var = step_3(max_iterations, mu_max, na, nb, train, mu,
delta)
print('Estimated parameters : ', est_params)
print('Estimated Covariance matrix : ', cov)
print('Estimated variance of error : ', e_var)

# SSE Plot
SSEplot(SSE)
confidence_interval(cov, na, nb, est_params)
zeros, poles = zeros_and_poles(est_params, na, nb)

# 1-step ahead prediction
y_hat_t_1 = []
for i in range(0,len(train)):
    if i==0:
```

```
        y_hat_t_1.append(-train[i]*est_params[0] + est_params[1]* train[i])
    elif i==1:
        y_hat_t_1.append(-train[i]*est_params[0] + est_params[1]*(train[i] -
y_hat_t_1[i - 1] ) + est_params[2]*(train[i - 1]))
    else:
        y_hat_t_1.append( -train[i]*est_params[0] + est_params[1]*(train[i] -
y_hat_t_1[i - 1] ) + est_params[2]*(train[i - 1] - y_hat_t_1[i-2]))

train_plot = train.to_frame()
predn = dataframe_create_arma(y_hat_t_1,train_plot,a='Close')
plt.figure()
plt.plot(train, label='Train')
plt.plot(predn, label='1-step prediction')
plt.legend()
plt.xlabel('Years')
plt.ylabel('Value')
plt.title('Plot for ARMA(2,2) process')
plt.show()

# h-step ahead prediction
y_hat_t_h = []
for h in range(0,len(test)):
    if h==0:
        y_hat_t_h.append(-train[-1]*est_params[0] + est_params[1]*(train[-1] -
y_hat_t_1[-2]) + est_params[2]*(train[-2]-y_hat_t_1[-3]))
    elif h==1:
         y_hat_t_h.append(-y_hat_t_h[h-1]*est_params[0] +
est_params[1]*(y_hat_t_h[-1] - y_hat_t_1[-1]))
    else:
        y_hat_t_h.append(-y_hat_t_h[h-1]*est_params[0])

test_plot = test.to_frame()
forec = dataframe_create_arma(y_hat_t_h,test_plot,a='Close')
plt.figure()
plt.plot(test, label='Test')
plt.plot(forec, label='h-step prediction')
plt.legend()
plt.xlabel('Years')
plt.ylabel('Value')
plt.xticks(rotation='vertical')
plt.title('Plot for ARMA(2,2) process')
plt.show()

# Model statistics
pred_err = residual_arma(train, y_hat_t_1)
forec_err = forecast_err_arma(test, y_hat_t_h)
mse_pred, mse_forec = mse(pred_err, forec_err)
var_pred, var_forec = var(pred_err, forec_err)
residual_acf = acf_values(pred_err, lags)
Q_val1 = Q_val(train, residual_acf)
acf_plot(residual_acf, a='ARMA(2,2) Process')
chi_square_test(Q_val1, 20, na, nb)

# ARMA(1,0)
# Estimating parameters using LM algorithm
na1 = 1
nb1 = 0
```

```
SSE1, est_params1, cov1, e_var1 = step_3(max_iterations, mu_max, na1, nb1, train,
mu, delta)
print('Estimated parameters : ', est_params1)
print('Estimated Covariance matrix : ', cov1)
print('Estimated variance of error : ', e_var1)

# SSE Plot
SSEplot(SSE1)
confidence_interval(cov1, na1, nb1, est_params1)
zeros1, poles1 = zeros_and_poles(est_params1, na1, nb1)

# 1-step ahead prediction
y_hat_t_11 = []
for i in range(len(train)):
    y_hat_t_11.append(-est_params1[0] * train[i])

predn1 = dataframe_create_arma(y_hat_t_1,train_plot,a='Close')
plt.figure()
plt.plot(train, label='Train')
plt.plot(predn1, label='1-step prediction')
plt.legend()
plt.xlabel('Years')
plt.ylabel('Value')
plt.title('Plot for ARMA(1,0) process')
plt.show()

