TIME SERIES ANALYSIS STOCK MARKET

INTRODUCTION

Time series analysis is a powerful statistical tool used to analyze time-ordered data points. In this project, we leverage historical stock price data of Tata Steel to build an ARIMA (AutoRegressive Integrated Moving Average) model. ARIMA is a popular method used in time series forecasting due to its ability to model and predict based on past data points while accounting for different components such as trends and seasonality.

WHAT IS TIME SERIES ?

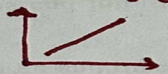
Time Series

→ Data points recorded at specific time intervals to understand underlying patterns.

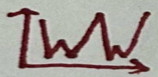
(1) Components → Trends (Overall direction of the data over time) ↑ increasing ↓ decreasing

graph

flat | going up | going down

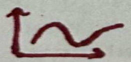


→ Seasonality (Repeating pattern of data over a set period of time).



spikes

→ cycle (Repeating but non-seasonal Patterns in the data.)



→ Variation (Unpredictable ups and downs in the data that cannot be explained by these other components)

ARIMA MODEL

FORECASTING MODEL → ARIMA

AR → Auto Regressive Component
↳ How past values affect future Values

I → Integrated (accounts for trends and Seasonality)

MA → Moving Average component
(Remove non-deterministic or Random movement from time Series)

Exponential Smoothing ?

forecast time series data that doesn't have a clean trend or seasonality.

LEARING IN THIS PROJECT

- CHOOSE YOUR STOCK DATA
- DATA VISUALIZATION
- DATA CLEANING
- TIME SERIES PLOT
- ACF AND PACF PLOT
- STATIONARITY TEST (ADF
- DECOMPOSITION
- ARIMA MODELING
- FORECASTING WITH ARIMA
- SARIMA MODELING
- FORECAST VS ACTUAL

START WITH THE CODE

IMPORT LIBRARIES

```
import pandas as pd
import seaborn as sns
import numpy as np
import matplotlib.pyplot as plt
from statsmodels.tsa.arima.model import ARIMA
from statsmodels.tsa.statespace.sarimax import SARIMAX
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
from statsmodels.tsa.stattools import adfuller
from sklearn.metrics import mean_squared_error, mean_absolute_error
import statsmodels.api as sm
from statsmodels.tsa.seasonal import seasonal_decompose
import yfinance as yf
import warnings
warnings.filterwarnings("ignore")
```

- **Pandas** is used for data manipulation and analysis.
- **Seaborn** is a visualization library based on matplotlib.
- **Matplotlib** is a plotting library.
- Plot the **Autocorrelation Function** (ACF) and **Partial Autocorrelation Function** (PACF) of a time series.
- **ADFuller** is the Augmented Dickey-Fuller test, used to test the stationarity of a time series.
- Performance metrics used to evaluate the accuracy of the forecast models.
- Satistical models.
- Decompose a time series.
- yFinance is a library that allows for easy downloading of stock market data.
- Use to ignore warnings.

DOWNLOADING THE DATA AND SAVING IT AS A CSV FILE IN THE GOOGLE COLAB NOTEBOOK

```
[27] # Download historical stock data for Tata Steel from Yahoo Finance
     data = yf.download('TATASTEEL.NS', start='2020-01-01', end='2021-01-01')

     # Save the data to a CSV file
     data.to_csv('tatasteel_stock.csv')
```

LOADING THE DATA

```
#Read the csv file
df = pd.read_csv('/content/tatasteel_stock.csv')
# Display the first few rows of the DataFrame
print(df.head())
```

```
         Date       Open       High        Low      Close  Adj Close  \
0  2020-01-01  47.299999  47.650002  46.480000  46.775002  40.974754
1  2020-01-02  47.200001  48.779999  47.200001  48.485001  42.472713
2  2020-01-03  48.299999  48.619999  47.945000  48.369999  42.371967
3  2020-01-06  48.000000  48.000000  47.055000  47.325001  41.456554
4  2020-01-07  47.549999  48.459999  47.355000  47.610001  41.706207

      Volume
0  121005300
1  216749610
2  129568630
3   96016080
4  131957880
```

