# Time Series analysis for the S&P 500

## Introduction and Project Outline

The general purpose of this part of the project is to analyse two different time series models on a dataset containing the monthly price of the S&P 500 stock price for a span of over 3yrs. Each of the models will attempt to make forecasts or predictions of the price of S&P 500 stock price either for the months in a testing set or future months. The S&P 500 stock will briefly be described below so we can develop an understanding of the possible impact each model would have in real-world use.

The S&P 500 (Standard and Poor's 500) is a stock market index tracking the performance of 500 leading U.S publicly traded companies, with a primary emphasis on market capitalisation. A stock index can be simply put as a collection of stocks that are tracked together to give people an overall idea of how the stock market is doing. It is also known as a float-weighted index, meaning the market capitalisations of the companies in the index are adjusted by the number of shares available for public trading. Investing with the S&P 500 is a diversified and a relatively low-risk way to invest into the stock market.

Building and evaluating each of the Time series models for this project will be developed with Python and its various libraries. The two libraries which are most applicable for our purpose are '*statsmodels*' and '*pmdarima*'.

Once both Time series models are fully developed, they will be compared to determine which model is optimal for the time series problem.

## Additional Software Used

### *statsmodels*

‘*statsmodels*’ is a Python package that provides a complement to SciPy for statistical computations including descriptive statistics and estimation and inference for statistical models.

The package was released under the open-source Modified BSD (3 – clause) licence so it’s free to use for both academic and commercial use. It was developed on top of NumPy, SciPy and matplotlib by Skipper Seabold and Josef Perktold.

Most methods required for building time series models used in this project will be implemented and evaluated through the ‘*statsmodels.tsa*’ API within the ‘*statsmodels*’ library. ‘*statsmodels.tsa*’ contains model classes and functions that are useful for time series analysis, such as a non-seasonal ARIMA model used for analysing the S&P 500 dataset.

### *pmdarima*

‘*pmdarima*’ is a statistical library designed to fill a void in Python's time series analysis capabilities which includes an equivalent of R's *auto.arima* function with the *auto_arima* function. It wraps statsmodels under the hood but is designed with an interface that's familiar to users coming from a scikit-learn background. The project was developed by Taylor G Smith as a result of a long-standing debate between colleagues.

The *auto_arima* function will be used in this project to build the ARIMA component of the SAIRMA model to identify the most optimal p, d, q parameters and return a fitted ARIMA model.

## Background Theory

Whenever any data or observations are recorded at regular time intervals, it's referred to as Time Series data. A Time Series model attempts to observe this data over a time period to forecast or predict what will happen in the future. It is important to note that Time Series data are often based on patterns or recurring trends from previous time periods due to the fact the time in which data is collected has an influence on future outcomes. This is often referred to as the Time Series data is non-stationary.

Time Series Analysis consists of using a statistical model to predict future values of a time series, such as in Finance when predicting stock price, Healthcare professional when analysing Covid-19 positive cases/death and Meteorology for weather prediction.

Time series analysis involves working with data comprising of the following components:

- **Trend:** Data growing in a general direction over a period of time. This can involve trends increasing (upward), decreasing (downward) or horizontal (stationary).
- **Seasonality:** Data exhibits predictable, repeating patterns. It is critical to control for seasonality because it could impact the accuracy of the results.
- **Cyclical Component:** A trend that has no set repetition over a certain time period. A cycle can be a period of ups and downs, mostly seen in business cycles – cycles do not exhibit a seasonality trend
- **Irregular Variation:** Fluctuations in time-series data that is erratic, unpredictable and may/may not be random.

The main task of the project is to analyse two different time series models to predict or forecast the Stock Price of the S&P 500 index fund. Once both time series models are fully functional, each model is compared with each other to find the most optimal solution.

The two statistical models used in this project are as follows:

- Model 1: ARIMA model
- Model 2: SARIMA model

**Model 1: ARIMA model**

ARIMA (AutoRegressive Integrated Moving Average) model is a non-seasonal forecasting algorithm based on the idea that the information on the past values of the time series can alone be used to predict the future values. The model in a sense uses its own lags and the lagged forecast errors to forecast future values.

**Note:** An ARIMA model can be described as combining differencing with autoregression and a moving average model.

The ARIMA model can be isolated by the following components:

- **Auto Regressive (AR)** model is a specific type of regression model where the past values and its lags can impact current and future values.
- **Integrated (I)** aspect of the model refers to when it uses differencing to reduce seasonality from a time series. ARIMA models have a degree of differencing which makes the time series stationary.
- **Moving Average (MA)** model works by analysing how wrong the predicted values were for the previous time-periods to make a better estimate for the current time-period.

The equation for the ARIMA model:

$$Y_t = c + \beta_1 Y_{t-1} + \beta_2 Y_{t-2} + \ldots + \beta_p Y_{t-p} + \theta_1 \epsilon_{t-1} + \ldots + \theta_q \epsilon_{t-q} + \epsilon_t$$

where:

- *c* is the intercept
- $\beta_1 Y_{t-1} + \beta_2 Y_{t-2} + \ldots + \beta_p Y_{t-p}$ are the lags (AR)
- $\theta_1 \epsilon_{t-1} + \ldots + \theta_q \epsilon_{t-q} + \epsilon_t$ are the errors (MA)

In English, the formula can be described as:

$$\begin{gathered} Predicted\, Y_t = Constant + Linear\ Combination\ Lags\ of\ Y\ upto\ p\ lags \\ + Linear\ Combination\ of\ Lagged\ forecast\ error\ upto\ q\ lags \end{gathered}$$

An ARIMA model is characterised by 3 terms ($p$, $d$, $q$):

- $p$ is the order of the AR term. It refers to the number of lags.
- $d$ is the number of differencing required to make the time series stationary referring to the I term.
- $q$ is the order of the MA term. It refers to the number of lagged forecast errors that should go into the ARIMA model.

To fit the ARIMA model, the time series needs to be stationary or as close to stationarity as possible. Stationarity basically means that seasonality doesn't exist. Seasonality occurs when the time series is predictable or has a representing pattern. It is critical for this reason to control for seasonality because it could impact the accuracy of the results.

The easiest way to remove seasonality from a time series is using differencing. The goal of differencing in this context is to stabilise the mean of a time series by removing changes in the level of a time series, and therefore eliminating (or reducing) trend and seasonality. This essentially transforms the Time Series to stationary as the data will have no predictable pattern in the long-term.

Some of the topics we meet when implemented a ARIMA model:

- Augmented Dickey–Fuller Test (ADF Test)
- Autocorrelation plot (ACF Plot)
- Partial Autocorrelation Plot (PACF Plot)

**ADF Test: Augmented Dickey Fuller Test**

ADF (Augmented Dickey-Fuller) test is a statistical significance test which means the test will check if the Time Series is stationary or not.

It acts as a Hypothesis test for the Time Series when implementing an ARIMA model.

- $H_0$: Time Series is non-stationary, therefore ARIMA model cannot be implemented.
- $H_A$: Time Series is stationary, therefore the ARIMA model can be implemented.

The statsmodels Python library allows easy execution of the ADF test on any time series data with the *adfuller()* function. The function produces the time series p-values which can be used to make inferences about the time series, whether it is stationary or not.

The p–value is a proxy for the influence of trends on time series data which can be inferred as follows:

- **p–value > 0.05:** Fail to reject the null hypothesis ($H_0$), therefore the time series is non-stationary so the ARIMA model cannot be implemented.
- **p–value <= 0.05:** Reject the null hypothesis ($H_0$), therefore the time series is stationary so the ARIMA model can be implemented.

Below the ACF and PACF plots will be explained which are crucial when choosing optimal *q* and *p* values.

### ACF: Autocorrelation Function Plot

The Autocorrelation function (ACF) is a statistical technique used to identify how correlated the values in a time series are with each other.

The ACF plots the correlation coefficient against the lag, which is measured in terms of a number of periods or units. A lag corresponds to a certain point in time after which we observe the first value in the time series.

### PACF: Partial Autocorrelation Plot

Partial autocorrelation is a statistical measure that captures the correlation between two variables after controlling for the effects of other variables.

### Model 2: SARIMA - Seasonal ARIMA

Seasonal Autoregressive Integrated Moving Average (SARIMA) model builds upon the ARIMA model. It is formed by including additional seasonal terms ($P$, $D$, $Q$, $m$) to the ARIMA model. It can be denoted as follows:

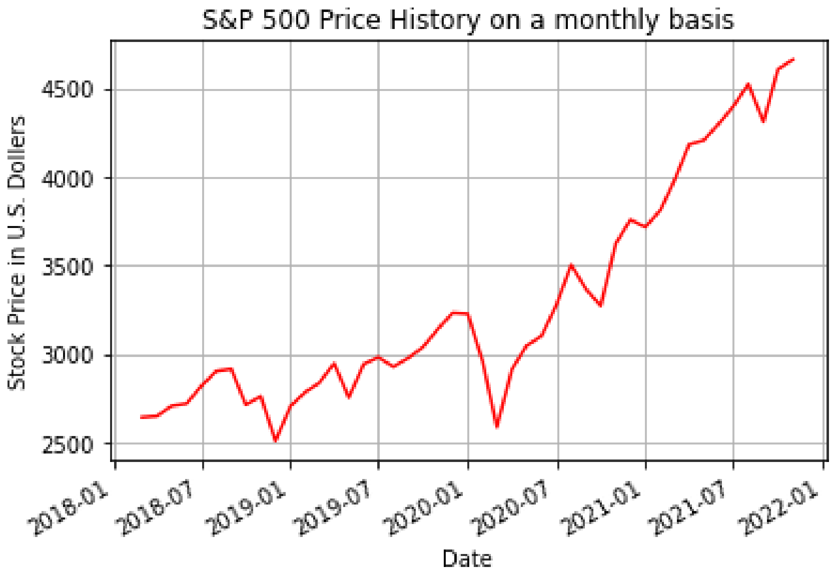
$$\text{ARIMA} \quad \underbrace{(p, d, q)}_{\substack{\uparrow \\ \text{Non-seasonal part} \\ \text{of the model}}} \quad \underbrace{(P, D, Q)_m}_{\substack{\uparrow \\ \text{Seasonal part of} \\ \text{of the model}}}$$

A SARIMA model is characterised by 4 additional terms ( $P$, $D$, $Q$, $m$):

- $P$ is the order of the seasonal AR term.
- $D$ is the number of seasonal differencing referring to the I term.
- $Q$ is the order of the seasonal MA term.
- $m$ is the number of time steps for a single seasonal period

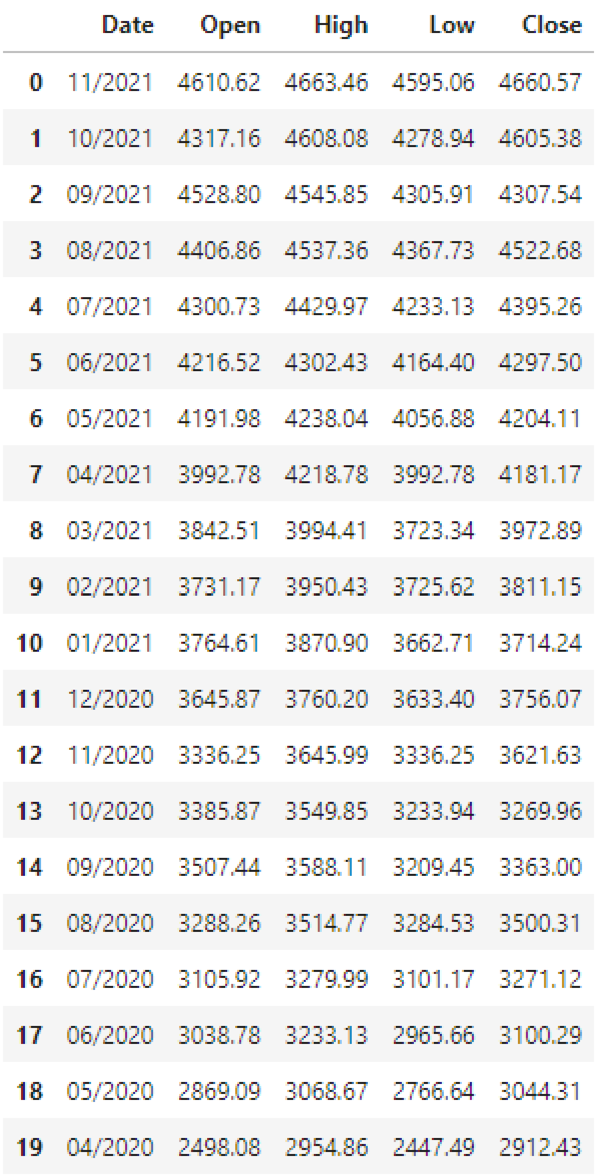
## Dataset

The dataset used in this project represents the monthly price of the S&P 500 index fund, from '03/2018' to '11/2021' so the dataset contains 45 data points in total spanning over 3 years. The index fund tracks the performance of the top 500 performing companies listed on stock exchanges in the United States, which gives a general indication of the performance of the entire market. We renamed the dataset to '*SaP_500_3yrs_Monthly.csv*' to improve the readability of the project.



