CSIP5203: Big Data Analytics Applications

Time Series Assignment (Coursework 1)

Iftekhirul

P2777638

(Words Count: 2905)

Table of Contents

Abstract ***4***

Problem Description and Methodology ***4***

Activity 1 ***4***

Exploratory Data Analysis **4**

Time Series Visualization **5**

Time Series Decomposition **7**

Naive Method **8**

Fit, forecast and visualization 8

Accuracy Metrics. 9

Comments 10

Average Historical method **10**

Fit, forecast and visualization 10

Accuracy Metrics. 11

Comments 12

Activity 2 ***12***

Time Series Decomposition **12**

Simple Average Method **13**

Fit, forecast and visualization 13

Accuracy Metrics. 14

Comments 14

Exponential Smoothing methods (Single Exponential Smoothing, Holt's Linear, Holt-Winters) **14**

Fit, forecast and visualization 14

Accuracy Metrics. 15

Comments 15

Activity 3 ***17***

Time Series Stationary test and Differencing **17**

ACF and PACF **19**

ARIMA **20**

Fit, forecast and visualization 20

Accuracy Metrics. 22

Comments 22

SARIMA **23**

Fit, forecast and visualization 23

Accuracy Metrics. 25

Comments 25

Conclusion ***25***

References and Bibliography ***26***

Other Resources ***27***

Appendix (if necessary) ***27***

Abstract

Time series analysis and forecasting are the foundation of this assignment. The data will first be thoroughly analyzed and visualized. The time series will be broken down into trend, seasonal, and residual components using time series decomposition. Future trends will be predicted using a variety of forecasting techniques, including the Naive method, Average historical method, Simple average method, three different types of exponential smoothing methods, ARIMA, and SARIMA. Accuracy metrics are used to evaluate the methods.

Problem Description and Methodology

Chariot Oil & Gas Limited is chosen for this assessment taken from Yahoo Finance UK ("0.1"). The data collection period spans 60 months, beginning on January 1, 2019, and ending on December 1, 2023. The only columns taken are "Date" and "Close Price" because they are essential to the project. Here, the "Close Price" refers to the last traded price of a stock or other financial instrument at the end of a particular trading month. It is the final price at which a security was traded during the regular market hours for the month.

Activity 1

EDA, Time series visualization, Time series decomposition, naive method, and average historical method are all included in activity 1. For both the Naive method and the average historical method, fit, forecast, visualization, and accuracy metrics are required.

Exploratory Data Analysis

First, using df.isnull().sum(), no missing values are found in the data after loading the dataset in a dataframe as df. The column 'Close' is renamed as 'Close Price'.

The format of the date is "yyyy-dd-mm" (Fig: 1.1), but it has to be "yyyy-mm-dd" format. I changed the date string format by using the dt.strftime() function.

```
df['Date'] = df['Date'].dt.strftime('%Y-%d-%m')
```

Date string to datetime format is converted using pd.to_datetime() ("1.1")

```
df['Date'] = pd.to_datetime(df['Date'],infer_datetime_format=True)
```

Now, the date is in the right format and set as the index (Fig: 1.2).

```
df.head()
```

	Date	Close Price
0	2019-01-01	2.7163
1	2019-01-02	2.7014
2	2019-01-03	2.3226
3	2019-01-04	4.4359
4	2019-01-05	4.7349

Fig: 1.1

```
df = df.set_index('Date')
df.head()
```

	Close Price
Date	
2019-01-01	2.7163
2019-02-01	2.7014
2019-03-01	2.3226
2019-04-01	4.4359
2019-05-01	4.7349

Fig: 1.2

The table provided by the df.describe() function shows that the total counts are 60, the mean is 9.262, the minimum is 1.47, the standard deviation is 6.17 which indicates high, the first quartile (25%) is 3.76, the second quartile (50%) is 7.751, the third quartile (75%) is 15.162, and the maximum is 22.6 (Fig: 1.3).

```
df.describe().T
```

	count	mean	std	min	25%	50%	75%	max
Close Price	60.0	9.262783	6.170219	1.4703	3.7605	7.75135	15.1625	22.6

Fig: 1.3

Moreover, the values 7.751 and 19.4 for the median and mode, respectively, are provided by the df['Close Price'].median() and df['Close Price'].mode() functions. Using df['Close Price'].var() function I got the variance as 38.071. So, 38.071 is the average of the squared distances from each point to the mean which is 9.262. And the standard deviation 6.17 means the average difference between the mean of a dataset and the distance between every single data point in the dataset is 6.17.

The data are fairly symmetrical because the skewness is 0.451, which is between -0.5 and 0.5 (Fig: 1.4). Furthermore, The data have platykurtic distribution as kurtosis is -1.199 which is less than 3 and 0 (Fig: 1.4). The platykurtic distribution has thinner tails than the normal distribution .

```
print('Skewness:', df['Close Price'].skew().round(3))
print('Kurtosis:', df['Close Price'].kurt().round(3))
```

```
Skewness: 0.451
Kurtosis: -1.199
```

Fig: 1.4

Time Series Visualization

I used the Seaborn Distplot, a type of distribution that shows the overall distribution of continuous data variables, to understand the data. (“1.2”)

```
sns.distplot(df['Close Price'], bins=10, hist=True, kde=True, label = 'Close Price')
```

The two plots in this distplot are the kernel density estimate plot and the plot histogram. The histogram in Figure 1.5 indicates that data frequencies are higher from 2 to 10 and 15 to 20. According to the smoothed line KDE in Fig: 1.5, peaks in the KDE indicate regions with higher data density.

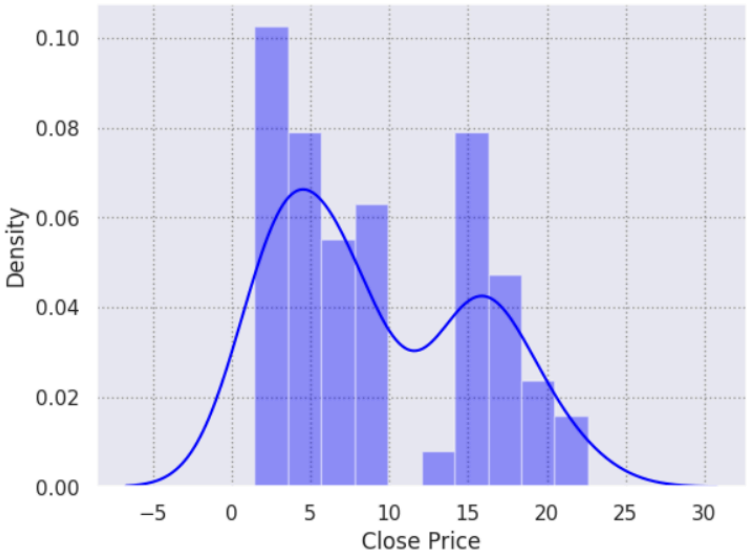


Fig: 1.5

From the boxplot in Fig: 1.6, the term "interquartile range" (IQR) refers to the boxplot's range from the front edge, which is 3.76, to the rare edge, which is 15.16, which contains the middle 50% of the data. The middle line inside of the box is the median, 7.751. The line from the box edge to minimum, 1.47 and maximum, 22.6 is called whiskers. We can see, there are no outliers beyond whiskers.

```
df['Close Price'].plot(kind='box', vert=False, figsize=(14,6))
```

```
<Axes: >
```