# h-step ahead prediction
y_hat_t_h1 = []
for h in range(len(test)):
    if h == 0:
        y_hat_t_h1.append(-est_params1[0] * test[-1])
    else:
        y_hat_t_h1.append(-est_params1[0] * y_hat_t_h1[h-1])

forec1 = dataframe_create_arma(y_hat_t_h1,test_plot,a='Close')
plt.figure()
plt.plot(test, label='Test')
plt.plot(forec1, label='h-step prediction')
plt.legend()
plt.xlabel('Years')
plt.ylabel('Value')
plt.xticks(rotation='vertical')
plt.title('Plot for ARMA(1,0) process')
plt.show()

# Model statistics
pred_err1 = residual_arma(train, y_hat_t_11)
forec_err1 = forecast_err_arma(test, y_hat_t_h1)
mse_pred1, mse_forec1 = mse(pred_err1, forec_err1)
var_pred1, var_forec1 = var(pred_err1, forec_err1)
residual_acf1 = acf_values(pred_err1, lags)
Q_val2 = Q_val(train, residual_acf1)
acf_plot(residual_acf1, a='ARMA(1,0) Process')
chi_square_test(Q_val2, 20, na1, nb1)
# -------------------------------------------------------------------------------------
---------
```

```python
# ARIMA MODELS
# ARIMA(2,1,2)
arima_fit1 = sm.tsa.arima.ARIMA(train_nd['Close'],order=(2,1,2)).fit()
arima_pred = arima_fit1.fittedvalues

plt.figure()
plt.plot(train_nd['Close'], label='Train')
plt.plot(arima_pred, label='1-step prediction')
plt.xlabel('Index')
plt.ylabel('Price')
plt.legend()
plt.title('Plot for ARIMA(2,1,2)')
plt.show()

arima_forec = arima_fit1.forecast(len(test_nd['Close']))

plt.figure()
plt.plot(test_nd['Close'], label='Test')
plt.plot(arima_forec, label='h-step prediction')
plt.xlabel('Index')
plt.ylabel('Price')
plt.legend()
plt.title('Plot for ARIMA(2,1,2)')
plt.show()

arima_res = residual(train_nd['Close'], arima_pred)
arima_forec_err = forecast_err(test_nd['Close'], arima_forec)
mse_pred_arima, mse_forec_arima = mse(arima_res, arima_forec_err)
var_pred_arima, var_forec_arima = var(arima_res, arima_forec_err)
residual_acf_arima = acf_values(arima_res, lags)
Q_val_arima = Q_val(train_nd['Close'], residual_acf_arima)
acf_plot(residual_acf_arima, a='ARIMA(2,1,2)')

# ARIMA(1,1,0)
arima_fit2 = sm.tsa.arima.ARIMA(train_nd['Close'],order=(1,1,0)).fit()
arima_pred1 = arima_fit2.fittedvalues

plt.figure()
plt.plot(train_nd['Close'], label='Train')
plt.plot(arima_pred1, label='1-step prediction')
plt.xlabel('Index')
plt.ylabel('Price')
plt.legend()
plt.title('Plot for ARIMA(1,1,0)')
plt.show()

arima_forec1 = arima_fit2.forecast(len(test_nd['Close']))

plt.figure()
plt.plot(test_nd['Close'], label='Test')
plt.plot(arima_forec1, label='h-step prediction')
plt.xlabel('Index')
plt.ylabel('Price')
plt.legend()
plt.title('Plot for ARIMA(1,1,0)')
plt.show()
```

```
arima_res1 = residual(train_nd['Close'], arima_pred1)
arima_forec_err1 = forecast_err(test_nd['Close'], arima_forec1)
mse_pred_arima1, mse_forec_arima1 = mse(arima_res1, arima_forec_err1)
var_pred_arima1, var_forec_arima1 = var(arima_res1, arima_forec_err1)
residual_acf_arima1 = acf_values(arima_res1, lags)
Q_val_arima1 = Q_val(train_nd['Close'], residual_acf_arima1)
acf_plot(residual_acf_arima1, a='ARIMA(1,1,0)')
# ---------------------------------------------------------------------------------
---------