The dataset was sourced from '*marketwatch.com*', which is a website that provides the latest stock prices updated daily. This website makes it possible to download all the S&P 500 Stock Price information at a selected date range and a frequency as a '*csv*' file.

The dataset created from '*marketwatch.com*' is clean, so there is very little editing of the data required.



|  | Date | Open | High | Low | Close |
|---|---|---|---|---|---|
| 0 | 11/2021 | 4610.62 | 4663.46 | 4595.06 | 4660.57 |
| 1 | 10/2021 | 4317.16 | 4608.08 | 4278.94 | 4605.38 |
| 2 | 09/2021 | 4528.80 | 4545.85 | 4305.91 | 4307.54 |
| 3 | 08/2021 | 4406.86 | 4537.36 | 4367.73 | 4522.68 |
| 4 | 07/2021 | 4300.73 | 4429.97 | 4233.13 | 4395.26 |
| 5 | 06/2021 | 4216.52 | 4302.43 | 4164.40 | 4297.50 |
| 6 | 05/2021 | 4191.98 | 4238.04 | 4056.88 | 4204.11 |
| 7 | 04/2021 | 3992.78 | 4218.78 | 3992.78 | 4181.17 |
| 8 | 03/2021 | 3842.51 | 3994.41 | 3723.34 | 3972.89 |
| 9 | 02/2021 | 3731.17 | 3950.43 | 3725.62 | 3811.15 |
| 10 | 01/2021 | 3764.61 | 3870.90 | 3662.71 | 3714.24 |
| 11 | 12/2020 | 3645.87 | 3760.20 | 3633.40 | 3756.07 |
| 12 | 11/2020 | 3336.25 | 3645.99 | 3336.25 | 3621.63 |
| 13 | 10/2020 | 3385.87 | 3549.85 | 3233.94 | 3269.96 |
| 14 | 09/2020 | 3507.44 | 3588.11 | 3209.45 | 3363.00 |
| 15 | 08/2020 | 3288.26 | 3514.77 | 3284.53 | 3500.31 |
| 16 | 07/2020 | 3105.92 | 3279.99 | 3101.17 | 3271.12 |
| 17 | 06/2020 | 3038.78 | 3233.13 | 2965.66 | 3100.29 |
| 18 | 05/2020 | 2869.09 | 3068.67 | 2766.64 | 3044.31 |
| 19 | 04/2020 | 2498.08 | 2954.86 | 2447.49 | 2912.43 |

The dataset contains the following variables:

- **'*Date*':** Date associated with the S&P 500 prices
- **'*Open*':** Opening Price of the S&P 500 on the respective date
- **'*High*':** Highest Price of the S&P 500 on the respective date
- **'*Low*':** Lowest Price of the S&P 500 on the respective date
- **'*Close*':** Price the S&P 500 finished on the respective date (in U.S. Dollars)

The ‘*Open*’, ‘*High*’ and ‘*Low*’ variables will be removed from the dataset as they are not useful when applying Time Series models.

The aim of the Time Series models in this context will be used to forecast or predict the price of the S&P 500 Stock Price by using the ‘*Date*’ and ‘*Close*’ variables.

Each Time Series model used in this project will be briefly described below:

- ARIMA model uses the ‘*Close*’ variable to forecast the future values of the stock price.
- SARIMA model uses both the ‘*Date*’ and ‘*Close*’ variables to predict the future values of the stock price.

## Implementing the Time Series models

The overall goal of the project is to analyse the stock price of the S&P 500 index fund with two relevant Time Series models. This analysis will primarily focus on predicting or forecasting the potential future stock price of the index fund whilst paying particular attention to the stock market trends with respect to the index fund. The results from such models are extremely useful for investors and hedge funds as it gives them crucial information about the prospects of the index fund. The investors in turn are equipped with the knowledge to make better decisions when investing into the index fund.

The main two Time Series models we will investigate to accomplish this goal are as follows:

- ARIMA model
- SARIMA model

The software required to analyse the S&P 500 stock price with each model will be completed with Python and its data analytics libraries. A brief description for the main libraries and its relevance will be given below:

- Pandas - Loads and manipulates the dataset.
- matplotlib - Displays all the necessary plots in the project.
- statsmodels - Contains necessary methods to build each Time Series model.
- pmdarima - Contains a function to automatically fit the ARIMA model to the dataset.

The general idea of the project is to compare each model once fully built and functional to assess each model's merits.

To begin the project, the dataset obtained for this project will need to be manipulated to comply with the format required for each respective model. The manipulations prepare the data for the Time Series models. The main steps taken to prepare the data are as follows:

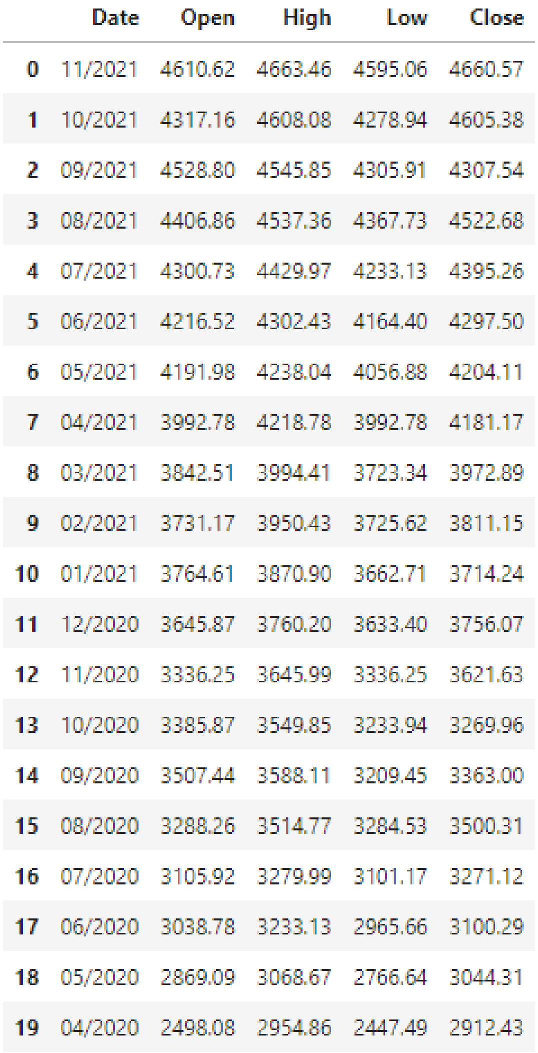
1. Loading the dataset
2. Visualising the dataset
3. Preparing the data for the model

### Loading the dataset

Firstly, we will load the '*SaP_500_3yrs_Monthly.csv*' file as a DataFrame ('*SP_500*'), which contains our entire dataset.

Then we will print the first 20 entries in our dataset.

```
# Importing the pandas library
import pandas as pd

# Loads the 'SaP_500_3yrs_Monthly.csv' which contains our dataset
SP_500 = pd.read_csv('SaP_500_3yrs_Monthly.csv')

# Prints the first 20 entries from our dataset
SP_500.head(20)
```



| | Date | Open | High | Low | Close |
|---|---|---|---|---|---|
| **0** | 11/2021 | 4610.62 | 4663.46 | 4595.06 | 4660.57 |
| **1** | 10/2021 | 4317.16 | 4608.08 | 4278.94 | 4605.38 |
| **2** | 09/2021 | 4528.80 | 4545.85 | 4305.91 | 4307.54 |
| **3** | 08/2021 | 4406.86 | 4537.36 | 4367.73 | 4522.68 |
| **4** | 07/2021 | 4300.73 | 4429.97 | 4233.13 | 4395.26 |
| **5** | 06/2021 | 4216.52 | 4302.43 | 4164.40 | 4297.50 |
| **6** | 05/2021 | 4191.98 | 4238.04 | 4056.88 | 4204.11 |
| **7** | 04/2021 | 3992.78 | 4218.78 | 3992.78 | 4181.17 |
| **8** | 03/2021 | 3842.51 | 3994.41 | 3723.34 | 3972.89 |
| **9** | 02/2021 | 3731.17 | 3950.43 | 3725.62 | 3811.15 |
| **10** | 01/2021 | 3764.61 | 3870.90 | 3662.71 | 3714.24 |
| **11** | 12/2020 | 3645.87 | 3760.20 | 3633.40 | 3756.07 |
| **12** | 11/2020 | 3336.25 | 3645.99 | 3336.25 | 3621.63 |
| **13** | 10/2020 | 3385.87 | 3549.85 | 3233.94 | 3269.96 |
| **14** | 09/2020 | 3507.44 | 3588.11 | 3209.45 | 3363.00 |
| **15** | 08/2020 | 3288.26 | 3514.77 | 3284.53 | 3500.31 |
| **16** | 07/2020 | 3105.92 | 3279.99 | 3101.17 | 3271.12 |
| **17** | 06/2020 | 3038.78 | 3233.13 | 2965.66 | 3100.29 |
| **18** | 05/2020 | 2869.09 | 3068.67 | 2766.64 | 3044.31 |
| **19** | 04/2020 | 2498.08 | 2954.86 | 2447.49 | 2912.43 |

As we can see the dataset begins at the most current value of the S&P price which is not ideal for the ARIMA model. Therefore, we will have to reverse the order of the data to make it usable.

Below, we will remove all the columns in the '*SP_500*' DataFrame except the '*Close*' column which is necessary for implementing the ARIMA model.

```
## Remove all the columns in 'SP_500' DataFrame except
## the 'Close' column which is useful for plotting the data
SP_500_Close = SP_500[['Close']].dropna()
```

**Visualising the dataset**

We will plot the dataset so we can plan our approach for the Time series problem.