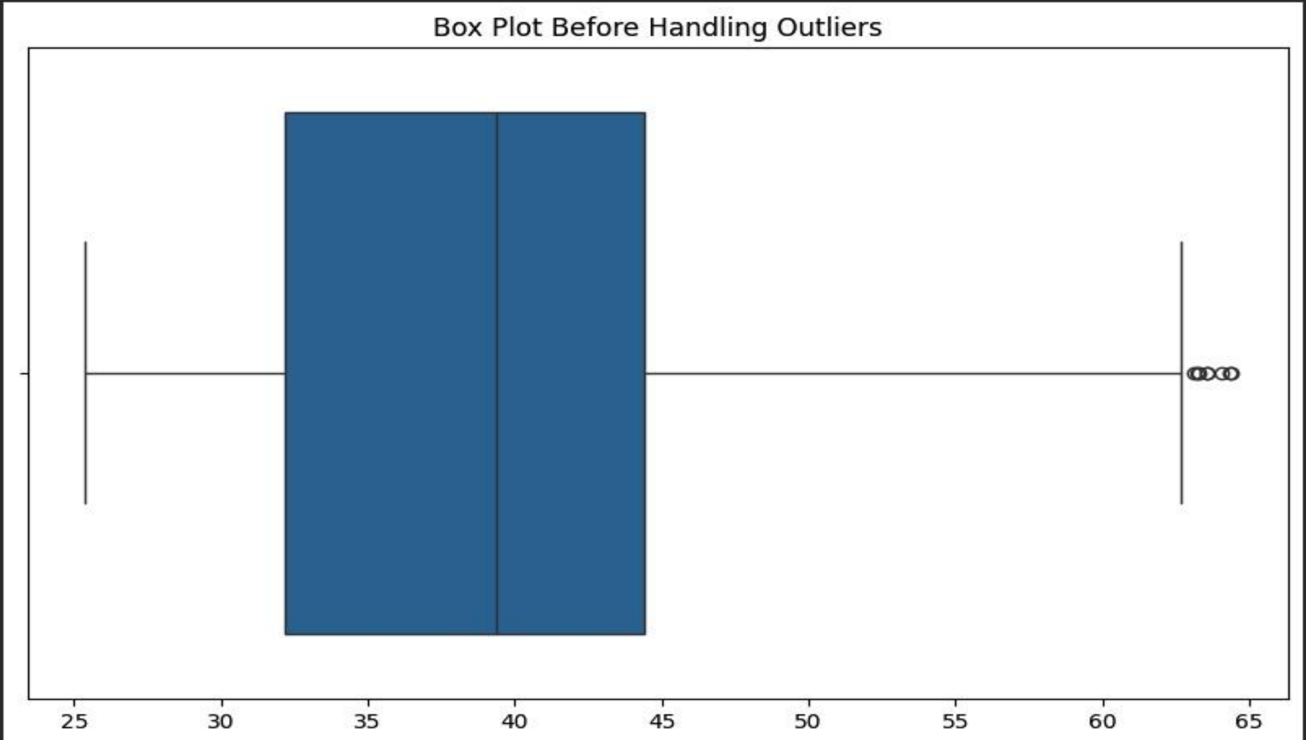
```
[30] #fill the missing values with forward fill
     if df.isnull().sum().sum() > 0:
       df = df.fillna(method='ffill')
```

OUTLIER HANDING

- Detect the outlier.
- Visualize the outlier using BOX PLOT.
- Handle outlier using IQR.