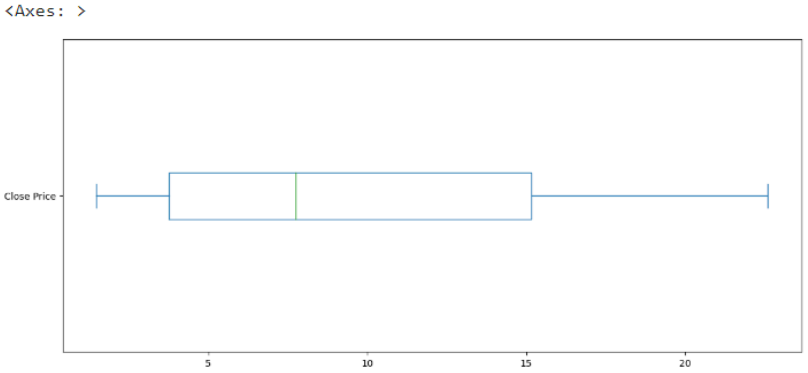


Fig: 1.6

The line graph in Fig: 1.7 for Chariot Oil & Gas Limited's entire closing stock price period is displayed. Prices rose from 2019 until the middle of 2022, at which point they started to decline after that. And The significant years' prices are displayed in Fig. 1.8 as a coloured box with highlights.

```
ax = df.plot(color='orange', figsize=(9,6))
ax.set_xlabel('Years', color = 'blue', fontsize = 14)
ax.set_ylabel('Close Price', color = 'blue', fontsize = 14)
ax.set_title('Chariot Oil & Gas Limited from 2019 to 2023', color='green', fontstyle = 'italic')
plt.show()
```

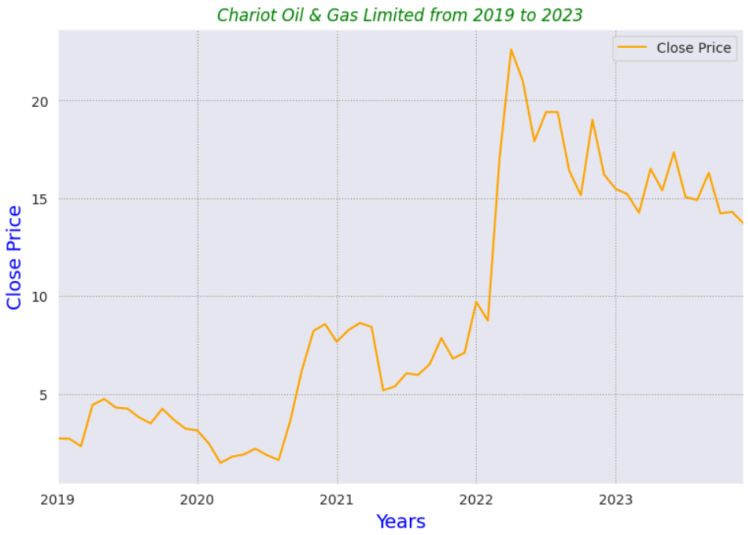


Fig: 1.7

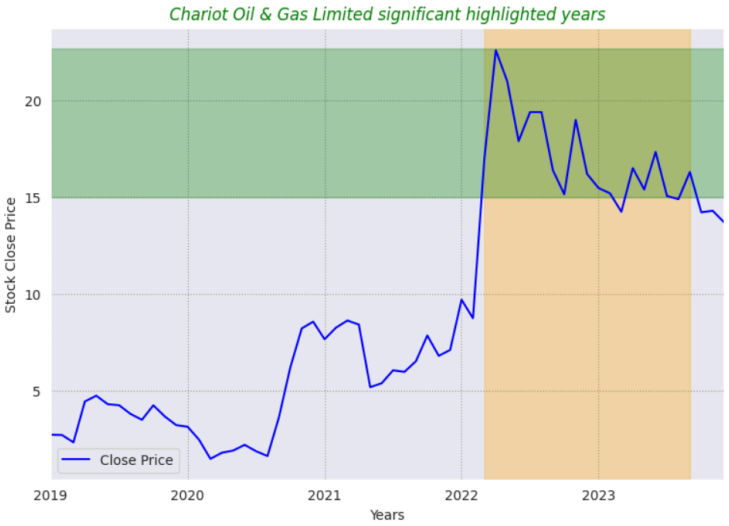


Fig: 1.8

Time Series Decomposition

Time series decomposition is done using seasonal decomposition which is a statistical analysis to classify data into three categories: trend, seasonality, and residual to better understand the pattern of data. (“1.3”)

Additive decomposition is basically the sum of trend, seasonality and residual. Mathematically, It can be represented as
y(t) = T(t) + S(t) + R(t).

```
additive_decomposition = seasonal_decompose(df['Close Price'], model='additive')
fig = additive_decomposition.plot()
fig.set_size_inches((9,6))
fig.tight_layout()
plt.show()
```

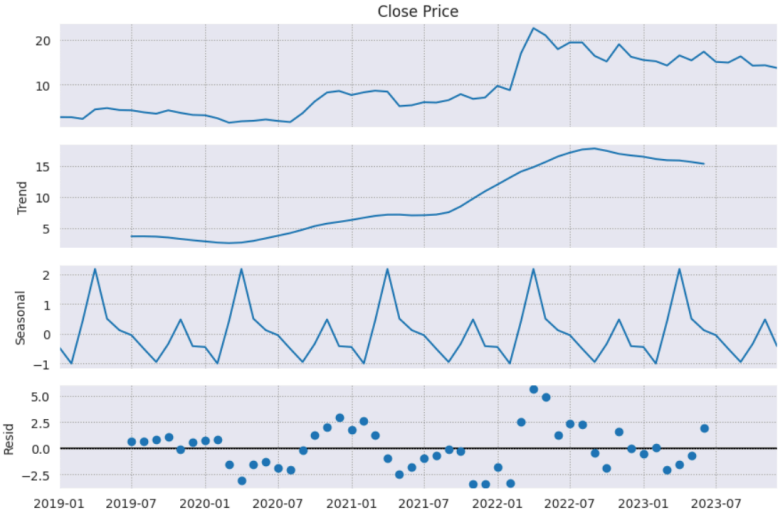


Fig 1.9

As can be seen in Fig. 1.9, the price increased in a nonlinear manner until roughly 2022–08, at which point it began to gradually decline for the trend part. Within the seasonal category, there are yearly variations in prices that occur during specific months or seasons. The price peaked in March or April, after initially falling to its lowest point. It dropped to its lowest point at the end of the year. When it comes to the residual, the majority of them are almost zero. Due to their price, some residuals are unusual (Fig: 1.9).

Multiplicative decomposition is the product of trend, seasonality and residual: y(t) = T(t) × S(t) × R(t). Multiplicative Trend and additive trend are the same. In the seasonal plot, values between 0 and 1 indicate that the observations are below the overall trend, and values greater than 1 indicate that the observations are higher than the overall trend. The majority of the observations for the residual are near to one (Fig 1.10).

```
multiplicative_decomposition = seasonal_decompose(df['Close Price'], model='multiplicative')
fig = multiplicative_decomposition.plot()
fig.set_size_inches((9,6))
fig.tight_layout()
plt.show()
```

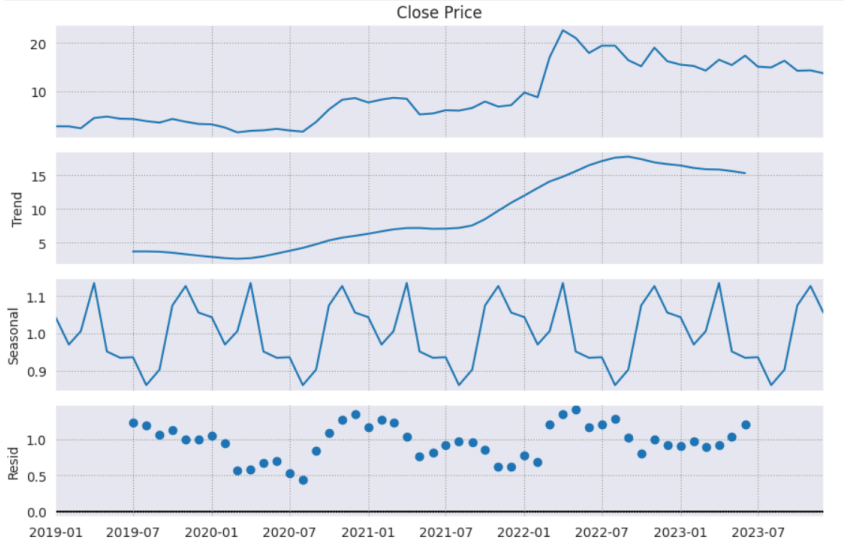


Fig 1.10

Naive Method

I used the naive method to forecast for 6 and 12 months.