# FINAL MODEL SELECTION
print('STATISTICS')
pd.set_option('display.max_rows', None,'display.max_columns', None)
stat_table = pd.DataFrame({'MSE Prediction':
[mse_p_avg,mse_p_nai,mse_p_dri,mse_p_ses,mse_p_hs,mse_p_hl,mse_p_hl_d,

mse_pred_lr,mse_pred1_lr,mse_pred,mse_pred1,mse_pred_arima,mse_pred_arima1],
                           'MSE Forecast':
[mse_f_avg,mse_f_nai,mse_f_dri,mse_f_ses,mse_f_hs,mse_f_hl,mse_f_hl_d,

'N/A','N/A',mse_forec,mse_forec1,mse_forec_arima,mse_forec_arima1],
                           'Q Values':
[Q_avg,Q_nai,Q_dri,Q_ses,Q_hs,Q_hl,Q_hl_d,Q_back,Q_forw,Q_val1,Q_val2,Q_val_arima,
                                        Q_val_arima1],
                           'Variance Residuals':
[var_p_avg,var_p_nai,var_p_dri,var_p_ses,var_p_hs,var_p_hl,var_p_hl_d,

pred_var_lr,pred_var1,var_pred,var_pred1,var_pred_arima,var_pred_arima1],
                           'Variance Forecast Error':
[var_f_avg,var_f_nai,var_f_dri,var_f_ses,var_f_hs,var_f_hl,