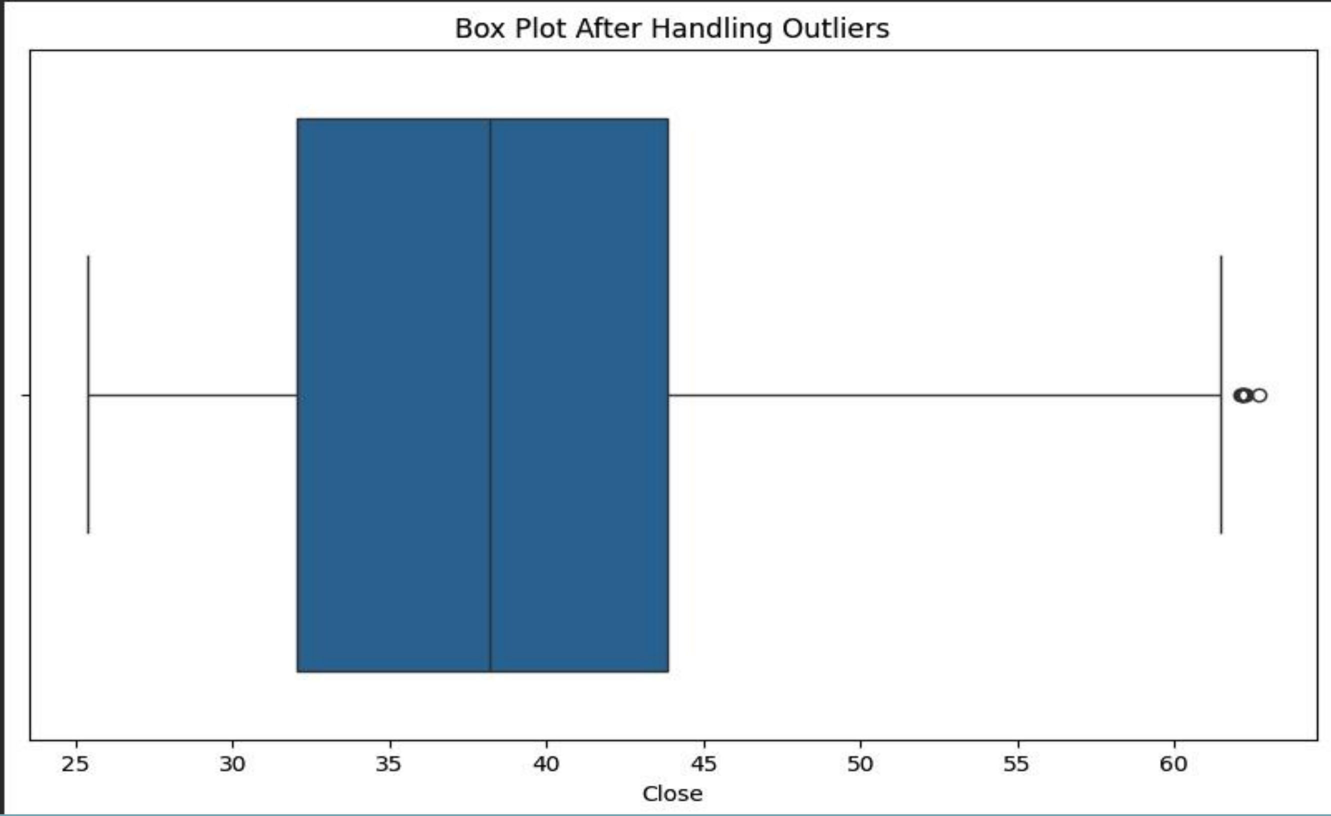
OUTLIERS IDENTIFICATION

```
#Outlier detection
plt.figure(figsize=(10, 6))
sns.boxplot(x=df['Close'])
plt.title('Box Plot Before Handling Outliers')
plt.show()
```