To plot out S&P 500 dataset we need to:

- Create a new DataFrame containing only the '*Date*' and '*Close*' Variables.
- Create a DataFrame containing only the '*Close*' variable and reverse the order of the variable.
- Create a separate DataFrame containing only the '*Date*' variable and reverse the order of the variable.
- Convert the 'Date' DataFrame to datetime
- Plot the final updated variables using matplotlib

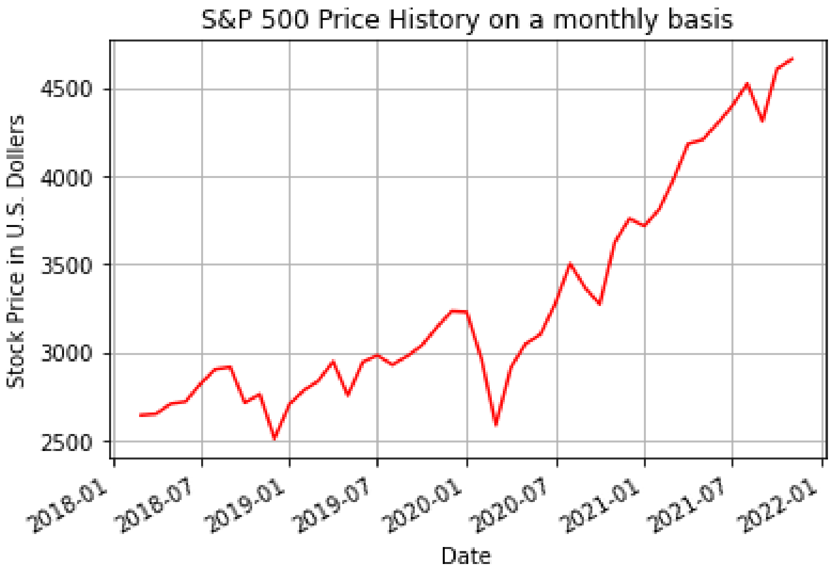
Creating a new DataFrame for plotting the S&P 500 dataset purposes only containing the '*Date*' and '*Close*' Variables as it won't affect the original dataset.

We will separate the 'Close' variable with the DataFrame '*Plotting_SP_500_Profit*' containing all the closing day prices in our dataset. The order of the 'Close' prices is reversed since our dataset is in the opposite order for our plot.

We will then separate the 'Date' variables with a unique DataFrame to convert the variable to datetime which is needed for the '*autofmt_xdate()*' function used to make the plot more readable. The order of the 'Date' will also need to be reversed to match the prices from above.

Finally, we can plot the complete dataset with the updated variables to observe the monthly price of the S&P 500 from '03/2018' to '11/2021'.

```
from matplotlib import pyplot as plt

## Creates a new Dataframe using only the 'Date' and 'Close' variables
Plotting_SP_500 = pd.DataFrame(SP_500, columns = ['Date', 'Close'])

## Creates a Variable 'Plotting_SP_500_Profit' which contains all
## the closing day prices in our dataset.
Plotting_SP_500_Profit = SP_500.iloc[:, -1].values
Plotting_SP_500_Profit = Plotting_SP_500_Profit[::-1]

## Creates a Variable 'Plotting_SP_500_Date' which contains all the
## dates associated with the closing price in the
## 'Plotting_SP_500_Profit' variable created above.
Plotting_SP_500_Date = SP_500.iloc[:, 0].values
Plotting_SP_500_Date = Plotting_SP_500_Date[::-1]

## Converts the content in the 'Plotting_SP_500_Date' variable to
## datetime. This is needed for the 'autofmt_xdate()' function
## used to make the plot of the dataset more readable.
Plotting_SP_500_Date = pd.to_datetime(Plotting_SP_500_Date)

## Creates a new DateFrame to store only the content needed to
## plot the dataset.
Plot_for_SP_500 = pd.DataFrame(Plotting_SP_500_Profit, Plotting_SP_500_Date)

## Uses 'matplotlib' library to plot the dataset
plt.title('S&P 500 Price History on a monthly basis')
plt.xlabel('Date')
plt.ylabel('Stock Price in U.S. Dollars')
plt.grid(True)
plt.plot(Plot_for_SP_500, color = 'red')
plt.gcf().autofmt_xdate()
plt.savefig('S&P_500_Price_History_monthly.png')

plt.show()
```


S&P 500 Price History on a monthly basis
Stock Price in U.S. Dollers
4500
4000
3500
3000
2500
2018-01
2018-07
2019-01
2019-07
2020-01
2020-07
2021-01
2021-07
2022-01
Date

### Model 1: ARIMA model

The first model we are going to test for this problem is an ARIMA (AutoRegressive Integrated Moving Average) model, which is a forecasting algorithm based on the idea that the information in the past values of the time series can alone be used to predict the future values. The model in a sense uses its own lags and the lagged forecast errors to forecast future values.

An ARIMA model is characterised by 3 terms (*p*, *d*, *q*):

- *p* is the order of the AR term. It refers to the number of lags.
- *d* is the number of differencing required to make the time series stationary referring to the I term.
- *q* is the order of the MA term. It refers to the number of lagged forecast errors that should go into the ARIMA model.

Main steps to implement the ARIMA model:

1. Checking if the Time Series is stationary
2. Determine *d* - the number of differencing required to make the time series stationary
3. Determining *p* - the order of the AR term
4. Determining *q* - the order of the MA term
5. Fitting the ARIMA model

We will begin building the ARIMA model by checking if the Time Series is stationary and subsequently determining the d term for the model.

### Checking if the Time Series is stationary

It's firstly important to check if the time series is stationary.

This can be done using the '*adfuller*' function from the '*statsmodels*' library which performs the Augmented Dickey-Fuller (ADF) test on the S&P 500 Closing prices to show the presence of serial correlation in our time series. With Time Series data, this test generally proves that the time series is non-stationary.

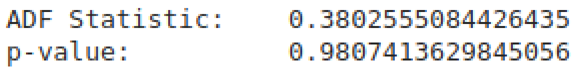
To perform an ADF test on our model, we need to first create a new variable '*SP_500_Close_Values*' which will contain the entire dataset. Then we will isolate and invert the '*Close*' column from said dataset which is all that is needed for the ARIMA model. This column will contain all the closing prices of the S&P 500 monthly from '03/2018' to '11/2021'.

The data from the dataset isn't in the correct format for the ARIMA model, therefore we will have to alter the data to suit our needs. This alteration will include:

- Creating a new variable '*SP_500_Close_Values*' to contain all the prices of S&P 500.
- Reversing the order of the variable to make the data usable
- Resetting the 'index' column to ensure that our plot is in the correct order

In our case, if the p-value > 0.05 meaning the time series is non-Stationary.

```
## Importing the 'adfuller' function from the 'statsmodels' library
from statsmodels.tsa.stattools import adfuller

## Creates a Variable 'SP_500_Close_Values' which contains all
## the content from the 'SP_500_Close' Variable and fit to our plot.
SP_500_Close_Values = SP_500_Close
Reverse_SP_500_Close_Values = SP_500_Close_Values[::-1]
Reverse_SP_500_Close_Values.reset_index(inplace=True, drop=True)

## Uses the 'adfuller' function to check if the time series
## is stationary or not
result = adfuller(Reverse_SP_500_Close_Values.dropna())

print('ADF Statistic:    ', result[0])
print('p-value:          ', result[1])
```



```
ADF Statistic:    0.3802555084426435
p-value:          0.9807413629845056
```

Our p-value is 0.98 so it is clear this price series is non-stationary. This isn't a problem since we could have probably guessed that the time in which the stock prices were collected will influence future results.

To make the ARIMA model work for this time series we need to difference them to remove the trends in the time series. It might take multiple differences to remove the influence of trends in the time series. The goal of this method is to make the time series stationary. The number of differences required is known as the order of differencing (*d*).

**Determine *d* - the number of differencing required to make the time series stationary**

We can use the '*plot_acf*' function from '*statsmodels*' and the ACF plot tells us how many terms are required to remove any autocorrelation in the series.

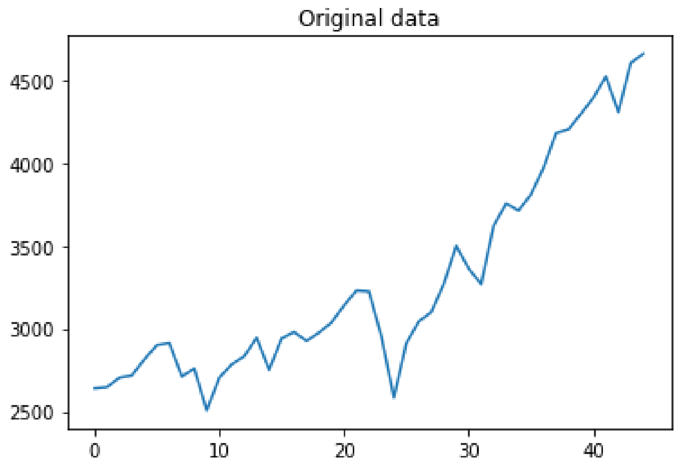
***Autocorrelation Function (ACF)***

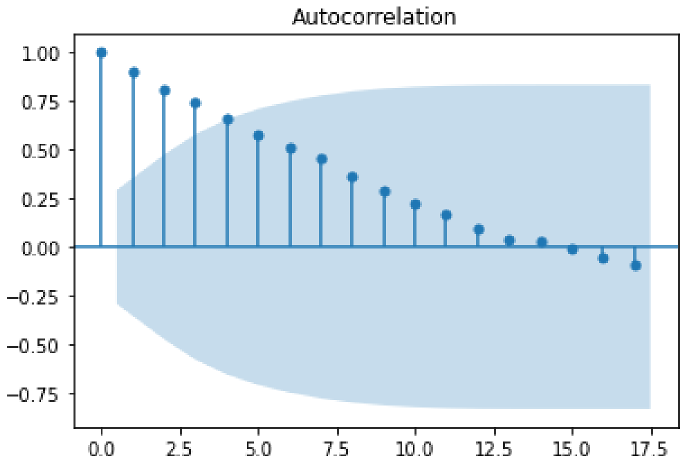
Autocorrelation refers to the correlation between a Time series and a previous version of the time series.

For this dataset, we shifted the time series back by one month. This shift is known as the lag.

The '*statsmodels*' library has a function '*plot_acf*' to plot the autocorrelation for our time series data. This function plots the lags on the horizontal axis and the correlations on the vertical axis.

```
from statsmodels.graphics.tsaplots import plot_acf

## Plots the autocorrelation for the closing prices of the S&P 500
## in our dataset
plt.plot(Reverse_SP_500_Close_Values)
plt.title('Original data')
plot_acf(Reverse_SP_500_Close_Values)
plt.show()
```

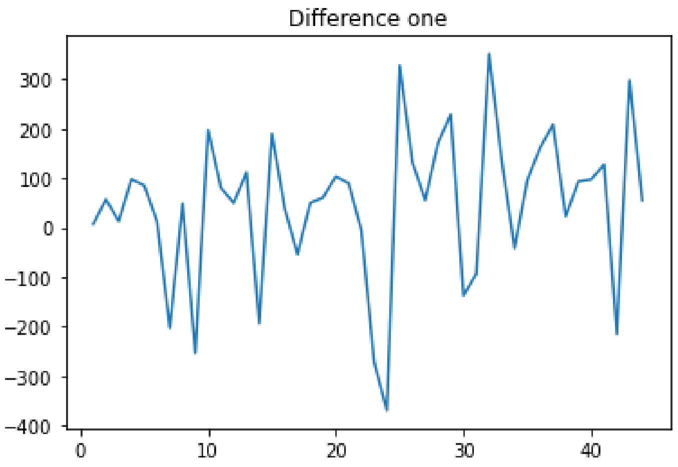


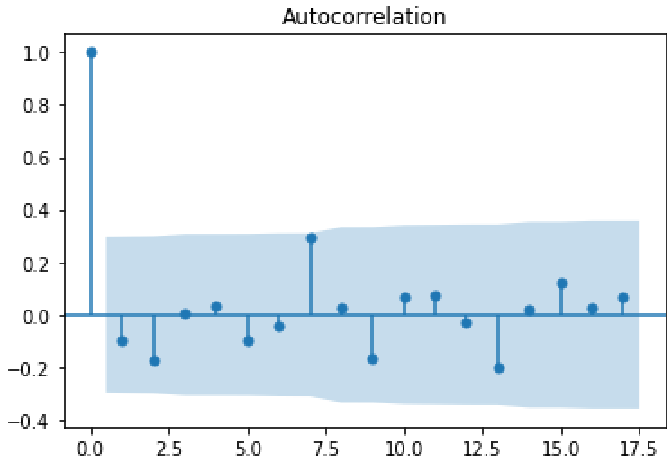


First-order differencing: $x'_t = x_t x_{t-1}$

Next, we will plot the difference of the closing price data along with the autocorrelation of the difference for the closing prices of the S&P 500 in our dataset.

```
## Calculates the difference of the closing price data
diff = Reverse_SP_500_Close_Values.diff().dropna()

## Plots the difference of the closing price data along with
## the autocorrelation of the difference for the closing prices
## of the S&P 500 in our dataset
plt.plot(diff)
plt.title('Difference one')
plot_acf(diff)
plt.show()
```