var_f_hl_d,'N/A','N/A',var_forec,var_forec1,var_forec_arima,var_forec_arima1]},
                          index=["Average","Naive","Drift","SES","Holt's
Seasonal","Holt's Linear (Diff)","Holt's Linear (Non-Diff)",
                                 "Backward LR","Forward
LR","ARMA(2,2)","ARMA(1,0)","ARIMA(2,1,2)","ARIMA(1,1,0)"])
print(stat_table)
# ---------------------------------------------------------------------------------
---------
```

FinalProjectFunctions.py

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import statsmodels.tsa.holtwinters as ets
import statsmodels.api as sm
from statsmodels.tsa.stattools import adfuller
import seaborn as sns
from scipy.stats import chi2
from scipy import signal


# FUNCTION FOR ADF TEST
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))
    return None


# AUTOCORRELATION FUNCTION
def cal_acf_df(x,tau):
    x_bar = np.mean(x)
    den = 0
    num = 0
    for ele in range(len(x)):
        d = (x.iloc[ele] - x_bar)**2
        den += d
    for ele in range(tau,len(x)):
        n = (x.iloc[ele]-x_bar)*(x.iloc[ele-tau]-x_bar)
        num += n
    tau_n = num/den
    return tau_n


def acf_values_df(x, lag):
    tau_list = []
    for tau in range(len(lag)):
        tau_list.append(cal_acf_df(x,tau))
    # tau_list.pop(0)
    new_tau = []
    # print(tau_list)
    for each in tau_list:
        new_val = each**2
        new_tau.append(new_val)
    # print('The values for tau are as follows:')
    return new_tau


# ACF OF RESIDUALS FOR PLOTTING
def acf_plot(tau_list, a=''):
    sym_tau = tau_list[::-1]
    sym_tau.pop(-1)
    sym_tau.extend(tau_list)
    # print(sym_tau)
    plt.figure()
    plt.stem(sym_tau)
    plt.title('Auto-correlation Function of {}'.format(a))
    plt.xlabel('Lags')
    plt.ylabel('Magnitude')
    return plt.show()


# FUNCTION TO CREATE DATAFRAME
def dataframe_create(prediction, test_set, a='', b=''):
    forecast1 = pd.DataFrame({a: test_set[a], b: prediction})
    return forecast1
```

```
# AVERAGE METHOD
def avg_method(train_s, test_s):
    T = len(train_s)
    test_len = len(test_s)
    # x = train_s.mean()
    # print(x)
    prediction = []
    sum = 0
    # prediction.append(train_s.cumsum())
    for i in range(T):
        sum += train_s.iloc[i]
        prediction.append(sum)
    last_val = prediction[-1]
    avg = last_val/T
    forecast = []
    for j in range(test_len):
        forecast.append(avg)
    return prediction, forecast


# RESIDUAL CALCULATION FOR AVERAGE METHOD
def residual_avg(train_s,prediction):
    predn = []
    for i in range(len(prediction)):
        predn.append(prediction[i]/(i+1))
    # print(predn)
    residual = []
    for i in range(len(train_s)):
        value = train_s.iloc[i] - predn[i-1]
        residual.append(value)
    residual.pop(0)
    return residual


# NAIVE METHOD
def naive_method(train_s, test_s):
    T = len(train_s)
    test_len = len(test_s)
    prediction = []
    for i in range(T):
        prediction.append(train_s.iloc[i - 1])
    # prediction.pop(0)
    # the 0th value is taken as the last obs in train set if pop is not done
    # for calculations, we must drop this value
    forecast = []
    for j in range(test_len):
        forecast.append(train_s.iloc[-1])
    return prediction, forecast


# DRIFT METHOD
def drift_method(train_s,test_s):
    T = len(train_s)
    prediction = []
    for i in range(1,T+1):
        if i == 1:
            prediction.append(train_s.iloc[0])
```

```python
        else:
            num = (train_s.iloc[i-1] - train_s.iloc[0])/(i-1)
            num1 = train_s.iloc[i-1] + num
            prediction.append(num1)
    forecast = []
    test_len = len(test_s)
    y_T = train_s.iloc[-1]
    y_1 = train_s.iloc[0]
    # print(y_T,y_1)
    for h in range(test_len):