Fit, forecast and visualization

Step1: 54 observations are made for train set 1 and the remaining 6 are made for test set 1, as can be seen on the right. With 48 and 12 observations, respectively, the same applied to train set 2 and test set 2.

```
t_len1 = 54
train_set1 = df[0:t_len1]
test_set1 = df[t_len1:]
```

Step 1

Step 2: This is the function for the naive method, which uses the train set as data and the forecasted number of predictions as the nforecast.

Step 3: The fit and forecast is the next step. A naive method function is called, using the train and test set along with the number of forecasts.

```
def naive_method(data, nforecast):
    """
    Forecast using the Naive Method.

    Parameters:
        data (list): A list of historical data points.
        nforecast: number of forecasts you want to make

    Returns:
        forecast (list): A list of forecasts for each period.
    """
    forecast = [data[-1]]  # Initial forecast is the last observed point
    number = nforecast-1 # You can adjust the number of forecast you want to make.

    # To forecast the next 24 points, a loop is necessary:
    for i in range(number):
        forecast.append(forecast[-i])

    return forecast
```

Step 2

```
y_hat_naive1 = test_set1.copy()
y_hat_naive1['naive_forecast'] = naive_method(train_set1['Close Price'], nforecast = 6)
```

Step 3

The 6-month and 12-month naive method forecasts are 17.34 and 16.2, respectively.

```
y_hat_naive1.head()
```

Date	Close Price	naive_forecast
2023-07-01	15.060000	17.34
2023-08-01	14.900000	17.34
2023-09-01	16.299999	17.34
2023-10-01	14.220000	17.34
2023-11-01	14.300000	17.34

```
y_hat_naive2.head()
```

Date	Close Price	naive_forecast
2023-01-01	15.475	16.200001
2023-02-01	15.200	16.200001
2023-03-01	14.250	16.200001
2023-04-01	16.500	16.200001
2023-05-01	15.400	16.200001

Now, The figures are plotted including train, test and naive forecast data.

```
plt.figure(figsize=(20,5))
plt.grid()
plt.plot(train_set1['Close Price'], label='Train Data')
plt.plot(test_set1['Close Price'], label='Test data(6 months)')
plt.plot(y_hat_naive1['naive_forecast'], label='Naive forecast')
plt.legend(loc='best')
plt.title('Naive Method')
plt.show()
```

The first plot is for 6 months and the second one is for 12 months(Step 4).

Step 4

Accuracy Metrics

To find the accuracy of the both models, I calculated root-mean-square-error as rmse, mean absolute percentage error as maps, mean absolute error as mae.

RMSE calculates the mean squared error between the actual 'Close Price' values in test_set1 and the forecasted values in y_hat_naive1['naive_forecast']. MAPE calculates the absolute difference between actual and forecasted values. MAE also calculates the absolute difference between actual and forecasted values.

```
from sklearn.metrics import mean_squared_error
rmse = np.sqrt(mean_squared_error(test_set1['Close Price'], y_hat_naive1['naive_forecast'])).round(2)
mape = np.round(np.mean(np.abs(test_set1['Close Price']-y_hat_naive1['naive_forecast'])/test_set1['Close Price'])*100,2)
mae  = np.round(np.mean(np.abs(test_set1['Close Price']-y_hat_naive1['naive_forecast'])),2)

results = pd.DataFrame({'Method':['Naive method(6 Months)'], 'MAPE': [mape], 'RMSE': [rmse], 'MAE': [mae]})
results = results[['Method', 'RMSE', 'MAE', 'MAPE']]
results
```

We know lower error means better accuracy. For rmse, mae, and mape, the naive method (12 months) yields a lower error than the naive method (6 months), if we compare the two predictions.

	Method	RMSE	MAE	MAPE
0	Naive method(6 Months)	2.72	2.59	17.94

	Method	RMSE	MAE	MAPE
0	Naive method(12 Months)	1.42	1.24	8.41

Comments

I only showed all the code for 6 months forecasting. Between 6 months and 12 months forecasting, 12 months forecasting shows better results. The graph in step 4 shows the visual comparison. RMSE and MAE in the naive methods for 12 months is close to zero which indicates better accuracy and MAPE under 10% is pretty good.

Average Historical method

A technique for projecting future values based on historical averages is known as the "Average Historical Method" in finance.

Fit, forecast and visualization

Step 01: This is the average historical method function.

```
def average_historical_method(data, nforecast):
    """
    Forecast using the average of all historical values Method.

     Parameters:
        data (list): A list of historical data points.
        nforecast: number of forecasts you want to make

    Returns:
        forecast (list): A list of forecasts for each period.

    """
    average = sum(data) / len(data)  # Calculate the average of historical data
    forecast = [average] * nforecast  # Create a list of forecasts with the same average value

    return forecast
```

Step 02: The fit and forecast is done with the average historical method function.

```
y_hat_average_hist1 = test_set1.copy()
y_hat_average_hist1['average_hist_forecast'] = average_historical_method(train_set1['Close Price'], nforecast = 6)
```

```
round(y_hat_average_hist1,2).head()
```

Date	Close Price	average_hist_forecast
2023-07-01	15.06	8.65
2023-08-01	14.90	8.65
2023-09-01	16.30	8.65
2023-10-01	14.22	8.65
2023-11-01	14.30	8.65

```
round(y_hat_average_hist2,2).head()
```

Date	Close Price	average_hist_forecast
2023-01-01	15.48	7.77
2023-02-01	15.20	7.77
2023-03-01	14.25	7.77
2023-04-01	16.50	7.77
2023-05-01	15.40	7.77

Step 03: Now, visualizing the result will give us visual representation.

```
plt.figure(figsize=(20,5))
plt.grid()
plt.plot(train_set1['Close Price'], label='Train data')
plt.plot(test_set1['Close Price'], label='Test data')
plt.plot(y_hat_average_hist1['average_hist_forecast'], label='Average all historical forecast(6 months)')
plt.legend(loc='best')
plt.title('Average all historical Method')
plt.show()
```

Fig 1.11

We can conclude from the model that the predictions are not very accurate.

Accuracy Metrics.

root-mean-square-error, mean absolute percentage error, mean absolute error is calculated to observe the accuracy.

```
from sklearn.metrics import mean_squared_error
rmse = np.sqrt(mean_squared_error(test_set1['Close Price'], y_hat_average_hist1['average_hist_forecast'])).round(2)
mape = np.round(np.mean(np.abs(test_set1['Close Price']-y_hat_average_hist1['average_hist_forecast'])/test_set1['Close Price'])*100,2)
mae  = np.round(np.mean(np.abs(test_set1['Close Price']-y_hat_average_hist1['average_hist_forecast'])),2)

results = pd.DataFrame({'Method':['Average all historical Method'], 'MAPE': [mape], 'RMSE': [rmse], 'MAE': [mae]})
results = results[['Method', 'RMSE', 'MAE', 'MAPE']]
results
```

Both models' accuracy is unsatisfactory.

	Method	RMSE	MAE	MAPE
0	Average all historical Method(6 months)	6.15	6.09	41.14

	Method	RMSE	MAE	MAPE
0	Average all historical Method(12 months)	7.52	7.45	48.7

Comments

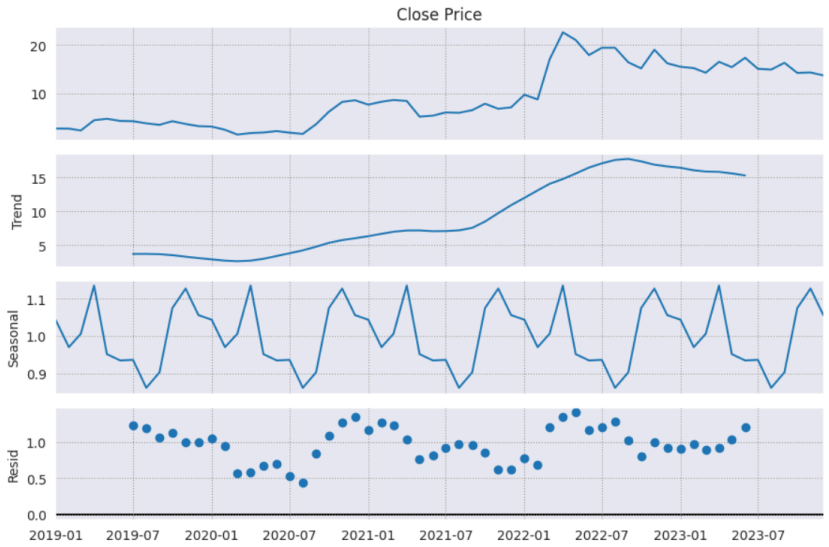
Compared to the Naive method, Average historical method didn't perform well.