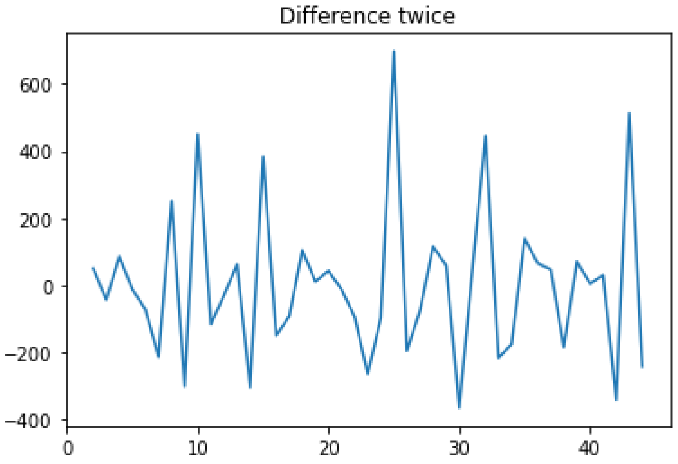


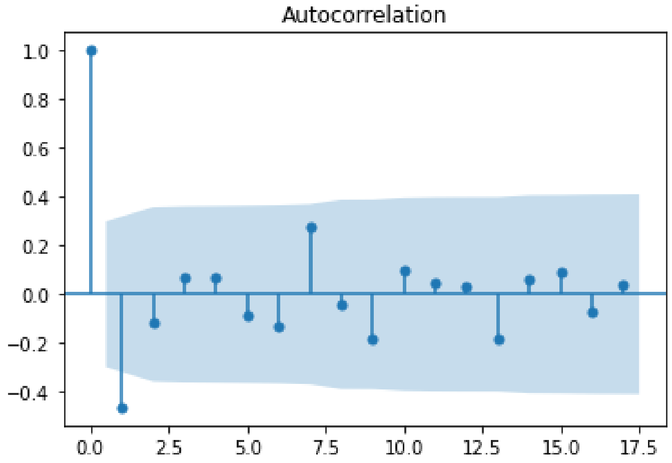
$$\text{Second-order differencing: } x_t* = x'_t - x'_{t-1}$$

$$= (x_t - x_{t-1}) - (x_{t-1} - x_{t-2})$$

$$= x_t - 2x_{t-1} + x_{t-2}$$

```
## Calculates the difference of the difference of the closing price data
diff = Reverse_SP_500_Close_Values.diff().diff().dropna()

## Plots the second difference of the closing price data along with
## the autocorrelation of the difference for the closing prices
## of the S&P 500 in our dataset
plt.plot(diff)
plt.title("Difference twice")
## Add ; to the end of the plot function so that the plot is not
## duplicated
plot_acf(diff);
plt.show()
```



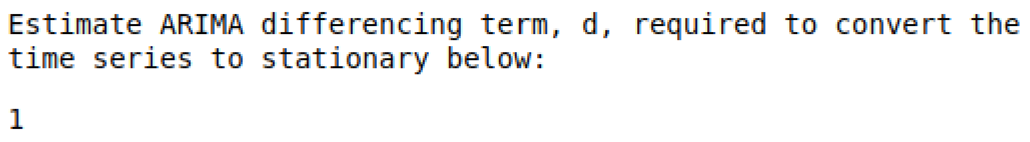


Lag terms in the graph skew to the lag axis when the series is differenced twice indicating that the series is over-differenced. So, we will set the *d* term as 1.

Our order of differencing term (*d*) = 1

The d term will also be confirmed below with the function '*pmdarima.arima.ndiffs*':

```
## Estimates the ARIMA differencing term d required to make the time
## series stationary
from pmdarima.arima.utils import ndiffs

print("Estimate ARIMA differencing term, d, required to convert the time
series to stationary below:")
ndiffs(Reverse_SP_500_Close_Values, test="adf")
```



```
Estimate ARIMA differencing term, d, required to convert the
time series to stationary below:

1
```

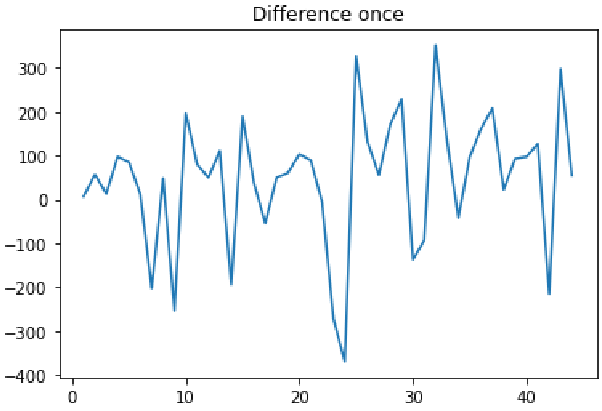
Since we came to the same conclusion, we can be reasonably confident in our result.

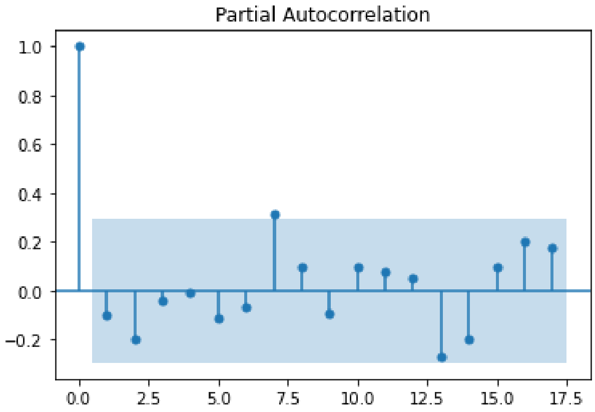
### Determining *p* - the order of the AR term

*p* is the order of the Auto Regressive (AR) term. It refers to the number of lags to be used as predictors.

We can find out the required number of AR terms by inspecting the Partial Autocorrelation (PACF) plot. The partial autocorrelation represents the correlation between the series and its lags.

```
from statsmodels.graphics.tsaplots import plot_pacf

diff = Reverse_SP_500_Close_Values.diff().dropna()

plt.plot(diff)
plt.title('Difference once')
plot_pacf(diff)
plt.show()
```



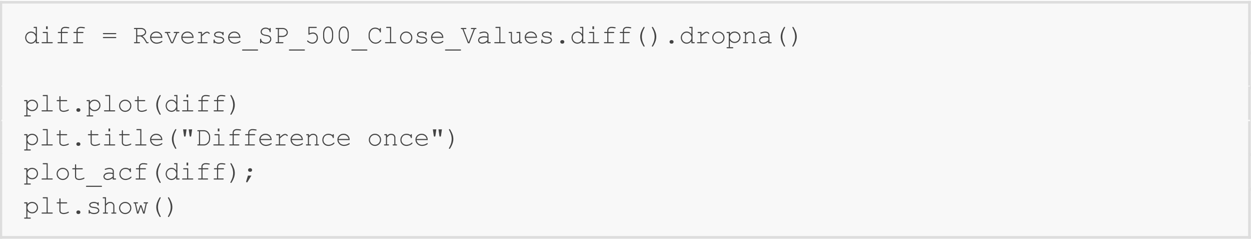


The partial autocorrelation lag number 7 is above the significance line; hence we will set the *p* term as 7.

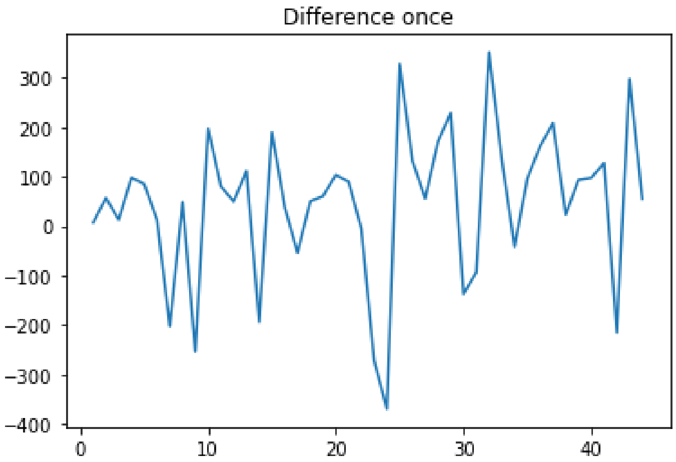
### Determining *q* - the order of the MA term

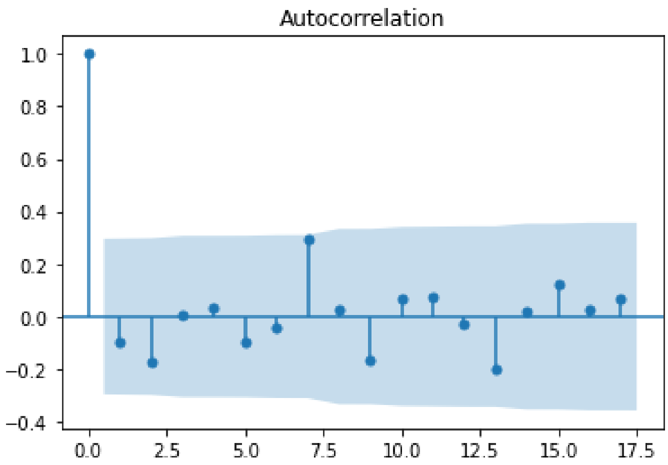
The *q* term is the order of the Moving Average (MA), which refers to the number of lagged forecast errors that should go in the ARIMA model.