        h = h + 1
        num2 = y_T + (h * ((y_T-y_1)/(T - 1)))
        forecast.append(num2)
    return prediction, forecast


# SES METHOD
def ses_method(train_s, test_s, alpha):
    T = len(train_s)
    prediction = []
    for i in range(1,T+1):
        if i == 1:
            prediction.append(train_s.iloc[0])
        else:
            val1 = alpha*train_s.iloc[i-1]
            val2 = (1-alpha)*(prediction[-1])
            num = val1 + val2
            prediction.append(num)
    test_len = len(test_s)
    forecast = []
    for i in range(test_len):
        forecast.append(prediction[-1])
    return prediction, forecast


# RESIDUAL
def residual(train_s, prediction):
    residual = []
    T = len(train_s)
    for i in range(T):
        error = train_s.iloc[i]-prediction[i]
        residual.append(error)
    residual.pop(0)
    return residual


# FORECAST ERROR
def forecast_err(test_s, forec):
    f_error = []
    T = len(test_s)
    for i in range(T):
        error = test_s.iloc[i]-forec.iloc[i]
        f_error.append(error)
    return f_error


# MSE CALCULATIONS
```

```python
def mse(pred_err, forec_err):
    mse_f = []
    mse_p = []
    for i in pred_err:
        mse_p.append(i**2)
    for j in forec_err:
        mse_f.append(j**2)
    return np.mean(mse_p), np.mean(mse_f)


# VARIANCE CALCULATION
def var(pred_err, forec_err):
    return np.var(pred_err), np.var(forec_err)


# AUTOCORRELATION FUNCTION
# Calculates tau value
def cal_acf(x,tau):
    x_bar = np.mean(x)
    den = 0
    num = 0
    for ele in range(len(x)):
        d = (x[ele] - x_bar)**2
        den += d
    for ele in range(tau,len(x)):
        n = (x[ele]-x_bar)*(x[ele-tau]-x_bar)
        num += n
    tau_n = num/den
    return tau_n


# Calculates the acf values
def acf_values(x, lag):
    tau_list = []
    for tau in range(len(lag)):
        tau_list.append(cal_acf(x,tau))
    # tau_list.pop(0)
    new_tau = []
    # print(tau_list)
    for each in tau_list:
        new_val = each**2
        new_tau.append(new_val)
    # print('The values for tau are as follows:')
    return new_tau


# Q VALUES
def Q_val(train_s, rk_vals):
    T = len(train_s)
    value = 0
    for val in rk_vals[1:]:
        value += (val**2)
    # print(value)
    Q = T*value
    return Q
```

```
# CORRELATION COEFFICIENT for dataset
def correlation_coefficient_cal_data(x,y):
    x_bar = np.mean(x)
    y_bar = y.mean()
    num = float(np.sum((x-x_bar)*(y-y_bar)))
    den1 = float(np.sqrt(np.sum((x-x_bar)**2)))
    den2 = float(np.sqrt(np.sum((y-y_bar)**2)))
    den = np.floor(den1*den2)
    r = num/den
    # print(num, den)
    return r


# CORRELATION COEFFICIENT for numpy array
def correlation_coefficient_cal1(x,y):
    a = np.matmul(x-np.mean(x),y-np.mean(y))
    b = np.sqrt(np.sum((x-np.mean(x))**2))
    c = np.sqrt(np.sum((y-np.mean(y))**2))
    corr_coeff = a / (b*c)
    return round(corr_coeff,2)


# BASIC PLOT FUNCTION
def plot_func(train_var1, test_var1, forecast_var1, train_var2, test_var2,
forecast_var2, a=''):
    plt.figure()
    plt.plot(train_var1,train_var2, label='Training set')
    plt.plot(test_var1,test_var2, label='Testing set')
    plt.plot(forecast_var1, forecast_var2, label='h-step Forecast')
    plt.legend()
    plt.title('Plot for {} Method'.format(a))
    plt.ylabel('Data')
    plt.xlabel('Year')
    return plt.show()


# BASIC PLOT FUNCTION FOR SES
def plot_func_ses(train_var1, test_var1, forecast_var1, train_var2, test_var2,
forecast_var2, alpha, a=''):
    plt.figure()
    plt.plot(train_var1,train_var2, label='Training set')
    plt.plot(test_var1,test_var2, label='Testing set')
    plt.plot(forecast_var1, forecast_var2, label='h-step Forecast')
    plt.legend()
    plt.title('Plot for {} Method when Alpha is {}'.format(a,alpha))