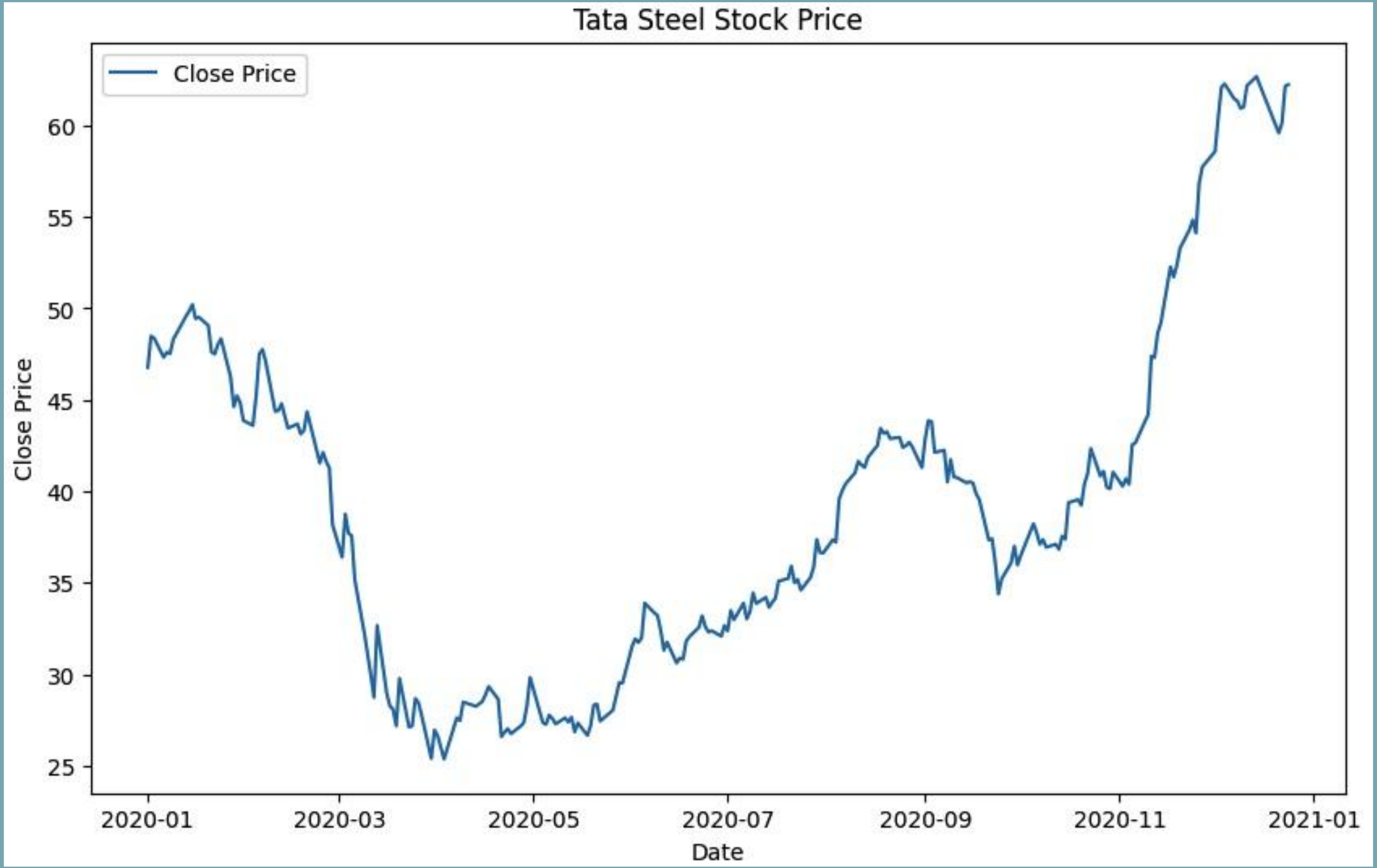
```python
#Calculsting the quartiles
Q1 = df['Close'].quantile(0.25)
Q3 = df['Close'].quantile(0.75)
IQR = Q3 - Q1 #Calculating Interquartile Range
#Calculating upper and lower bound
lower_bound = Q1 - 1.5 * IQR
upper_bound = Q3 + 1.5 * IQR
#Filtering out the outliers
df = df[(df['Close'] >= lower_bound) & (df['Close'] <= upper_bound)]
```

```
plt.figure(figsize=(10, 6))
sns.boxplot(x=df['Close'])
plt.title('Box Plot After Handling Outliers')
plt.show()
```



PLOTING THE TIME SERIES GRAPH

```
[40] # DATE is containing date and converting it to datetime
     df['Date'] = pd.to_datetime(df['Date'])
     df.set_index('Date', inplace=True)

     # Plot the 'Close' price against the date
     plt.figure(figsize=(10, 6))
     plt.plot(df.index, df['Close'], label='Close Price')
     plt.title('Tata Steel Stock Price')
     plt.xlabel('Date')
     plt.ylabel('Close Price')
     plt.legend()
     plt.show()
```



Tata Steel Stock Price
Close Price
60
55
50
45
40
35
30
25
Close Price
2020-01
2020-03
2020-05
2020-07
2020-09
2020-11
2021-01
Date

- Stationarity means that the statistical properties of the time series, such as mean, variance, and autocorrelation, are constant over time.
- Stationary data simplifies the modeling process. Non-stationary data can exhibit trends, seasonality, and other patterns that complicate the model.
- In non-stationary data, the probability distributions can change over time, making statistical inference less reliable.

```
[41] #check the stationarity
     def adf_test(series):
         result = adfuller(series)
         print('ADF Statistic: %f' % result[0])
         print('p-value: %f' % result[1])
         for key, value in result[4].items():
             print('\t%s: %.3f' % (key, value))

     adf_test(df['Close'])
```

```
ADF Statistic: -1.313067
p-value: 0.623173
        1%: -3.459
        5%: -2.874
        10%: -2.573
```