We can look at the ACF plot for the number of MA terms.



```
diff = Reverse_SP_500_Close_Values.diff().dropna()

plt.plot(diff)
plt.title("Difference once")
plot_acf(diff);
plt.show()
```





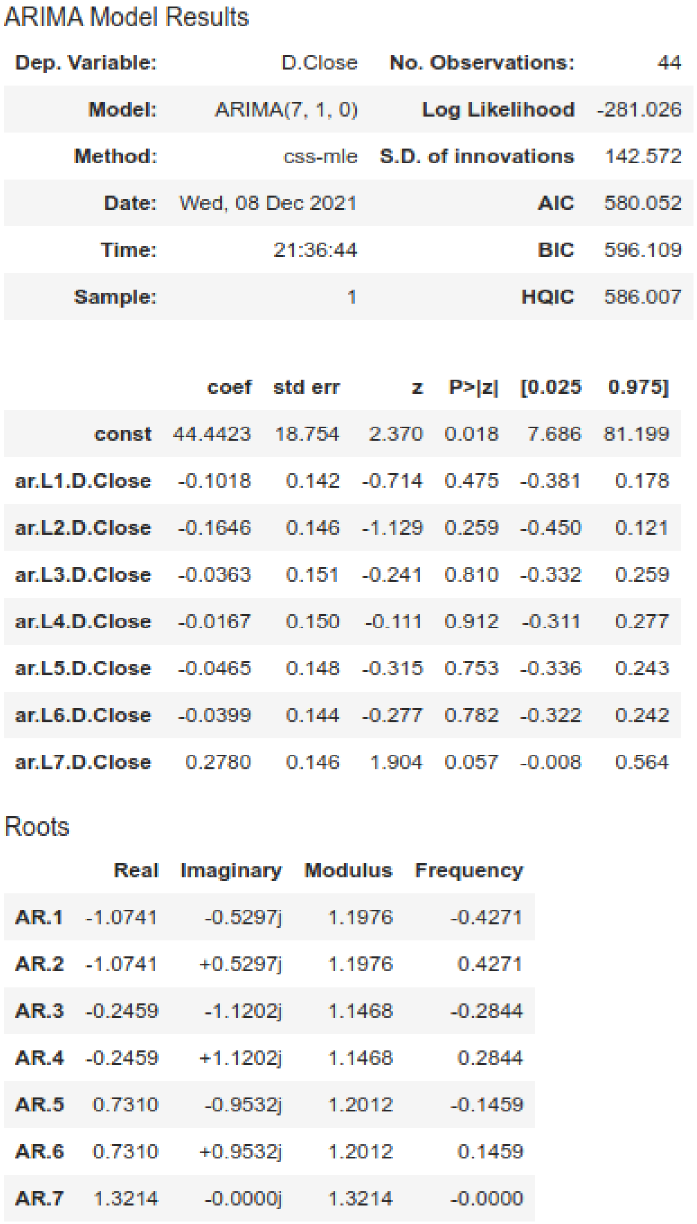
We choose a forecast of 0 due to the fact that no lag is over the significant level

**Fitting the ARIMA model**

ARIMA (*p*, *d*, *q*) = ARIMA (7, 1, 0)

The model is prepared on the training data by calling the *fit()* function.

```
from statsmodels.tsa.arima_model import ARIMA

import warnings
warnings.filterwarnings('ignore')      # Ignore unnecessary warnings

# ARIMA Model
model = ARIMA(Reverse_SP_500_Close_Values, order=(7, 1, 0))
result = model.fit()
result.summary()
```



ARIMA Model Results

| | | | |
|---|---|---|---|
| **Dep. Variable:** | D.Close | **No. Observations:** | 44 |
| **Model:** | ARIMA(7, 1, 0) | **Log Likelihood** | -281.026 |
| **Method:** | css-mle | **S.D. of innovations** | 142.572 |
| **Date:** | Wed, 08 Dec 2021 | **AIC** | 580.052 |
| **Time:** | 21:36:44 | **BIC** | 596.109 |
| **Sample:** | 1 | **HQIC** | 586.007 |

| | **coef** | **std err** | **z** | **P>\|z\|** | **[0.025** | **0.975]** |
|---|---|---|---|---|---|---|
| **const** | 44.4423 | 18.754 | 2.370 | 0.018 | 7.686 | 81.199 |
| **ar.L1.D.Close** | -0.1018 | 0.142 | -0.714 | 0.475 | -0.381 | 0.178 |
| **ar.L2.D.Close** | -0.1646 | 0.146 | -1.129 | 0.259 | -0.450 | 0.121 |
| **ar.L3.D.Close** | -0.0363 | 0.151 | -0.241 | 0.810 | -0.332 | 0.259 |
| **ar.L4.D.Close** | -0.0167 | 0.150 | -0.111 | 0.912 | -0.311 | 0.277 |
| **ar.L5.D.Close** | -0.0465 | 0.148 | -0.315 | 0.753 | -0.336 | 0.243 |
| **ar.L6.D.Close** | -0.0399 | 0.144 | -0.277 | 0.782 | -0.322 | 0.242 |
| **ar.L7.D.Close** | 0.2780 | 0.146 | 1.904 | 0.057 | -0.008 | 0.564 |

Roots

| | **Real** | **Imaginary** | **Modulus** | **Frequency** |
|---|---|---|---|---|
| **AR.1** | -1.0741 | -0.5297j | 1.1976 | -0.4271 |
| **AR.2** | -1.0741 | +0.5297j | 1.1976 | 0.4271 |
| **AR.3** | -0.2459 | -1.1202j | 1.1468 | -0.2844 |
| **AR.4** | -0.2459 | +1.1202j | 1.1468 | 0.2844 |
| **AR.5** | 0.7310 | -0.9532j | 1.2012 | -0.1459 |
| **AR.6** | 0.7310 | +0.9532j | 1.2012 | 0.1459 |
| **AR.7** | 1.3214 | -0.0000j | 1.3214 | -0.0000 |

The model summary gives us a lot of information about our ARIMA model. The important areas of focus from the summary:

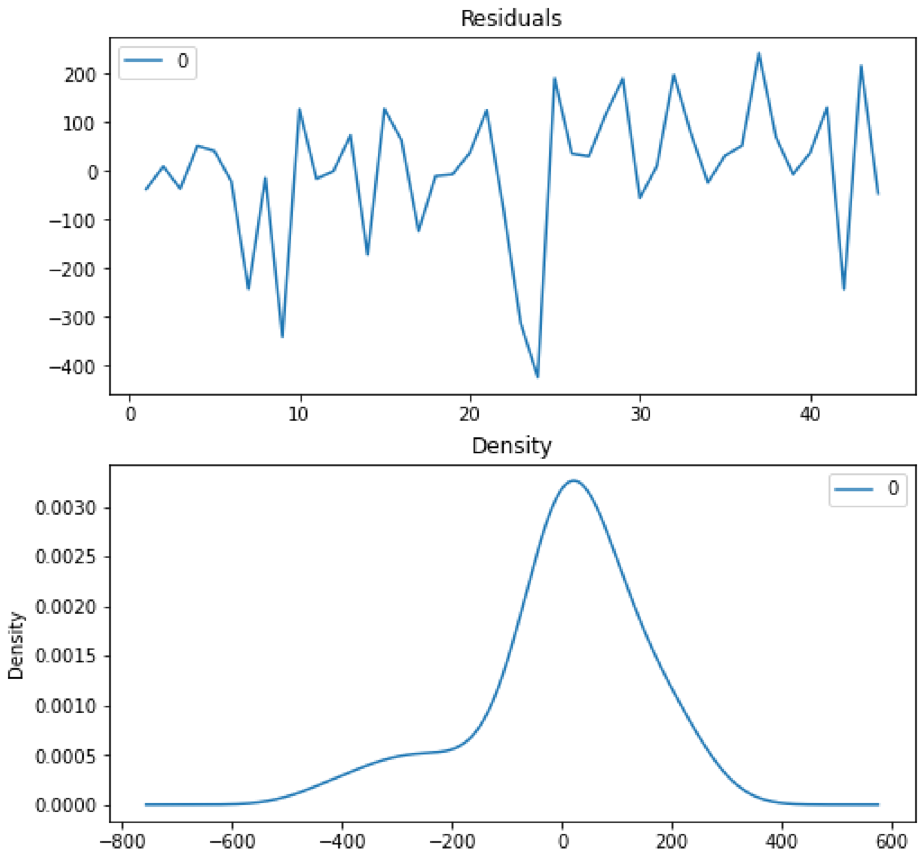
***Residuals***

The '*residuals*' in a time series model are what is left over after fitting a model.

The residual errors of a time series model are equal to the difference between the expected values and the predicted values.

Plot the residual errors to ensure there are no patterns (that is, looking for constant mean and variance).

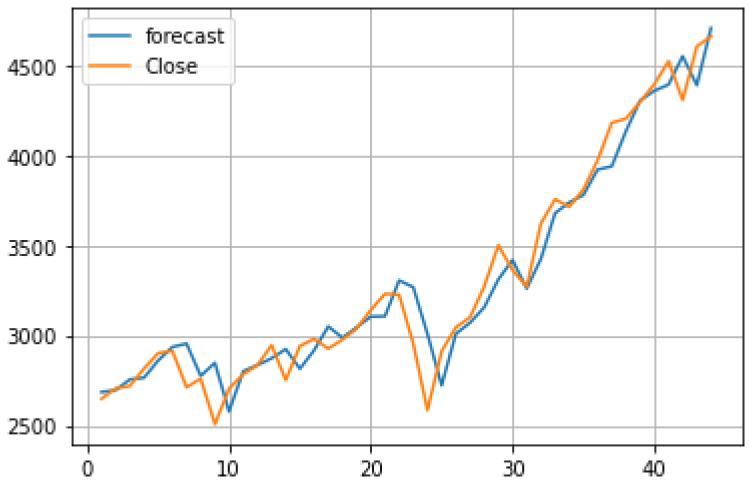
```
# Plot residual errors
residuals = pd.DataFrame(result.resid)

fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(8, 8))

residuals.plot(title = 'Residuals', ax=ax1)
residuals.plot(kind='kde', title = 'Density', ax=ax2)
plt.show()
```



The residual errors seem reasonably fine with a near zero mean and uniform variance.

Now, let's plot the actual values against the predicted values using the '*plot_predict()*'.

```
# Actual values vs Predicted values
result.plot_predict(
    start=1, end=44, dynamic=False,
)

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

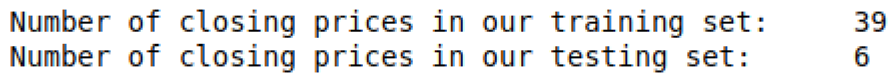


Creating a train and test set to use the ARIMA model:

- The train set will contain 87.5% of our data
- The test set will then contain the remaining 12.5% of the data

```
## 'n' is an integer that is the equivalent to 87.5% of the
## amount of closing prices from our dataset
n = int(len(Reverse_SP_500_Close_Values) * 0.875)
## Our training set contains the first 87.5% of closing prices
## from our dataset
train = Reverse_SP_500_Close_Values[:n]
## Our testing set will contain the remainder of closing prices
## from our dataset
test = Reverse_SP_500_Close_Values[n:]
```

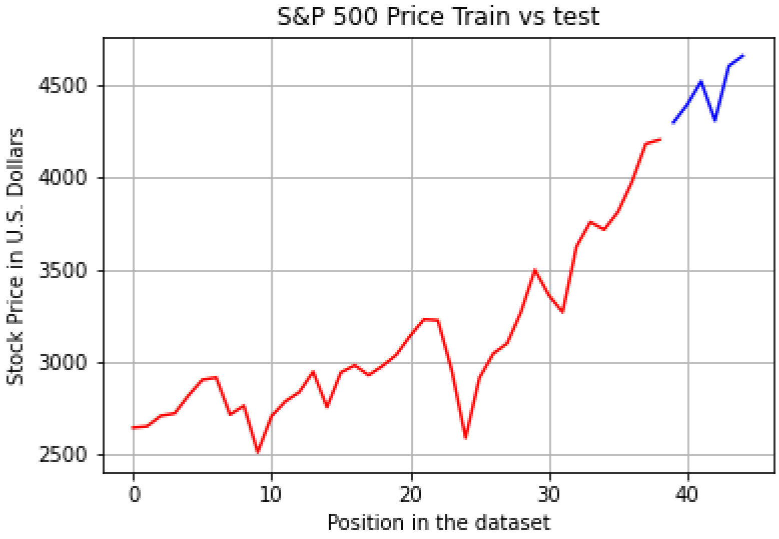
```
print('Number of closing prices in our training set:    ', len(train))
print('Number of closing prices in our testing set:     ', len(test))
```



```
Number of closing prices in our training set:     39
Number of closing prices in our testing set:      6
```

We will plot the training vs testing data from the S&P 500 closing price data.