Activity 2

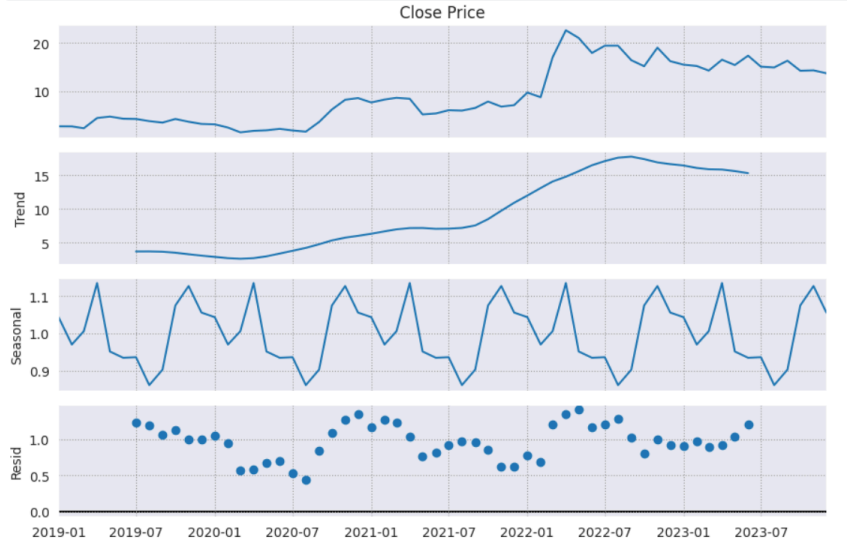
Time Series Decomposition

A method for analyzing a time series into its component parts - trend, seasonality, and residual - is called time series decomposition. Both additive decomposition and multiplicative decomposition is explained in Task 1.

```
additive_decomposition = seasonal_decompose(df['Close Price'], model='additive')
fig = additive_decomposition.plot()
fig.set_size_inches((9,6))
fig.tight_layout()
plt.show()
```



```
multiplicative_decomposition = seasonal_decompose(df['Close Price'], model='multiplicative')
fig = multiplicative_decomposition.plot()
fig.set_size_inches((9,6))
fig.tight_layout()
plt.show()
```



Simple Average Method

A basic statistical method for forecasting, the simple average method (also called the simple mean or arithmetic mean method) involves taking the average of a collection of historical values and using that average to predict values for future periods.

Fit, forecast and visualization

Here ma_window is the rolling window which is essentially a moving subset of data that moves through the time series. The movie window size is set as 12. The last line calculates a moving average forecast using a rolling window over the 'Close Price' column in train_set2 and assigns the last value of the rolling mean to the 'moving_avg_forecast' column in y_hat_MA_meth.

```
y_hat_MA_meth = test_set2.copy()

# MA method
ma_window = 12 # the size of the moving window
y_hat_MA_meth['moving_avg_forecast'] = train_set2['Close Price'].rolling(ma_window).mean().iloc[-1]
```

```
plt.figure(figsize=(20,5))
plt.grid()
plt.plot(train_set2['Close Price'], label='Train data')
plt.plot(test_set2['Close Price'], label='Test data')
plt.plot(y_hat_MA_meth['moving_avg_forecast'], label='Moving Average Forecast')
plt.legend(loc='best')
plt.show()
```

Fig 2.1

Accuracy Metrics.

Again, RMSR, MAPE and MAE are calculated to assess the performance of a simple moving average forecasting model.

```
from sklearn.metrics import mean_squared_error
rmse = np.sqrt(mean_squared_error(test_set2['Close Price'], y_hat_MA_meth['moving_avg_forecast'])).round(2)
mape = np.round(np.mean(np.abs(test_set2['Close Price']-y_hat_MA_meth['moving_avg_forecast'])/test_set2['Close Price'])*100,2)
mae  = np.round(np.mean(np.abs(test_set2['Close Price']-y_hat_MA_meth['moving_avg_forecast'])),2)

results = pd.DataFrame({'Method':['Simple Moving Averages method'], 'MAPE': [mape], 'RMSE': [rmse], 'MAE': [mae]})
results = results[['Method', 'RMSE', 'MAE', 'MAPE']]
results
```

	Method	RMSE	MAE	MAPE
0	Simple Moving Averages method	1.95	1.73	11.8

Comments

An RMSE and MAE close to zero indicate high accuracy. And For MAPE, close to 10% is good.

Exponential Smoothing methods (Single Exponential Smoothing, Holt's Linear, Holt-Winters)

Single Exponential Smoothing ("2.1")

Fit, forecast and visualization

The alpha is the smoothing parameter in simple Exponential Smoothing. A higher alpha gives more weight to recent observations.

SimpleExpSmoothing(np.asarray(train_set2['Close Price'])) initializes a Simple Exponential Smoothing model with the training data. And .fit(smoothing_level=alpha, optimized=False) fits the model to the training data with the alpha. Setting optimized to False means that the optimization algorithm will not be used, and the specified alpha will be used directly

fit2.forecast(len(test_set2)) generates forecasts for the same length as the test set and stores them in the 'SES' column of the y_hat_avg DataFrame.

```
from statsmodels.tsa.api import ExponentialSmoothing, SimpleExpSmoothing, Holt

alpha = 0.6

y_hat_avg = test_set2.copy()

# Function SimpleExpSmoothing represents the method

fit2 = SimpleExpSmoothing(np.asarray(train_set2['Close Price'])).fit(smoothing_level=alpha, optimized=False)
#fit2 = SimpleExpSmoothing(np.asarray(train['Number of Passengers'])).fit() # Here statsmodels to automatically find
                                                                            # an optimized value of alphafor us

y_hat_avg['SES'] = fit2.forecast(len(test_set2)) # number of forecasting equal to the test size

plt.figure(figsize=(16,8))
plt.plot(train_set2['Close Price'], label='Train')
plt.plot(test_set2['Close Price'], label='Test')
plt.plot(y_hat_avg['SES'], label='SES')
plt.legend(loc='best')
plt.show()
```

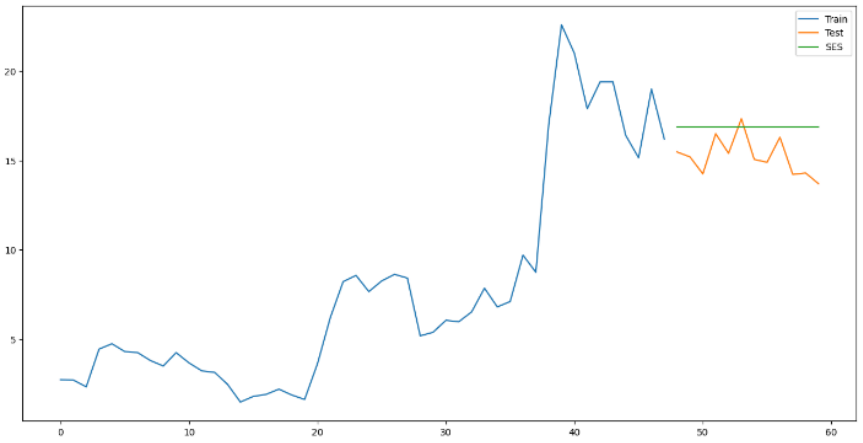


Fig 2.2

Accuracy Metrics

After calculation rmse, mape and mae as like other forecasting, this accuracy metrics is found.

	Method	RMSE	MAE	MAPE
0	Simple Exponential Smoothing	1.93	1.72	11.72

Comments

Single Exponential Smoothing provided good accuracy. Lower values for these metrics indicate better predictive accuracy.