Double-click (or enter) to edit

Since the p-value is greater than 0.05 the time series is non-stationary.

```
[42] # If the data is not stationary, take the first difference
     data_diff = df['Close'].diff().dropna()
     adf_test(data_diff)
```

```
ADF Statistic: -3.065495
p-value: 0.029214
        1%: -3.459
        5%: -2.874
        10%: -2.573
```

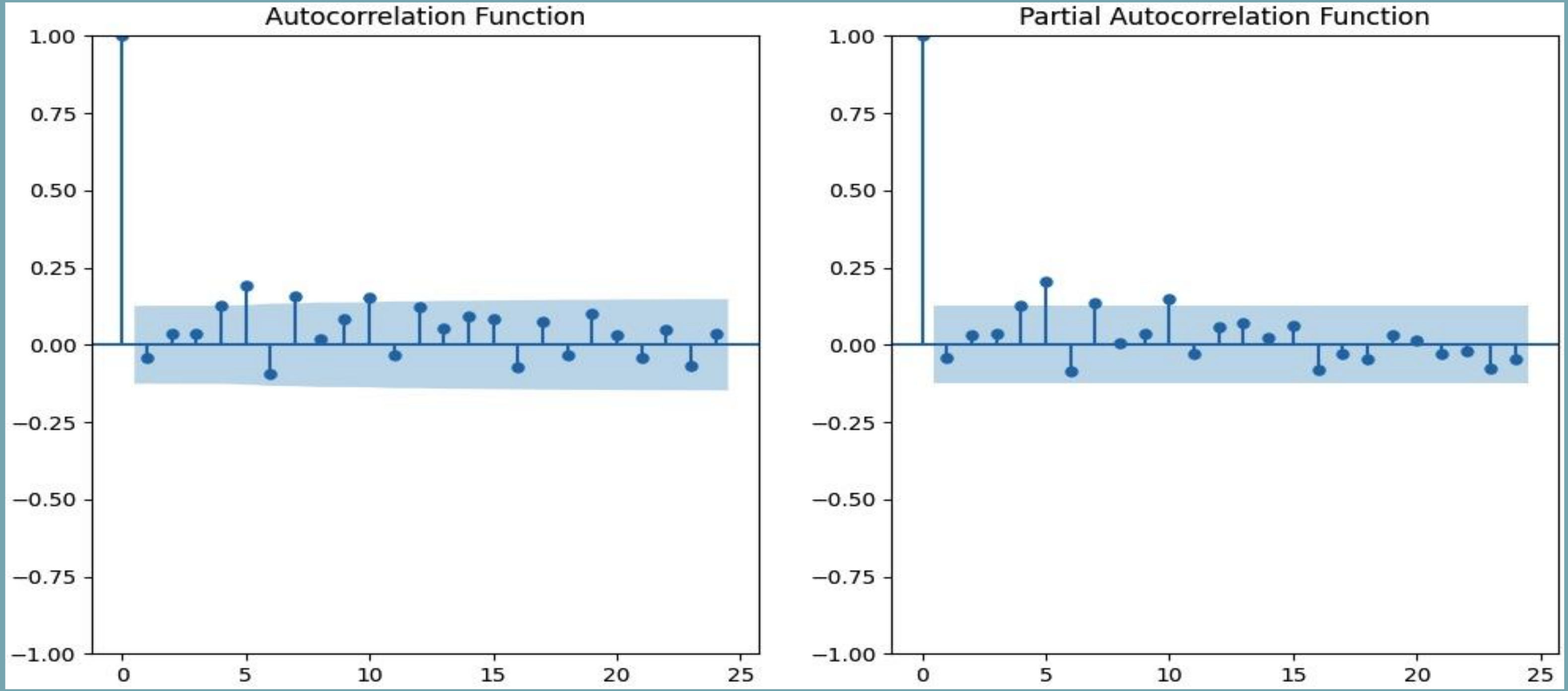
Autocorrelation Function (ACF)

- Quantifies the relationship between the values of the series at different points in time.
- The ACF helps to identify the extent of correlation between the current value and past values of the time series.