    plt.ylabel('Data')
    plt.xlabel('Year')
    return plt.show()


# PLOTTING DIFFERENT VALUES OF ALPHA - SES
def ses_plots(train_s, test_s, alpha, a='', b='',c=''):
    ses_pred, ses_forec = ses_method(train_s[a],test_s[a],alpha)
    final_predn = dataframe_create(ses_pred, train_s, b, a)
    final_forec = dataframe_create(ses_forec, test_s, b, a)
    return
plot_func_ses(train_s[b],test_s[b],final_forec[b],train_s[a],test_s[a],final_forec[
```

```
a],alpha,c)


# FUNCTION CALL FOR SES METHOD
def call_ses(train_s, test_s, alpha,lag, a='', b='', c=''):
    predn, forec = ses_method(train_s[a], test_s[a], alpha)
    final_predn = dataframe_create(predn, train_s, b, a)
    final_forec = dataframe_create(forec, test_s, b, a)

plot_func_ses(train_s[b],test_s[b],final_forec[b],train_s[a],test_s[a],final_forec[
a],alpha,c)
    residuals = residual(train_s[a], final_predn[a])
    forec_error = forecast_err(test_s[a], final_forec[a])
    mse_p, mse_f = mse(residuals, forec_error) # add another list in the function
when needed
    var_p, var_f = var(residuals, forec_error)
    residual_acf = acf_values(residuals, lag)
    Q = Q_val(train_s, residual_acf)
    acf_plot(residual_acf, c)
    return final_predn, final_forec, residuals, forec_error, mse_p, mse_f, var_p,
var_f, residual_acf, Q


# FUNCTION CALL FOR AVERAGE METHOD
def call_avg(train_s, test_s,lag, a='', b='', c=''):
    predn, forec = avg_method(train_s[a], test_s[a])
    final_predn = dataframe_create(predn, train_s, b, a)
    final_forec = dataframe_create(forec, test_s, b, a)

plot_func(train_s[b],test_s[b],final_forec[b],train_s[a],test_s[a],final_forec[a],c
)
    residuals = residual_avg(train_s[a], final_predn[a])
    forec_error = forecast_err(test_s[a], final_forec[a])
    mse_p, mse_f = mse(residuals, forec_error)
    var_p, var_f = var(residuals, forec_error)
    residual_acf = acf_values(residuals, lag)
    Q = Q_val(train_s, residual_acf)
    acf_plot(residual_acf, c)
    return final_predn, final_forec, residuals, forec_error, mse_p, mse_f, var_p,
var_f, residual_acf, Q


# FUNCTION CALL FOR NAIVE METHOD
def call_naive(train_s, test_s, lag, a='', b='', c=''):
    predn, forec = naive_method(train_s[a], test_s[a])
    final_predn = dataframe_create(predn, train_s, b, a)
    final_forec = dataframe_create(forec, test_s, b, a)

plot_func(train_s[b],test_s[b],final_forec[b],train_s[a],test_s[a],final_forec[a],c
)
    residuals = residual(train_s[a], final_predn[a])
    forec_error = forecast_err(test_s[a], final_forec[a])
    mse_p, mse_f = mse(residuals, forec_error)
    var_p, var_f = var(residuals, forec_error)
    residual_acf = acf_values(residuals, lag)
    Q = Q_val(train_s, residual_acf)
    acf_plot(residual_acf, c)
```

```
    return final_predn, final_forec, residuals, forec_error, mse_p, mse_f, var_p,
var_f, residual_acf, Q


# FUNCTION CALL FOR DRIFT METHOD
def call_drift(train_s, test_s, lag, a='', b='', c=''):
    predn, forec = drift_method(train_s[a], test_s[a])
    final_predn = dataframe_create(predn, train_s, b, a)
    final_forec = dataframe_create(forec, test_s, b, a)

plot_func(train_s[b],test_s[b],final_forec[b],train_s[a],test_s[a],final_forec[a],c
)
    residuals = residual(train_s[a], final_predn[a])
    forec_error = forecast_err(test_s[a], final_forec[a])
    mse_p, mse_f = mse(residuals, forec_error)
    var_p, var_f = var(residuals, forec_error)
    residual_acf = acf_values(residuals, lag)
    Q = Q_val(train_s, residual_acf)
    acf_plot(residual_acf, c)
    return final_predn, final_forec, residuals, forec_error, mse_p, mse_f, var_p,
var_f, residual_acf, Q


# FUNCTION CALL FOR HOLT'S SEASONAL METHOD
def call_holt_s(train_s,test_s,lag, a='',b='',c=''):