Holt's Linear ("2.2")

Fit, forecast and visualization

Holt(np.asarray(train_set2['Close Price'])) initializes a Holt's Linear Trend model with the training data and .fit(smoothing_level=0.1, smoothing_slope=0.9) fits the model to the training data with the specified smoothing levels.

fit1.forecast(len(test_set2)) generates forecasts for the same length as the test set and stores them in the 'Holt_linear' column of the y_hat_avg DataFrame.

```
from statsmodels.tsa.api import Holt

y_hat_avg = test_set2.copy()

fit1 = Holt(np.asarray(train_set2['Close Price'])).fit(smoothing_level = 0.1,smoothing_slope = 0.9)
#fit1 = Holt(np.asarray(train['Number of Passengers'])).fit() # statsmodels to automatically optimized the values

y_hat_avg['Holt_linear'] = fit1.forecast(len(test_set2))

plt.figure(figsize=(16,8))
plt.plot(train_set2['Close Price'], label='Train')
plt.plot(test_set2['Close Price'], label='Test')
plt.plot(y_hat_avg['Holt_linear'], label='Holt_linear')
plt.legend(loc='best')
plt.show()
```

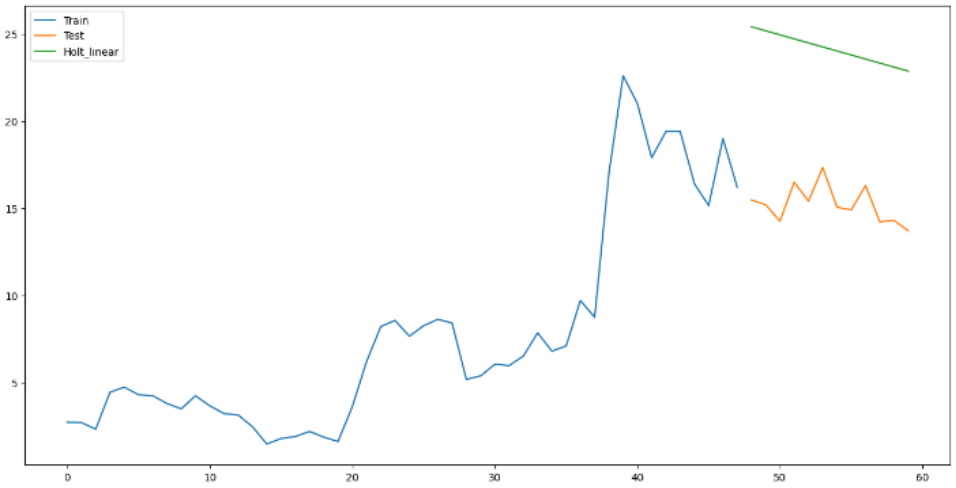


Fig 2.3

Accuracy Metrics

RMSE and MAE are not closed to 0 and MAPE is far from 10%.

	Method	RMSE	MAE	MAPE
0	Holt Linear	8.98	8.92	59.15

Comments

The accuracy result is not suitable. The error is relatively high.

Holt-Winters ("2.3")

Fit, forecast and visualization

ExponentialSmoothing(np.asarray(train_set2['Close Price']), seasonal_periods=12, trend='add', seasonal='add') initializes an Exponential Smoothing model with Holt-Winters seasonal decomposition using the training data and .fit() fits the model to the training data with the specified settings.

```
from statsmodels.tsa.api import ExponentialSmoothing
y_hat_avg = test_set2.copy()
fit1 = ExponentialSmoothing(np.asarray(train_set2['Close Price']) ,seasonal_periods=12 ,trend='add', seasonal='add',).fit()


y_hat_avg['Holt_Winter'] = fit1.forecast(len(test_set2))
plt.figure(figsize=(16,8))
plt.plot( train_set2['Close Price'], label='Train')
plt.plot(test_set2['Close Price'], label='Test')
plt.plot(y_hat_avg['Holt_Winter'], label='Holt_Winter')
plt.legend(loc='best')
plt.show()
```

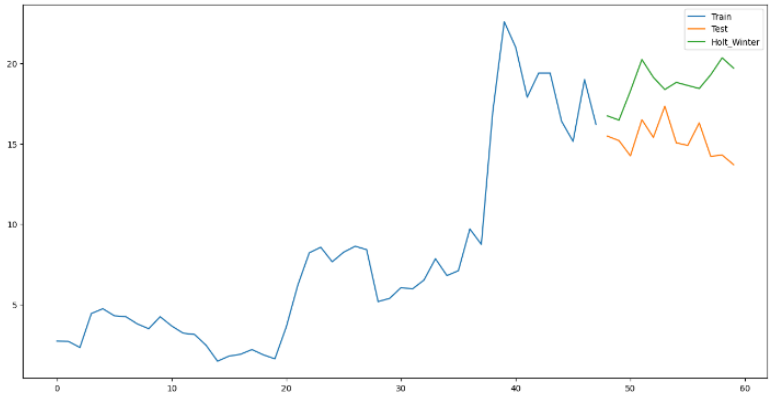


Fig 2.4

Accuracy Metrics

	Method	RMSE	MAE	MAPE
0	Holt Winter	3.87	3.49	23.57

Comments

The accuracy is better than holt linear trend method but still not acceptable.

Activity 3

Time Series Stationary test and Differencing

I Checked Augmented Dickey Fuller test(ADF Test) from the statsmodels package and got a p-value which is more than 0.05(“3.1”). That means it is not stationary.

```
from statsmodels.tsa.stattools import adfuller
from numpy import log
result = adfuller(df)
print('ADF Statistic: %f' % result[0])
print('p-value: %f' % result[1])
```

```
ADF Statistic: -1.320999
p-value: 0.619511
```

Let’s differentiate the series and find out stationary from the autocorrelation plot.

In 1st order differencing, the time series pattern changed. The time series reached stationarity in 1st order (Fig 3.1). So differencing, d is 1.

```
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
import matplotlib.pyplot as plt
plt.rcParams.update({'figure.figsize':(9,7), 'figure.dpi':120})


# Original Series
fig, axes = plt.subplots(3, 2, sharex='col')
axes[0, 0].plot(df['Close Price']); axes[0, 0].set_title('Original Series')
plot_acf(df['Close Price'], lags = 20, ax=axes[0, 1])

# 1st Differencing
axes[1, 0].plot(df['Close Price'].diff()); axes[1, 0].set_title('1st Order Differencing')
plot_acf(df['Close Price'].diff().dropna(), lags = 20, ax=axes[1, 1])

# 2nd Differencing
axes[2, 0].plot(df['Close Price'].diff().diff()); axes[2, 0].set_title('2nd Order Differencing')
plot_acf(df['Close Price'].diff().diff().dropna(), lags = 20, ax=axes[2, 1])

plt.show()
```