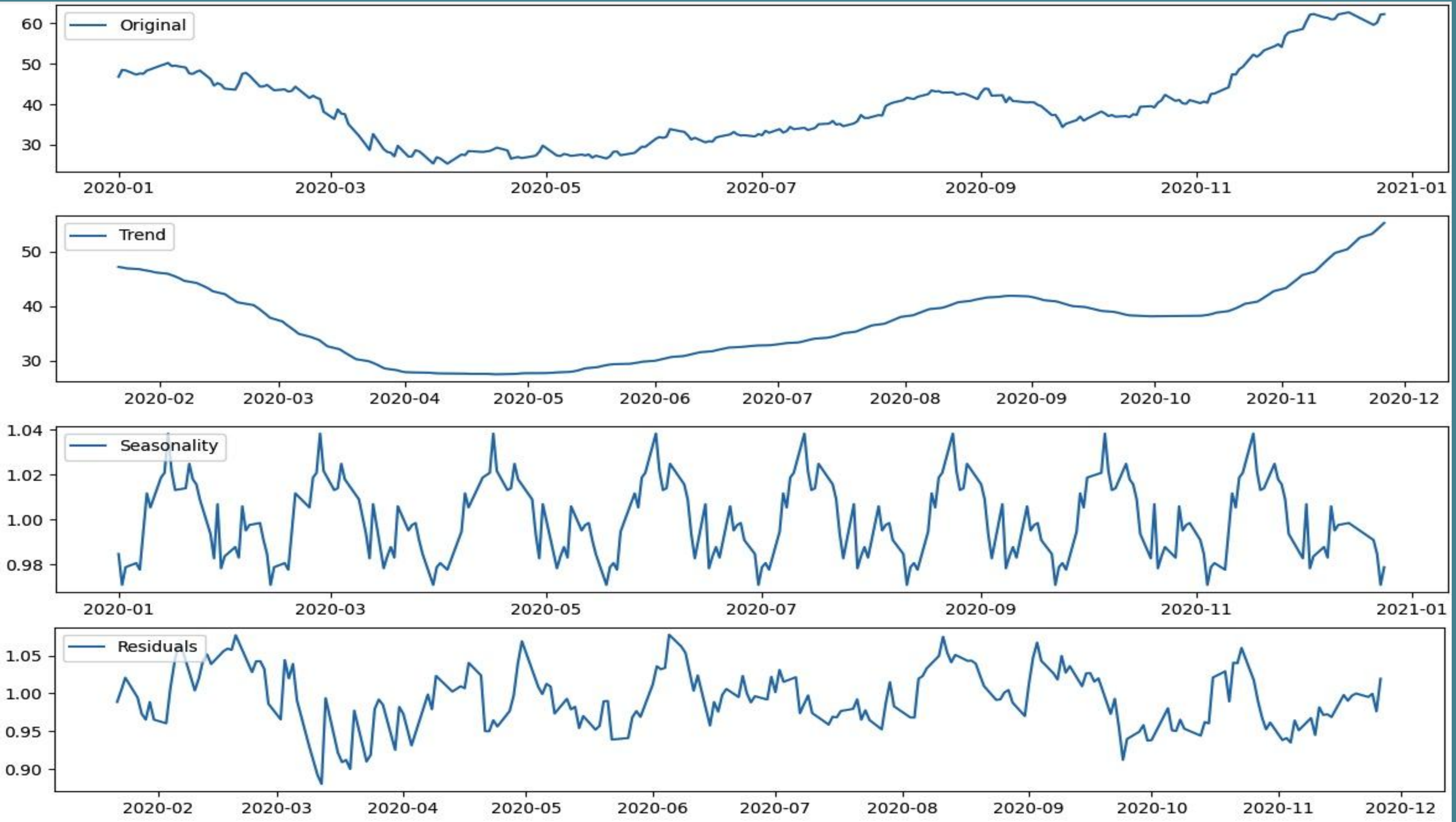
Partial Autocorrelation Function (PACF)

- It essentially isolates the direct relationship between the current value and a lagged value, excluding the influence of other lags.
- The PACF helps to determine the appropriate number of lag terms (autoregressive terms) to include in an ARIMA model.

```
# Plot ACF and PACF
plt.figure(figsize=(12, 6))
plt.subplot(121)
plot_acf(data_diff, ax=plt.gca())
plt.title('Autocorrelation Function')
plt.subplot(122)
plot_pacf(data_diff, ax=plt.gca())
plt.title('Partial Autocorrelation Function')
plt.show()
```