```
## Plots the training vs testing data from the S&P 500 data
plt.title('S&P 500 Price Train vs test')
plt.xlabel('Position in the dataset')
plt.ylabel('Stock Price in U.S. Dollars')
plt.grid(True)
plt.plot(train, color = 'red')
plt.plot(test, color = 'blue')

plt.show()
```



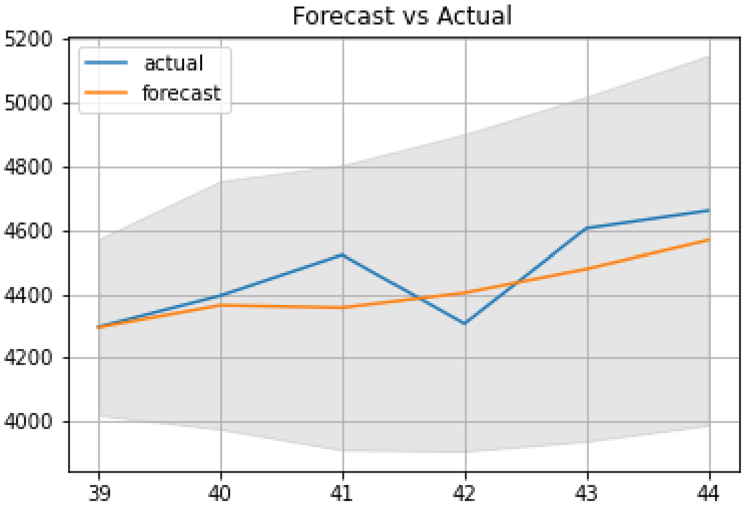
```
model = ARIMA(train, order=(7, 1, 0))
result = model.fit()
```

```
step = 6

# Forecast future values using our ARIMA model with the '.forecast' function.
fc, se, conf = result.forecast(step, alpha=0.05) # 95% confidence interval
```

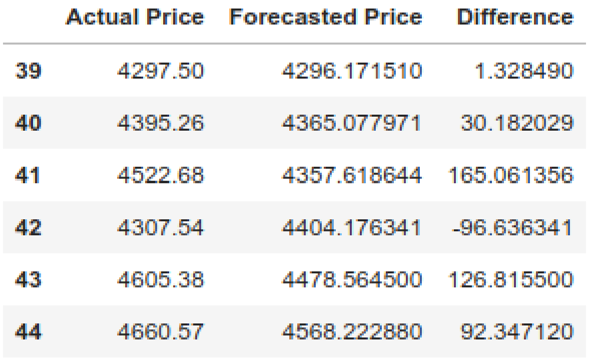
```
## Make as pandas series to store the forecasted values
fc = pd.Series(fc, index=test[:step].index)

## Make as separate pandas series to store the upper and lower confidence
levels
lower = pd.Series(conf[:, 0], index=test[:step].index)
upper = pd.Series(conf[:, 1], index=test[:step].index)
```

We can now plot the forecasted values vs the actual values showing our confidence levels.

```
plt.plot(test[:step], label="actual")
plt.plot(fc, label="forecast")
plt.fill_between(lower.index, lower, upper, color="k", alpha=0.1)
plt.title("Forecast vs Actual")
plt.legend(loc="upper left")
plt.grid(True)
plt.show()
```

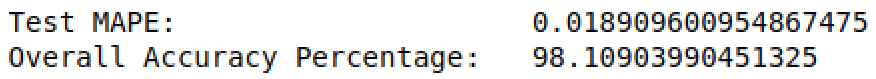


```
answer = pd.DataFrame({'Actual Price' :test['Close'], 'Forecasted Price' :fc,
'Difference' :test['Close']-fc})
answer
```



| | Actual Price | Forecasted Price | Difference |
|---|---|---|---|
| **39** | 4297.50 | 4296.171510 | 1.328490 |
| **40** | 4395.26 | 4365.077971 | 30.182029 |
| **41** | 4522.68 | 4357.618644 | 165.061356 |
| **42** | 4307.54 | 4404.176341 | -96.636341 |
| **43** | 4605.38 | 4478.564500 | 126.815500 |
| **44** | 4660.57 | 4568.222880 | 92.347120 |

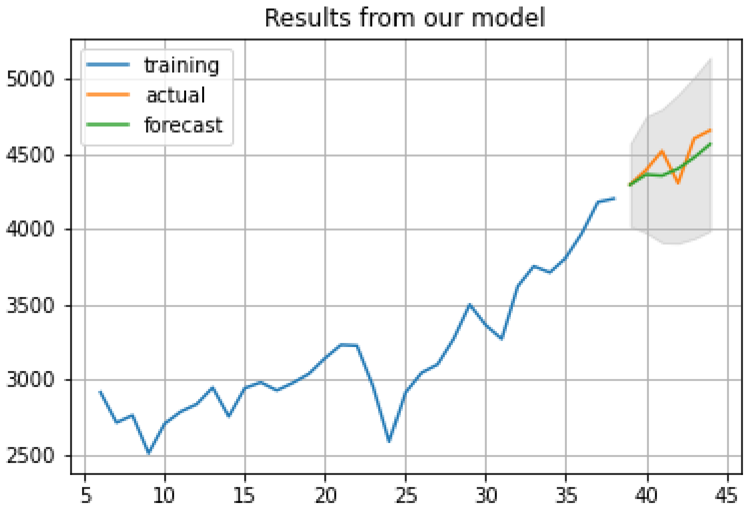
```
from math import sqrt
import numpy as np

# Calculate the Mean Absolute Percentage Error
mape = np.mean(np.abs(fc - test['Close'])/np.abs(test['Close']))
print('Test MAPE', mape)

print(1 - mape)
```



```
Test MAPE:                      0.018909600954867475
Overall Accuracy Percentage:    98.10903990451325
```

This implies that the model will be 9.81% accurate in predicting the next 6 observations.

```
plt.plot(train[step:], label='training')
plt.plot(test[:step], label='actual')
plt.plot(fc, label='forecast')
plt.fill_between(lower.index, lower, upper, color='k', alpha=0.1)
plt.title('Results from our model')
plt.legend(loc='upper left')
plt.grid(True)
plt.show()
```



**Model 2: SARIMA**

Seasonal Autoregressive Integrated Moving Average (SARIMA) model builds upon the ARIMA model. It is formed by including additional seasonal terms (P, D, Q, m) to the ARIMA model. It can be denoted as follows:

$$\text{ARIMA} \qquad \underbrace{(p, d, q)}_{\substack{\uparrow \\ \text{Non-seasonal part} \\ \text{of the model}}} \qquad \underbrace{(P, D, Q)_m}_{\substack{\uparrow \\ \text{Seasonal part of} \\ \text{of the model}}}$$

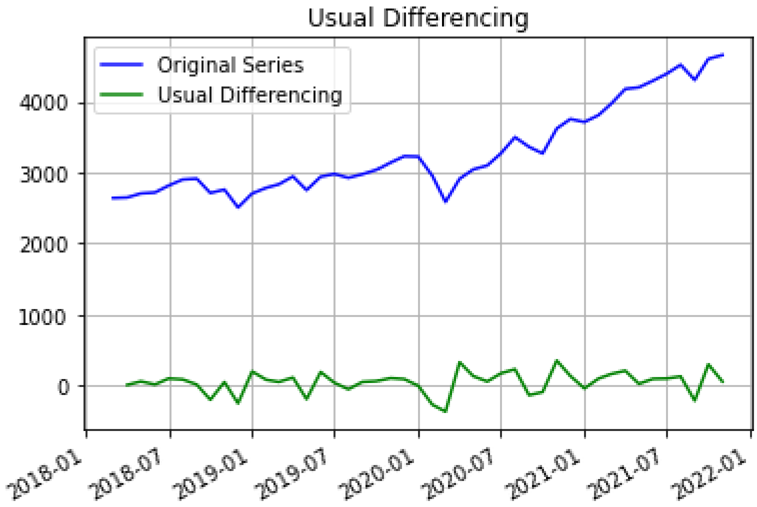
A SARIMA model is characterised by 4 additional terms ( *P*, *D*, *Q*, *m*):

- *P* is the order of the seasonal AR term.
- *D* is the number of seasonal differencing referring to the I term.
- *Q* is the order of the seasonal MA term.
- *m* is the number of time steps for a single seasonal period

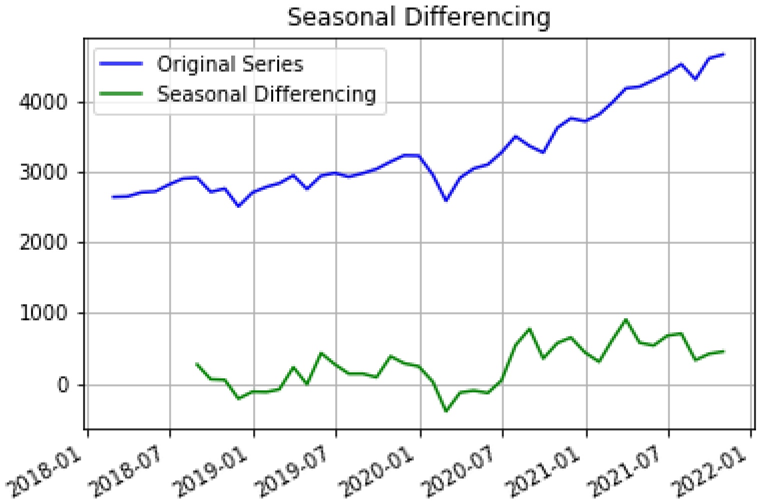
To begin, we will load the '*SaP_500_3yrs_Monthly.csv*' file as a new DataFrame ('*data*') to remove any influence from our ARIMA model built earlier. In this DataFrame, we will use the '*Date*' column as the index for the DataFrame.

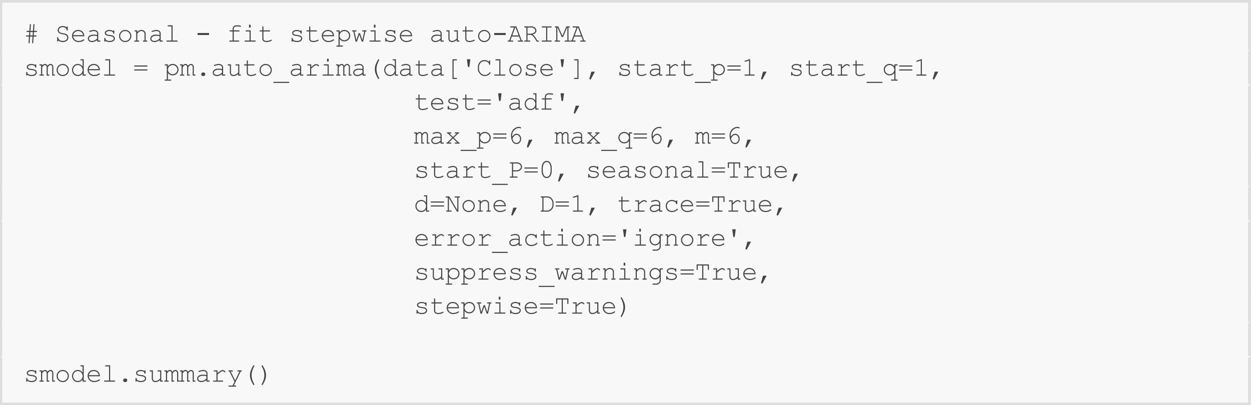
We will then remove all the columns in the '*data*' DataFrame except the '*Close*' column which is linked to the correct dates since the '*Date*' is the index for the DataFrame.