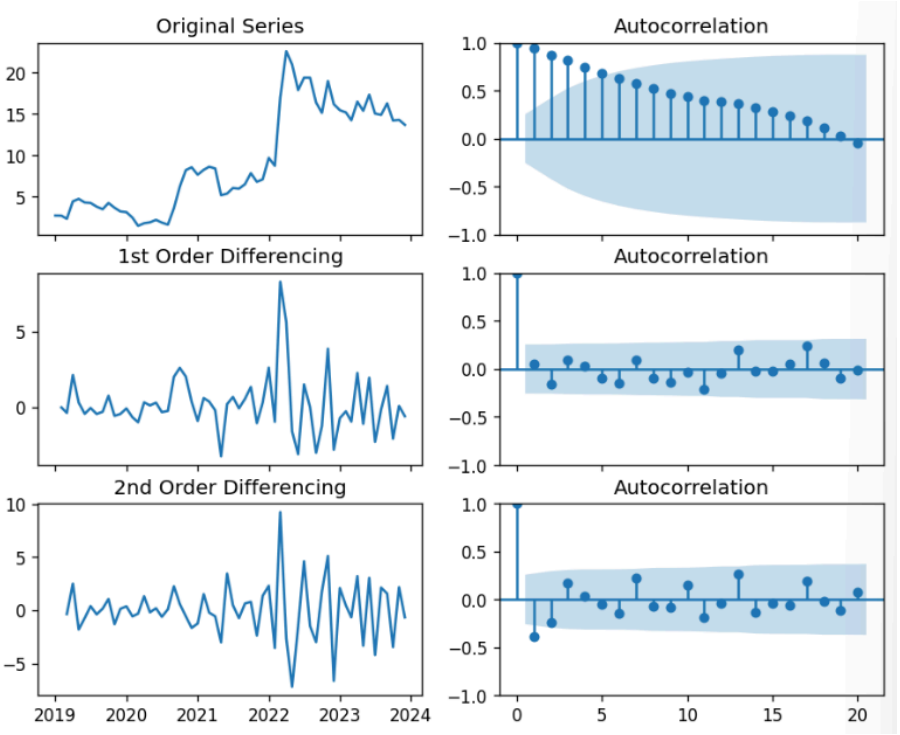


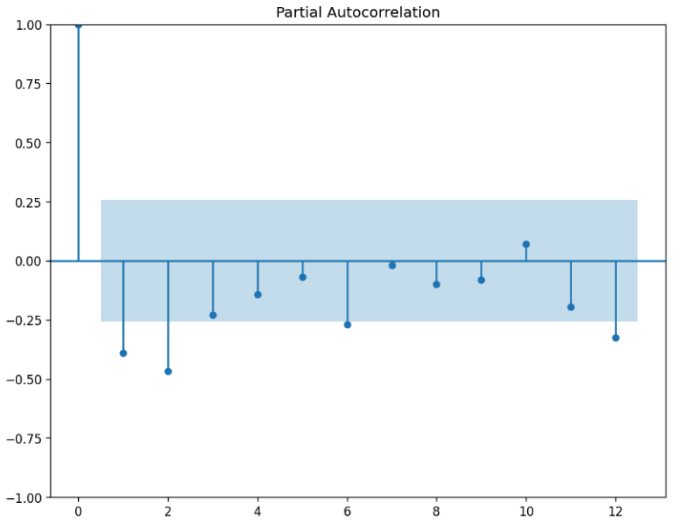
Fig 3.1

ACF and PACF

As 1st differencing PACF and ACF doesn't show any significance lag, AR model lag and q, MA model lag can be found from the 2nd differencing PACF and ACF.

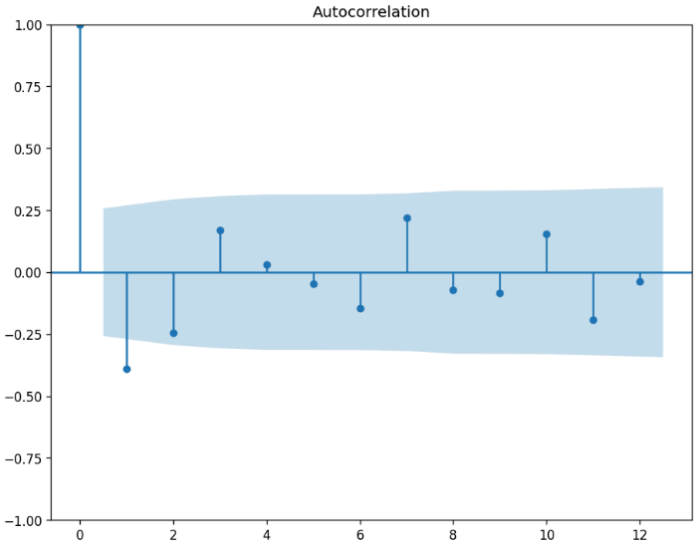
1st lag and 2nd lag are significant. After 2nd lag, there is a quick shutdown in the PACF ("3.2"). So p can be taken as 2.

```
plot_pacf(df['Close_Price'].diff().diff().dropna(), lags=12)

plt.show()
```



there is only one significant lag in the ACF ("3.3"). So, we can take q as 1.

```
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf

plot_acf(df['Close_Price'].diff().diff().dropna(), lags = 12)
plt.show()
```



ARIMA (“3.4”)

Fit, forecast and visualization

The following code fits an ARIMA(2,1,1) model to the time series and provides a summary of the model. AIC and BIC are lower than other models that were checked.

```
from statsmodels.tsa.arima.model import ARIMA
#(p,d,q) (2,1,1) ARIMA Model
model = ARIMA(df['Close_Price'], order=(2,1,1))
model_fit = model.fit()
print(model_fit.summary())
```

```
                               SARIMAX Results
==============================================================================
Dep. Variable:            Close_Price   No. Observations:                   60
Model:                 ARIMA(2, 2, 1)   Log Likelihood                -121.577
Date:                Sun, 04 Feb 2024   AIC                            251.154
Time:                        02:32:04   BIC                            259.396
Sample:                    01-01-2019   HQIC                           254.364
                         - 12-01-2023
Covariance Type:                  opg
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
ar.L1          0.0743      0.117      0.635      0.525      -0.155       0.303
ar.L2         -0.1410      0.152     -0.930      0.352      -0.438       0.156
ma.L1         -0.9998     18.165     -0.055      0.956     -36.603      34.603
sigma2         3.6020     65.197      0.055      0.956    -124.182     131.386
===================================================================================
Ljung-Box (L1) (Q):                   0.00   Jarque-Bera (JB):
87.91
Prob(Q):                              0.97   Prob(JB):
0.00
Heteroskedasticity (H):               6.27   Skew:
1.48
Prob(H) (two-sided):                  0.00   Kurtosis:
8.25
===================================================================================
```

The residual errors look good with near zero mean (Fig 3.2). Both line plot and KDE plot provide a visual representation of how well the model is capturing the patterns in the data.

```
residuals = pd.DataFrame(model_fit.resid)
fig, ax = plt.subplots(1,2)
residuals.plot(title="Residuals", ax=ax[0])
residuals.plot(kind='kde', title='Density', ax=ax[1])
plt.show()
```

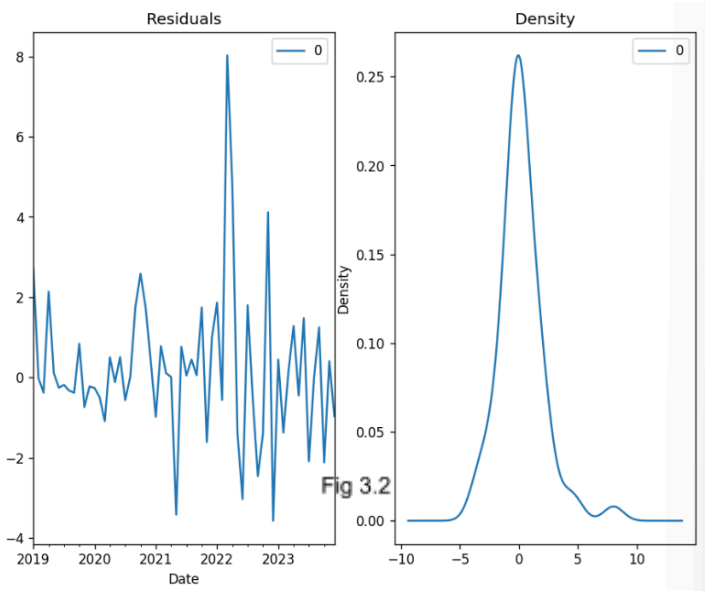


Fig 3.2

Now, we have to forecast using the ARIMA model on the Training dataset, and visualize it. I fitted an ARIMA(2,1,1) to the training data and forecast for selected periods ahead and converted everything into pandas series objects which are easy to plot against the actual data.

```
#Build Model
model = ARIMA(train, order=(2,1,1))
fitted = model.fit()


# Forecast
fc = fitted.forecast(num_obs_ahead, alpha=0.05)  # 95% conf
conf_ins = fitted.get_forecast(num_obs_ahead).summary_frame()

# Make as pandas series
fc_series = pd.Series(fc, index=test.index)
lower_series = pd.Series(conf_ins['mean_ci_lower'], index=test.index)
upper_series = pd.Series(conf_ins['mean_ci_upper'], index=test.index)

# Plot
plt.figure(figsize=(12,5), dpi=100)
plt.plot(train, label='training')
plt.plot(test, label='actual')
plt.plot(fc_series, label='forecast')
plt.fill_between(lower_series.index, lower_series, upper_series,
                 color='k', alpha=.15)
plt.title('Forecast vs Actuals')
plt.legend(loc='upper left', fontsize=8)
plt.show()
```

From the graph, it looks like the prediction is pretty good for the ARIMA(2,1,1) (Fig 3.3).