    prediction = ets.ExponentialSmoothing(train_s[a], trend='additive',
seasonal='additive', seasonal_periods=1440,damped_trend=True).fit()
    # trend='multiplicative', seasonal='multiplicative', seasonal_periods=12
    # 4141, 4160, 4140
    # trend='additive', seasonal='additive', damped=True, seasonal_periods=4
    # trend='additive', seasonal='additive', damped=True, seasonal_periods=365
    forecast = prediction.forecast(steps=len(test_s[a]))
    yt = prediction.fittedvalues
    final_predn = dataframe_create(yt,train_s,b,a)
    final_forec = dataframe_create(forecast,test_s,b,a)

plot_func(train_s[b],test_s[b],final_forec[b],train_s[a],test_s[a],final_forec[a],c
)
    residuals = residual_avg(train_s[a], final_predn[a])
    forec_error = forecast_err(test_s[a], final_forec[a])
    mse_p, mse_f = mse(residuals, forec_error)
    var_p, var_f = var(residuals, forec_error)
    residual_acf = acf_values(residuals, lag)
    Q = Q_val(train_s, residual_acf)
    acf_plot(residual_acf, c)
    return prediction, final_forec, residuals, forec_error, mse_p, mse_f, var_p,
var_f, residual_acf, Q


# FUNCTION CALL FOR HOLT'S LINEAR METHOD
def call_holt_l(train_s,test_s,lag, a='',b='',c=''):
    prediction = ets.ExponentialSmoothing(train_s[a], trend='additive',
seasonal=None, damped=True).fit()
    # trend='multiplicative', seasonal=None, damped=True
    # trend='additive', seasonal=None, damped=True
    forecast = prediction.forecast(steps=len(test_s[a]))
    yt = prediction.fittedvalues
```

```
    final_predn = dataframe_create(yt,train_s,b,a)
    final_forec = dataframe_create(forecast,test_s,b,a)

plot_func(train_s[b],test_s[b],final_forec[b],train_s[a],test_s[a],final_forec[a],c
)
    residuals = residual_avg(train_s[a], final_predn[a])
    forec_error = forecast_err(test_s[a], final_forec[a])
    mse_p, mse_f = mse(residuals, forec_error)
    var_p, var_f = var(residuals, forec_error)
    residual_acf = acf_values(residuals, lag)
    Q = Q_val(train_s, residual_acf)
    acf_plot(residual_acf, c)
    return prediction, final_forec, residuals, forec_error, mse_p, mse_f, var_p,
var_f, residual_acf, Q


# FUNCTION FOR NORMAL EQUATION (LINEAR REGRESSION)
def normal_eq(y_train,x_train):
    x_transpose = np.transpose(x_train)
    e1 = np.dot(x_transpose,x_train)
    inverse = np.linalg.inv(e1)
    e2 = np.dot(x_transpose,y_train)
    b_hat = np.dot(inverse,e2)
    return b_hat


# FUNCTION TO CALCULATE VARIANCE FOR LINEAR REGRESSION
def var_linreg(error_arr, num_features):
    T = len(error_arr)
    k = num_features
    e1 = 1/(T-k-1)
    val = np.sum(np.square(error_arr))
    sigma_e = np.sqrt(e1*val)
    return sigma_e


# FUNCTION FOR BACKWARD STEPWISE REGRESSION
def backward_reg(x_train, y_train, drop_feature=''):
    x_train = x_train.drop(columns=drop_feature)
    X = sm.add_constant(x_train)
    model = sm.OLS(y_train, X).fit()
    return model, x_train


# FUNCTION FOR FORWARD STEPWISE REGRESSION
def forward_reg(y_train,x_train, x_trainf,feature,name=''):
    x_trainf[name] = x_train[feature]
    X = sm.add_constant(x_trainf)
    model = sm.OLS(y_train, X).fit()
    return model, x_trainf


# FUNCTION FOR MSE MULTIPLE LINEAR REGRESSION
def mse_mlr(pred_err):
    mse_p = []
    for i in pred_err:
        mse_p.append(i**2)
```

```python
    return np.mean(mse_p)


# CORRELATION COEFFICIENT MULTIPLE LINEAR REGRESSION
def correlation_coefficient_mlr(x,y):
    x_bar = np.mean(x)
    y_bar = y.mean()
    num = float(np.sum((x-x_bar)*(y-y_bar)))
    den1 = float(np.sqrt(np.sum((x-x_bar)**2)))
    den2 = float(np.sqrt(np.sum((y-y_bar)**2)))
    den = np.floor(den1*den2)
    r = num/den
    # print(num, den)
    return r

# FUNCTION TO CALCULATE STRENGTH
def strength_stats(trend, seasonal, resid):
    ft = np.maximum(0,1 -