```
## We will load the 'SaP_500_3yrs_Monthly.csv' file as a new DataFrame
## ('data') to remove any influence from our ARIMA model built earlier
## using the 'Date' column as the index for the DataFrame
data = pd.read_csv('SaP_500_3yrs_Monthly.csv', parse_dates=['Date'],
index_col='Date')

## Removes all the columns in the 'data' DataFrame except the
## 'Close' column which is linked to the correct dates since the
## 'Date' is the index for the DataFrame
data = data[['Close']].dropna()
data = data[::-1]



plt.plot(data, label='Original Series', color = 'blue')
plt.plot(data.diff(1), label='Usual Differencing', color = 'green')
plt.title('Usual Differencing')
plt.legend(loc='upper left', fontsize=10)
plt.grid(True)
plt.gcf().autofmt_xdate()

plt.show()
```

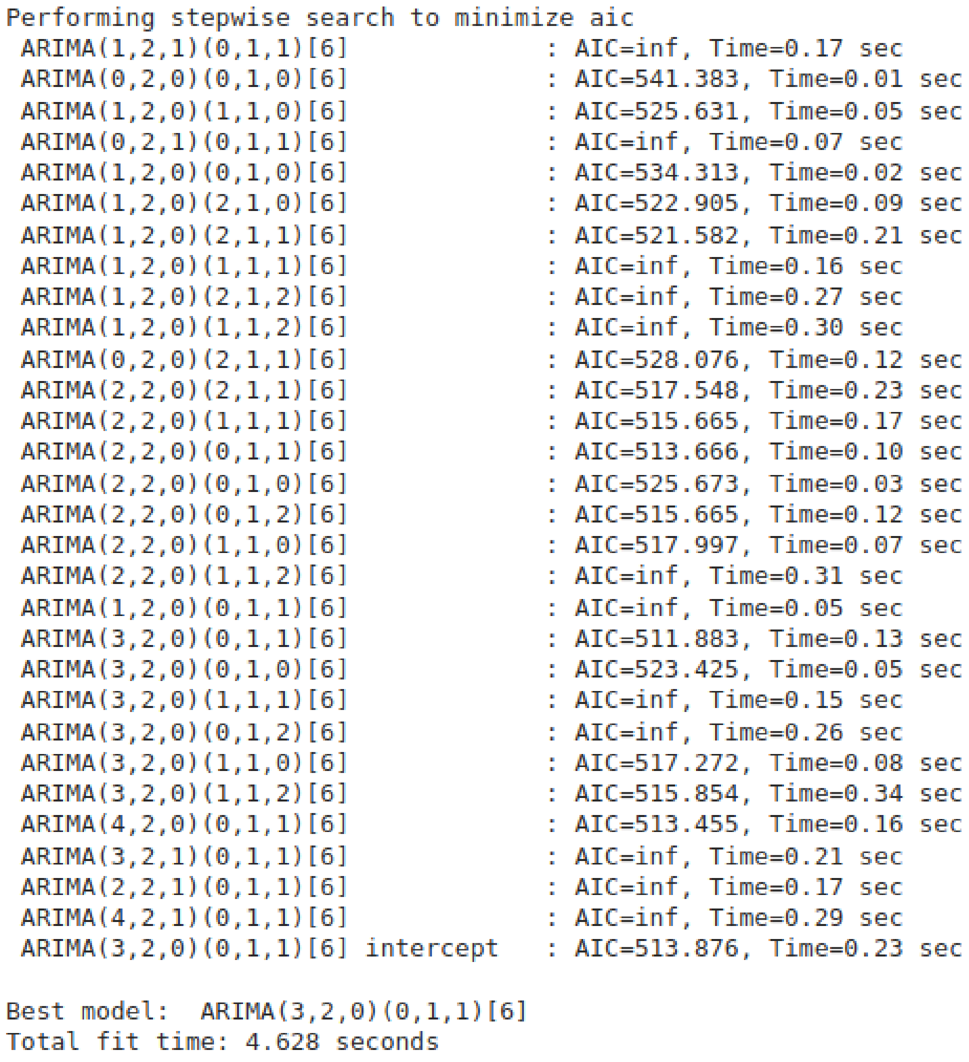


```
plt.plot(data, label='Original Series', color = 'blue')
plt.plot(data.diff(6), label='Seasonal Differencing', color = 'green')
plt.title('Seasonal Differencing')
plt.legend(loc='upper left', fontsize=10)
plt.grid(True)
plt.gcf().autofmt_xdate()

plt.show()
```

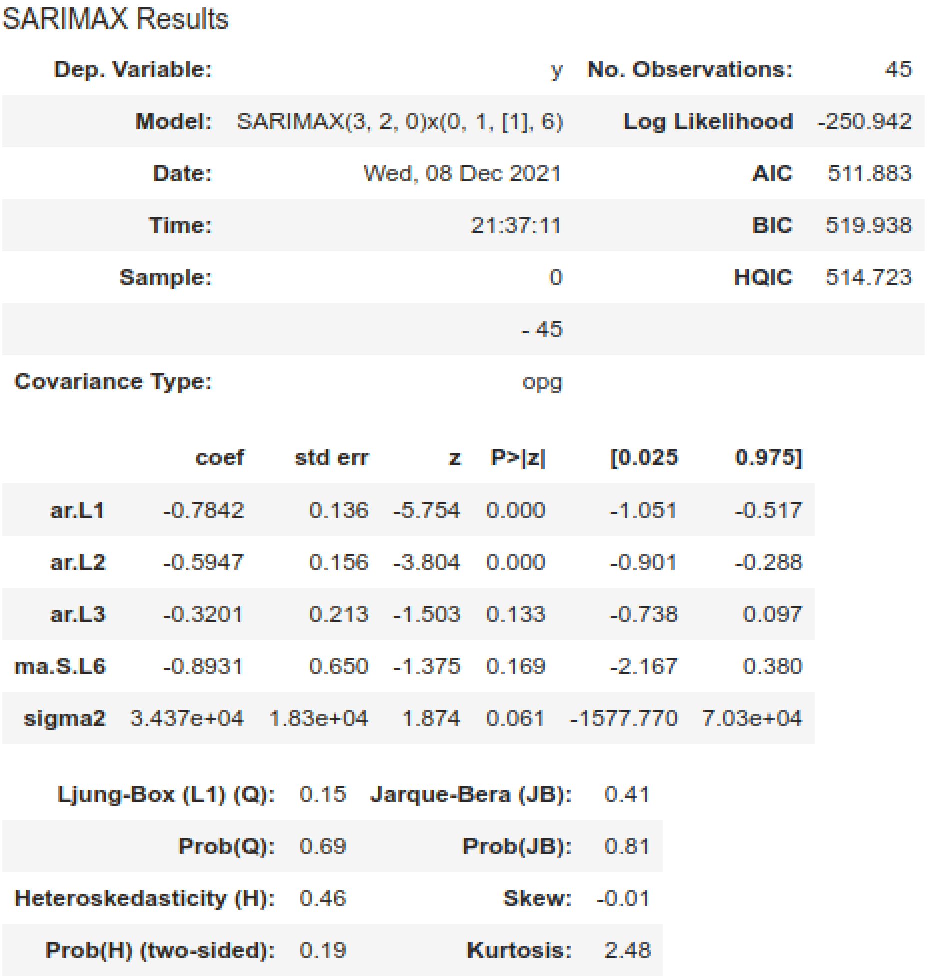




```
# Seasonal - fit stepwise auto-ARIMA
smodel = pm.auto_arima(data['Close'], start_p=1, start_q=1,
                         test='adf',
                         max_p=6, max_q=6, m=6,
                         start_P=0, seasonal=True,
                         d=None, D=1, trace=True,
                         error_action='ignore',
                         suppress_warnings=True,
                         stepwise=True)

smodel.summary()
```



```
Performing stepwise search to minimize aic
 ARIMA(1,2,1)(0,1,1)[6]             : AIC=inf, Time=0.17 sec
 ARIMA(0,2,0)(0,1,0)[6]             : AIC=541.383, Time=0.01 sec
 ARIMA(1,2,0)(1,1,0)[6]             : AIC=525.631, Time=0.05 sec
 ARIMA(0,2,1)(0,1,1)[6]             : AIC=inf, Time=0.07 sec
 ARIMA(1,2,0)(0,1,0)[6]             : AIC=534.313, Time=0.02 sec
 ARIMA(1,2,0)(2,1,0)[6]             : AIC=522.905, Time=0.09 sec
 ARIMA(1,2,0)(2,1,1)[6]             : AIC=521.582, Time=0.21 sec
 ARIMA(1,2,0)(1,1,1)[6]             : AIC=inf, Time=0.16 sec
 ARIMA(1,2,0)(2,1,2)[6]             : AIC=inf, Time=0.27 sec
 ARIMA(1,2,0)(1,1,2)[6]             : AIC=inf, Time=0.30 sec
 ARIMA(0,2,0)(2,1,1)[6]             : AIC=528.076, Time=0.12 sec
 ARIMA(2,2,0)(2,1,1)[6]             : AIC=517.548, Time=0.23 sec
 ARIMA(2,2,0)(1,1,1)[6]             : AIC=515.665, Time=0.17 sec
 ARIMA(2,2,0)(0,1,1)[6]             : AIC=513.666, Time=0.10 sec
 ARIMA(2,2,0)(0,1,0)[6]             : AIC=525.673, Time=0.03 sec
 ARIMA(2,2,0)(0,1,2)[6]             : AIC=515.665, Time=0.12 sec
 ARIMA(2,2,0)(1,1,0)[6]             : AIC=517.997, Time=0.07 sec
 ARIMA(2,2,0)(1,1,2)[6]             : AIC=inf, Time=0.31 sec
 ARIMA(1,2,0)(0,1,1)[6]             : AIC=inf, Time=0.05 sec
 ARIMA(3,2,0)(0,1,1)[6]             : AIC=511.883, Time=0.13 sec
 ARIMA(3,2,0)(0,1,0)[6]             : AIC=523.425, Time=0.05 sec
 ARIMA(3,2,0)(1,1,1)[6]             : AIC=inf, Time=0.15 sec
 ARIMA(3,2,0)(0,1,2)[6]             : AIC=inf, Time=0.26 sec
 ARIMA(3,2,0)(1,1,0)[6]             : AIC=517.272, Time=0.08 sec
 ARIMA(3,2,0)(1,1,2)[6]             : AIC=515.854, Time=0.34 sec
 ARIMA(4,2,0)(0,1,1)[6]             : AIC=513.455, Time=0.16 sec
 ARIMA(3,2,1)(0,1,1)[6]             : AIC=inf, Time=0.21 sec
 ARIMA(2,2,1)(0,1,1)[6]             : AIC=inf, Time=0.17 sec
 ARIMA(4,2,1)(0,1,1)[6]             : AIC=inf, Time=0.29 sec
 ARIMA(3,2,0)(0,1,1)[6] intercept   : AIC=513.876, Time=0.23 sec