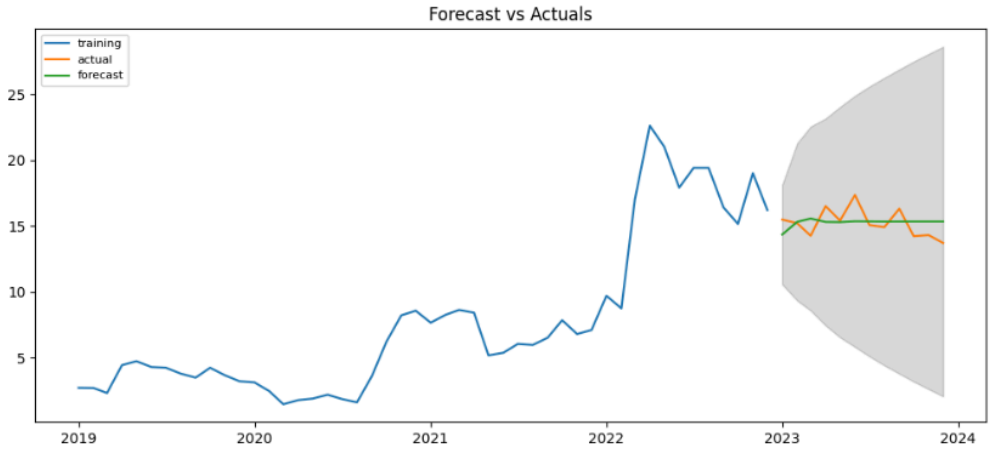


Fig 3.3

To check reliability of the ARIMA model, diagnostics plots are runned (Fig 3.4). These plots are useful for assessing the goodness of fit and identifying potential issues with the model. The residuals of the ARIMA have uniform variance around zero. Histogram and KDE show its normal distribution. In the correlogram, all the lag is inside the significance line. These plots show it's a good fit.

```
model_fit.plot_diagnostics(figsize=(10,8))
plt.show()
```

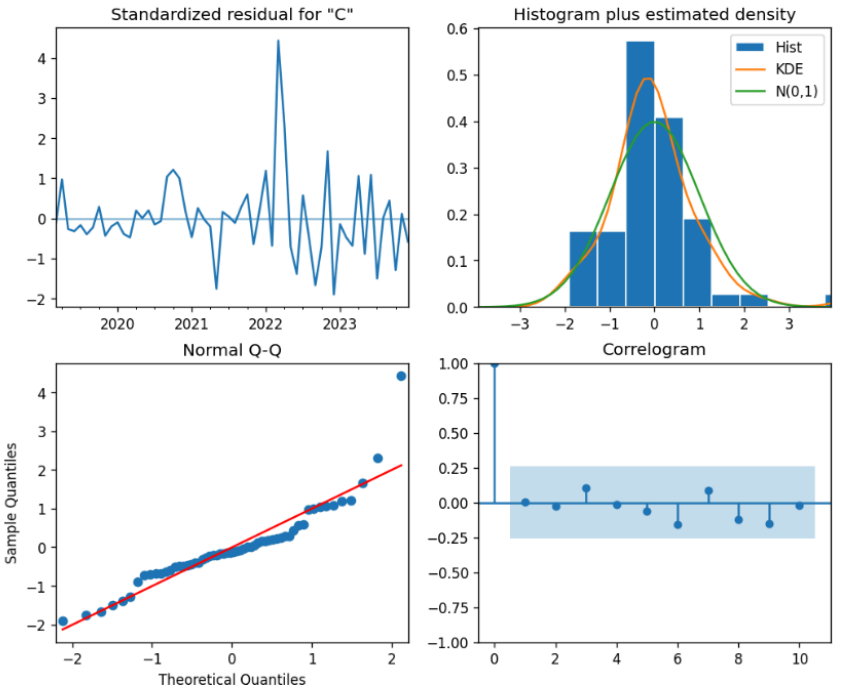


Fig 3.4

Accuracy Metrics.

We will just consider 3 accuracy metrics:

1. MAPE
2. Correlation
3. Min-Max Error

```
# Accuracy metrics
def forecast_accuracy(forecast, actual):
    mape = np.mean(np.abs(forecast - actual)/np.abs(actual))  # MAPE
    corr = np.corrcoef(forecast, actual)[0,1]   # corr
    mins = np.amin(np.hstack([forecast[:,None],
                              actual[:,None]]), axis=1)
    maxs = np.amax(np.hstack([forecast[:,None],
                              actual[:,None]]), axis=1)
    minmax = 1 - np.mean(mins/maxs)             # minmax
    return({'mape':mape,
            'corr':corr, 'minmax':minmax})

forecast_accuracy(fc.values, test.values)
```

```
'mape': 0.061886253141399374,
'corr': -0.14060792085157928,
'minmax': 0.059221819759787575
```

Comments

We got a MAPE of 6.19% which is pretty good. The correlation between the forecasted and actual values is approximately -0.1406 which means negative correlation, but relatively low. A 5.92% min-max accuracy suggests a reasonably good performance. The result indicates the model to be a good fit.

SARIMA (“3.5”)

Fit, forecast and visualization

p=2, d=1, q=1 is taken from ARIMA and P=2, D=1, and Q=1 and periodicity = 12 is taken for seasonal order.

```
import statsmodels.api as sm
# Fit the SARIMA model
modsar = sm.tsa.statespace.SARIMAX(df['Close_Price'], order=(2,1,1),
seasonal_order=(2,1,1,12))
modsar_fit = modsar.fit()
print(modsar_fit.summary())
```

```
                                     SARIMAX Results
==========================================================================================
Dep. Variable:                      Close_Price   No. Observations:                   60
Model:             SARIMAX(2, 1, 1)x(2, 1, 1, 12)   Log Likelihood                -105.925
Date:                            Sun, 04 Feb 2024   AIC                            225.850
Time:                                    03:14:05   BIC                            238.801
Sample:                                01-01-2019   HQIC                           230.724
                                     - 12-01-2023
Covariance Type:                              opg
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
ar.L1         -0.4691      0.591     -0.794      0.427      -1.628       0.689
ar.L2         -0.1079      0.286     -0.377      0.706      -0.669       0.453
ma.L1          0.5707      0.600      0.952      0.341      -0.605       1.746
ar.S.L12      -0.1536      2.075     -0.074      0.941      -4.221       3.914
ar.S.L24      -0.1443      1.206     -0.120      0.905      -2.507       2.219
ma.S.L12      -0.9957    305.805     -0.003      0.997    -600.362     598.370
sigma2         3.1751    964.924      0.003      0.997   -1888.042    1894.392
===================================================================================
Ljung-Box (L1) (Q):                   0.00   Jarque-Bera (JB):                60.83
Prob(Q):                              0.98   Prob(JB):                         0.00
Heteroskedasticity (H):               3.08   Skew:                             1.42
Prob(H) (two-sided):                  0.03   Kurtosis:                         7.80
===================================================================================
```

P value is not less than 0.05. However AIC and BIC are lower.