(np.var(np.array(resid))/(np.var(np.array(trend+resid)))))
    fs = np.maximum(0,1 -
(np.var(np.array(resid))/(np.var(np.array(seasonal+resid)))))
    return ft, fs


# FUNCTION FOR PARTIAL CORRELATION
def partial_corr(ab, ac, bc):
    r_partial = (ab-ac*bc)/(np.sqrt(1-ac**2)*np.sqrt(1-bc**2))
    return r_partial


# FUNCTION TO CALCULATE t0
def t_test_pc(r_ab_c, n, k):
    t0 = r_ab_c*np.sqrt((n-2-k)/(1-r_ab_c**2))
    return t0


# FUNCTION TO CALCULATE PHI - GPAC
def phical(acfcal,na,nb):
    den = []
    k = na
    j = nb

    for a in range(k):
        den.append([])
        for b in range(k):
            den[a].append(acfcal[np.abs(j + b)])
        j = j - 1

    j = nb
    num = den[:k - 1]
    num.append([])
    for a in range(k):
        num[k - 1].append(acfcal[j + a + 1])

    det_num = round(np.linalg.det(num),5)
    det_den = round(np.linalg.det(den),5)
```

```
    if det_den == 0:
        return float('inf')

    phi = det_num / det_den

    return round(phi,3)


# FUCNCTION TO CALCULATE GPAC AND PLOT SNS PLOT
def gpac_cal(acfcal,k,j):
    phi = []
    for b in range(j):
        phi.append([])
        for a in range(1, k+1):
            phi[b].append(phical(acfcal, a, b))

    gpac = np.array(phi).reshape(j, k)
    gpac_df = pd.DataFrame(gpac)
    cols = np.arange(1, k+1)
    gpac_df.columns = cols
    print(gpac_df)
    print()

    sns.heatmap(gpac_df, annot=True)
    plt.xlabel('AR process(k)')
    plt.ylabel('MA process(j)')
    plt.title('Generalized Partial Autocorrelation (GPAC) table')
    plt.show()


# FUNCTION FOR CONFIDENCE INTERVAL
def confidence_interval(cov, na, nb, params):
    print('\n Confidence Interval for Estimated parameters')
    for i in range(na):
        upper = params[i] + 2 * np.sqrt(cov[i][i])
        lower = params[i] - 2 * np.sqrt(cov[i][i])
        print(lower, '< a{} <'.format(i+1), upper)

    for j in range(nb):
        upper = params[na+j] + 2 * np.sqrt(cov[na+j][na+j])
        lower = params[na+j] - 2 * np.sqrt(cov[na+j][na+j])
        print(lower, '< b{} <'.format(j + 1), upper)


# FUNCTION FOR ZEROS AND POLES
def zeros_and_poles(est_params, na, nb):
    p_ceoff = [1] + list(est_params[:na])
    z_coeff = [1] + list(est_params[na:])
    poles = np.roots(p_ceoff)
    zeros = np.roots(z_coeff)
    print('\nZeros : ',zeros)
    print('Poles : ',poles)
    return zeros, poles


# Chi-square test
def chi_square_test(Q, lags, na, nb, alpha=0.01):
```

```
    dof = lags - na - nb
    chi_critical = chi2.isf(alpha, df=dof)

    if Q < chi_critical:
        print(f"The residual is white and the estimated order is n_a = {na} and n_b
= {nb}")
    else:
        print(f"The residual is not white with n_a={na} and n_b={nb}")

    return Q < chi_critical


# FUNCTION TO CREATE DATAFRAME
def dataframe_create_arma(prediction, test_set, a=''):
    forecast1 = pd.DataFrame({a: prediction}, index=test_set.index)
    return forecast1

# RESIDUAL ARMA
def residual_arma(train_s, prediction):
    residual = []
    T = len(train_s)
    for i in range(T):
        error = train_s.iloc[i]-prediction[i]
        residual.append(error)
    residual.pop(0)
    return residual


# FORECAST ERROR ARMA
def forecast_err_arma(test_s, forec):
    f_error = []
    T = len(test_s)
    for i in range(T):
        error = test_s.iloc[i]-forec[i]
        f_error.append(error)
    return f_error
```

References

1. Class Lectures and Videos
2. Codes developed in class
3. https://robjhyndman.com/hyndsight/seasonal-periods/
4. https://www.kaggle.com/yashvi/time-series-analysis-and-forecasting-reliance
5. https://otexts.com/fpp2/tspatterns.html
6. https://machinelearningmastery.com/decompose-time-series-data-trend-seasonality/
7. https://machinelearningmastery.com/arima-for-time-series-forecasting-with-python/