Best model:  ARIMA(3,2,0)(0,1,1)[6]
Total fit time: 4.628 seconds
```

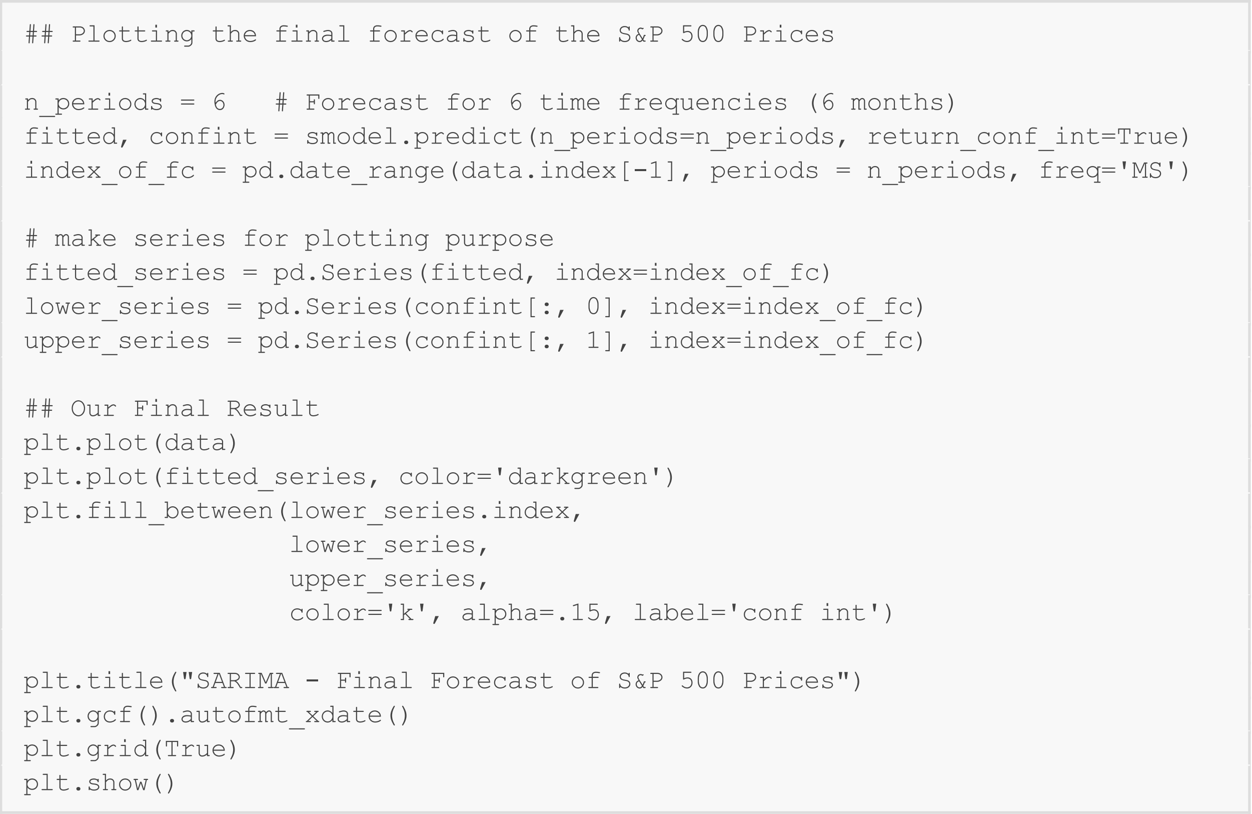


SARIMAX Results

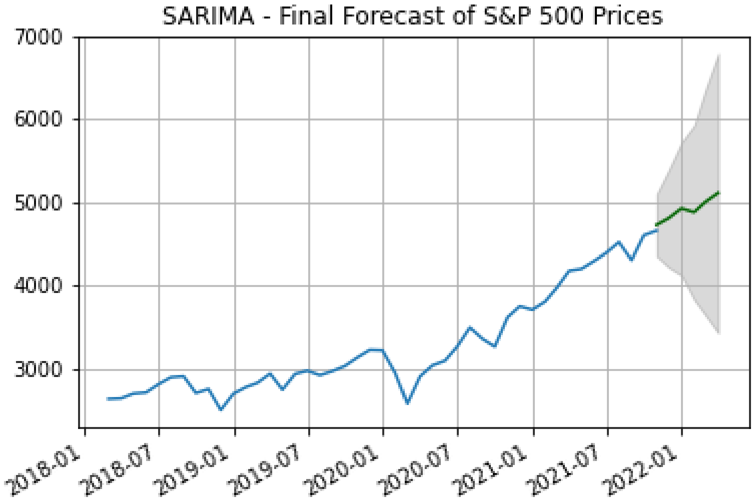
| | | | |
|---|---|---|---|
| **Dep. Variable:** | y | **No. Observations:** | 45 |
| **Model:** | SARIMAX(3, 2, 0)x(0, 1, [1], 6) | **Log Likelihood** | -250.942 |
| **Date:** | Wed, 08 Dec 2021 | **AIC** | 511.883 |
| **Time:** | 21:37:11 | **BIC** | 519.938 |
| **Sample:** | 0 | **HQIC** | 514.723 |
| | - 45 | | |
| **Covariance Type:** | opg | | |

| | **coef** | **std err** | **z** | **P>\|z\|** | **[0.025** | **0.975]** |
|---|---|---|---|---|---|---|
| **ar.L1** | -0.7842 | 0.136 | -5.754 | 0.000 | -1.051 | -0.517 |
| **ar.L2** | -0.5947 | 0.156 | -3.804 | 0.000 | -0.901 | -0.288 |
| **ar.L3** | -0.3201 | 0.213 | -1.503 | 0.133 | -0.738 | 0.097 |
| **ma.S.L6** | -0.8931 | 0.650 | -1.375 | 0.169 | -2.167 | 0.380 |
| **sigma2** | 3.437e+04 | 1.83e+04 | 1.874 | 0.061 | -1577.770 | 7.03e+04 |

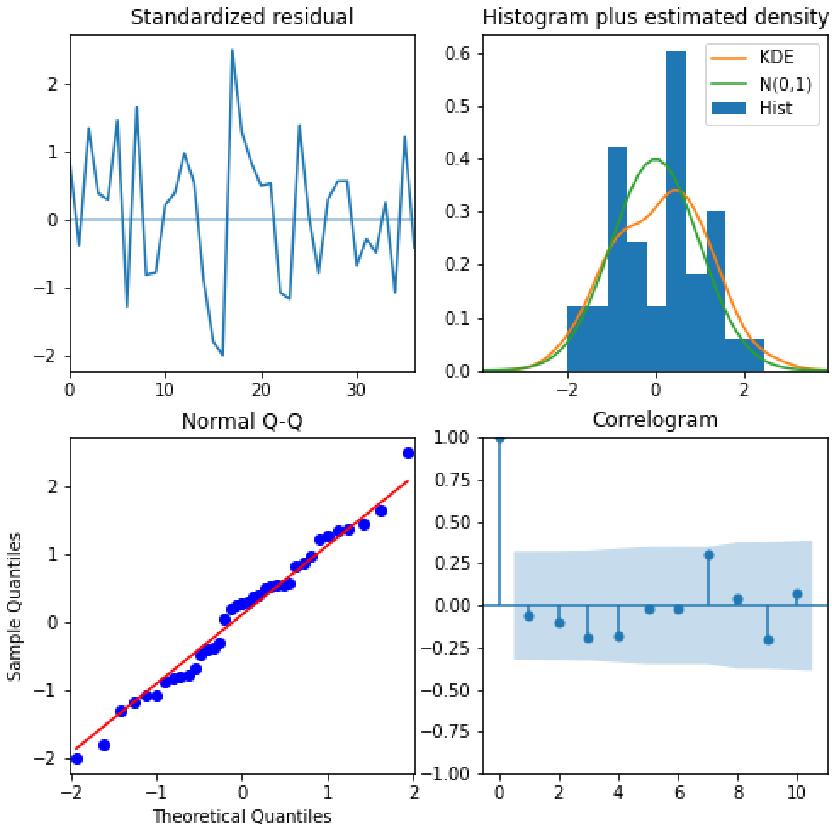
| | | | |
|---|---|---|---|
| **Ljung-Box (L1) (Q):** | 0.15 | **Jarque-Bera (JB):** | 0.41 |
| **Prob(Q):** | 0.69 | **Prob(JB):** | 0.81 |
| **Heteroskedasticity (H):** | 0.46 | **Skew:** | -0.01 |
| **Prob(H) (two-sided):** | 0.19 | **Kurtosis:** | 2.48 |



```
## Plotting the final forecast of the S&P 500 Prices

n_periods = 6   # Forecast for 6 time frequencies (6 months)
fitted, confint = smodel.predict(n_periods=n_periods, return_conf_int=True)
index_of_fc = pd.date_range(data.index[-1], periods = n_periods, freq='MS')

# make series for plotting purpose
fitted_series = pd.Series(fitted, index=index_of_fc)
lower_series = pd.Series(confint[:, 0], index=index_of_fc)
upper_series = pd.Series(confint[:, 1], index=index_of_fc)

## Our Final Result
plt.plot(data)
plt.plot(fitted_series, color='darkgreen')
plt.fill_between(lower_series.index,
                 lower_series,
                 upper_series,
                 color='k', alpha=.15, label='conf int')

plt.title("SARIMA - Final Forecast of S&P 500 Prices")
plt.gcf().autofmt_xdate()
plt.grid(True)
plt.show()
```

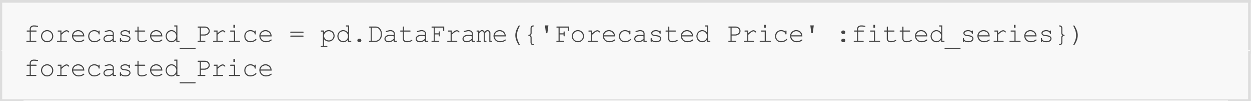


```
from statsmodels.tsa.arima.model import ARIMAResults

smodel.plot_diagnostics(figsize=(8,8))
plt.show()
```

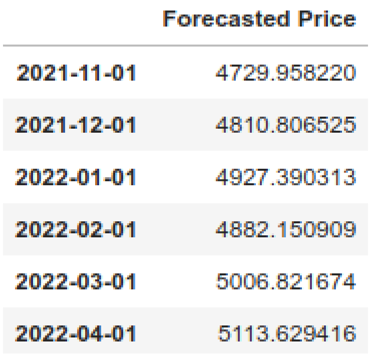


Interpreting the above plot diagnostics:

- The residual errors fluctuate around the mean of zero and have a uniform variance.
- The density plot suggests normal distribution with mean one.
- The Q-Q plot: shows that the ordered distribution of residuals (blue dots) follows the linear trend of the time series data from a standard normal distribution. All the dots should fall perfectly in line with the red line. Any significant deviations would imply the distribution is skewed.
- The Correlogram (ACF plot) shows the residual errors are not autocorrelated. Any autocorrelation would imply that there is some pattern in the residual errors which are not explained in the model.



```
forecasted_Price = pd.DataFrame({'Forecasted Price' :fitted_series})
forecasted_Price
```



| | Forecasted Price |
|---|---|
| **2021-11-01** | 4729.958220 |
| **2021-12-01** | 4810.806525 |
| **2022-01-01** | 4927.390313 |
| **2022-02-01** | 4882.150909 |
| **2022-03-01** | 5006.821674 |
| **2022-04-01** | 5113.629416 |

## Comparing the two models

Through this project, it is apparent that both ARIMA and SARIMA models are applicable Time Series models for forecasting/predicting the stock price of the S&P 500 index fund. Although each Time Series model tackles the Price prediction problem differently, both obtain vital information which can be used to predict future trends in the cryptocurrency. Assessing each isn't all clear cut since each model performs very differently.

Each models will be summarised below:

**Model 1: ARIMA model**

The ARIMA model uses the *'Close'* variable to forecast the future price values of the stock price. It consists of dividing the dataset provided into a training and testing set. The training set is used to train the ARIMA models to calculate the parameters, while the testing set allows us to assess the performance of the model on new and unseen data. This gives instance feedback to how accurate the model is on the price prediction problem.

The biggest issue with using the ARIMA model for forecasting the price of the S&P 500 is that the model doesn't provide new and unique price values for the cryptocurrency due to the fact that the *'Close'* variable in the test set which is the stock price being predicted is already in our dataset. The models also doesn't take into account seasonality which obviously has a huge impact on the results, reducing the reliability of the model.

**Model 2: SARIMA model**

The SARIMA model uses both the *'Date'* and *'Close'* variables to predict the future values of the stock price. The model is trained on past data from our dataset in order to forecast future values for the stock price. It takes into account seasonality by adding additional seasonal terms to the model. This new model is extremely useful since it forecasts future prices of the stock price which can be a good indication whether investing in the index fund is a good idea.

## Conclusion

To conclude, through implementing the ARIMA and SARIMA models to predict/forecast the price of the S&P 500 index fund, the applications of such models for investor's is evident.

Faults in our testing method became apparent