The residual is close to 0 which is good (Fig 3.4).

```
# Plot residual errors
residuals_sar = pd.DataFrame(modsar_fit.resid)
fig, ax = plt.subplots(1,2)
residuals_sar.plot(title="Residuals", ax=ax[0])
residuals_sar.plot(kind='kde', title='Density', ax=ax[1])
plt.show()
```

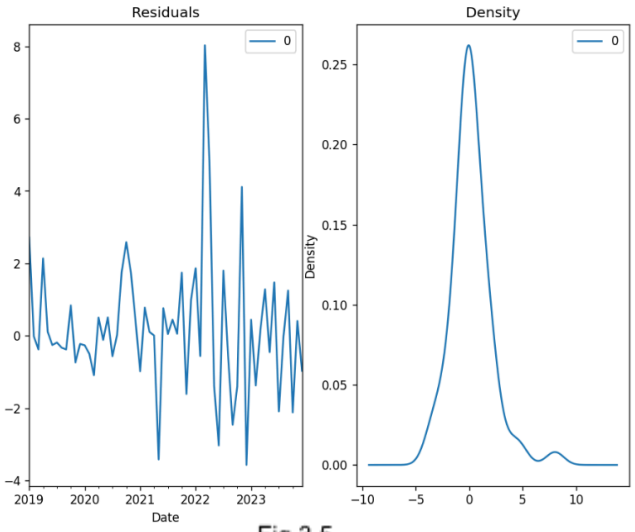


Fig 3.5

SARIMA(2,1,1)(2,1,1,12) is fitted to the training data and forecast for selected periods ahead and converted everything into pandas series objects which are easy to plot against the actual data.

```
# Build the SARIMA model
modsar = ARIMA(train, order=(2, 1, 1), seasonal_order=(2,1,1,12))
fitted_sar = modsar.fit()


# Forecast
fc_sar = fitted_sar.forecast(num_obs_ahead, alpha=0.05)  # 95% conf
conf_ins = fitted_sar.get_forecast(num_obs_ahead).summary_frame()

# Make as pandas series
fc_series = pd.Series(fc_sar, index=test.index)
lower_series = pd.Series(conf_ins['mean_ci_lower'], index=test.index)
upper_series = pd.Series(conf_ins['mean_ci_upper'], index=test.index)

# Plot
plt.figure(figsize=(12,5), dpi=100)
plt.plot(train, label='training')
plt.plot(test, label='actual')
plt.plot(fc_series, label='forecast')
plt.fill_between(lower_series.index, lower_series, upper_series,
                 color='k', alpha=.15)
plt.title('Forecast vs Actuals')
plt.legend(loc='upper left', fontsize=8)
plt.show()
```

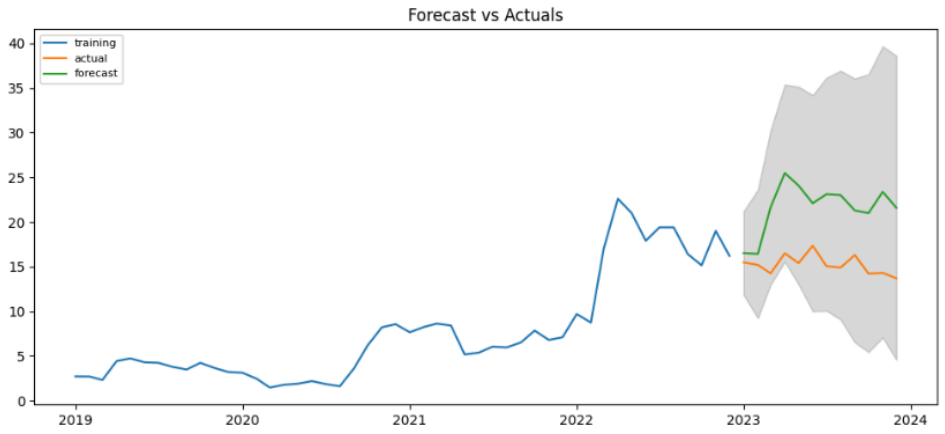


Fig 3.6

```
modsar_fit.plot_diagnostics(figsize=(10,8))
plt.show()
```

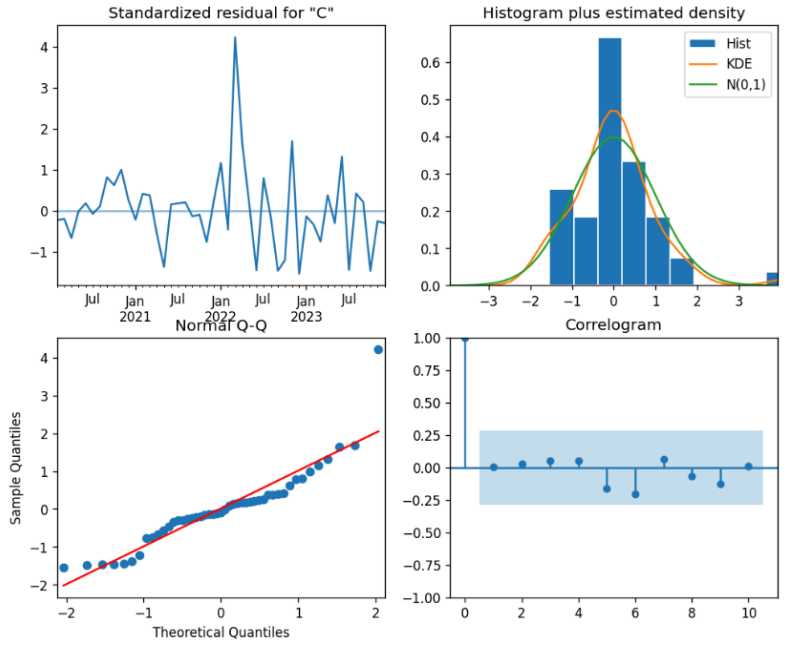


Fig 3.7

There is uniform variance around zero in the SARIMA residuals. Its normal distribution is shown by the histogram and KDE. Every lag point in the correlogram falls inside the significance line (Fig 3.7).

Accuracy Metrics.

forecast_accuracy(fc_sar.values, test.values) gives the accuracy result.

'mape': 0.4267070858238164,
'corr': 0.09914011924636046,
'minmax': 0.28492591671602086

Comments

A MAPE of 42.68% is relatively high, suggesting that the model's predictions have a considerable percentage error. A correlation of 0.099 suggests a weak positive correlation. A 28.49% min-max accuracy suggests that the model's predictions are relatively close to the actual values in terms of their normalized difference. So, it's not a good fit.

Conclusion

Seasonal Decompose illustrates the information about trend, seasonality and resid which helps to understand data patterns easily. The ARIMA model gives good accuracy whether SARIMA is not a good fit. The naive method also performs better than other methods. Among the 3 Exponential smoothing methods, the Single exponential method performs better.

References and Bibliography

"0.1." *Chariot Gas & Oil Limited*, https://uk.finance.yahoo.com/quote/CHAR.L/history?p=CHAR.L.

"1.1." *pandas.to_datetime*, https://pandas.pydata.org/docs/reference/api/pandas.to_datetime.html.

"1.2." *Distplot*, https://seaborn.pydata.org/generated/seaborn.distplot.html.

"1.3." *Seasonal Decompose*, https://www.statsmodels.org/dev/generated/statsmodels.tsa.seasonal.seasonal_decompose.html.

"2.1." *Single Exponential Smoothing*, https://www.statsmodels.org/devel/generated/statsmodels.tsa.holtwinters.SimpleExpSmoothing.html.

"2.2." *Holt Linear*, https://www.statsmodels.org/devel/generated/statsmodels.tsa.holtwinters.Holt.html.

"2.3." *Exponential Smoothing*, https://www.statsmodels.org/dev/generated/statsmodels.tsa.holtwinters.ExponentialSmoothing.html.

"3.1." *Augmented Dickey Fuller Test*, https://www.statsmodels.org/dev/generated/statsmodels.tsa.stattools.adfuller.html.

"3.2." *Autocorrelation Function*, https://www.statsmodels.org/stable/generated/statsmodels.graphics.tsaplots.plot_acf.html.

"3.3." *Partial Autocorrelation Function*, https://www.statsmodels.org/stable/generated/statsmodels.graphics.tsaplots.plot_pacf.html.

"3.4." *ARIMA*, https://www.statsmodels.org/stable/generated/statsmodels.tsa.arima.model.ARIMA.html.

"3.5." *SARIMA*, https://www.statsmodels.org/dev/generated/statsmodels.tsa.statespace.sarimax.SARIMAX.html.

Other Resources

1. Code Workbook, https://www.kaggle.com/code/iftekhi/bdapplication-assessment-01

Appendix (if necessary)