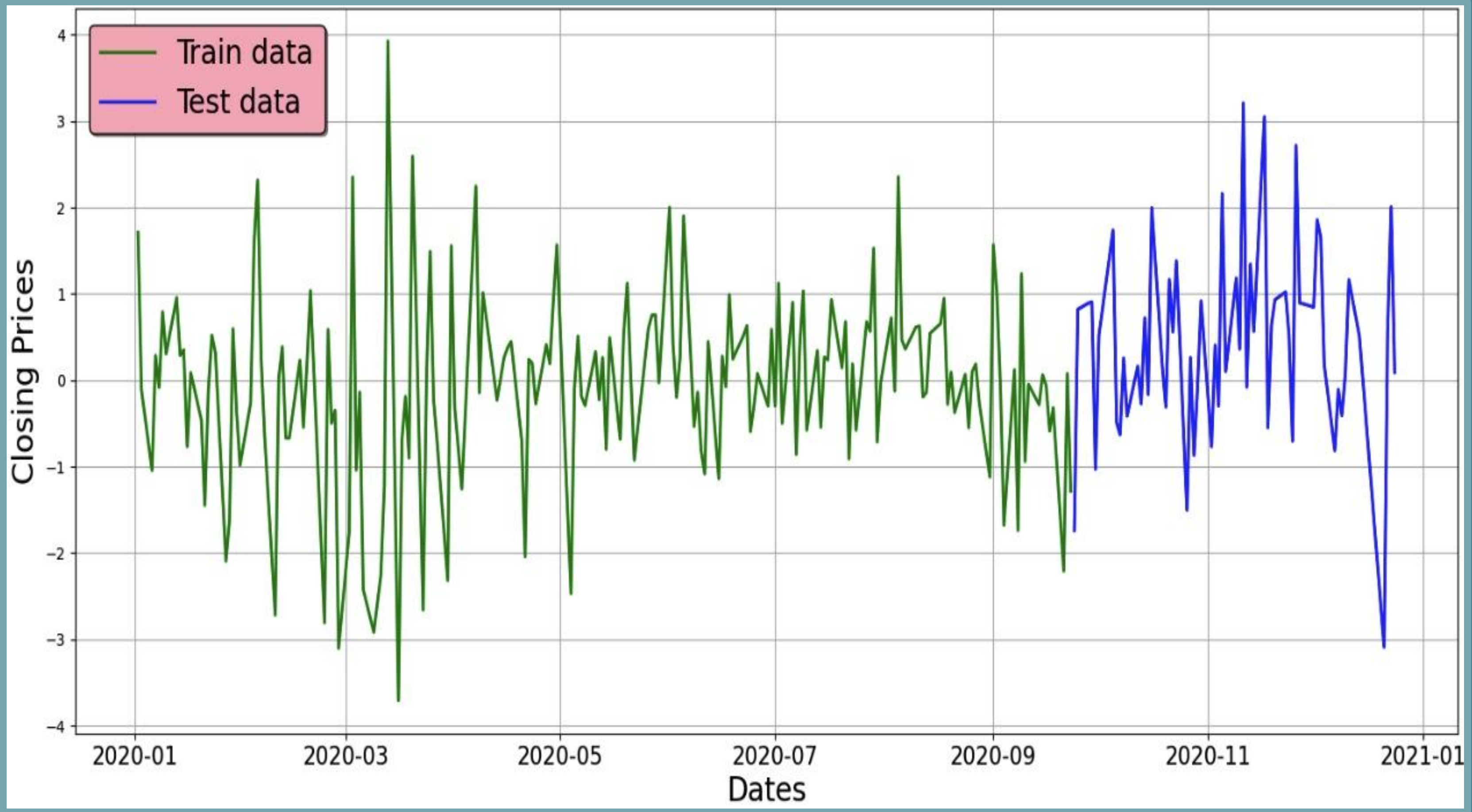


```
[73] # Decompose the time series into trend, seasonal, and residual components
     decomposition = seasonal_decompose(df['Close'], model='multiplicative', period=30)
     trend = decomposition.trend
     seasonal = decomposition.seasonal
     residual = decomposition.resid

     # Plot the decomposition components
     plt.figure(figsize=(12, 8))
     plt.subplot(411)
     plt.plot(df['Close'], label='Original')
     plt.legend(loc='upper left')
     plt.subplot(412)
     plt.plot(trend, label='Trend')
     plt.legend(loc='upper left')
     plt.subplot(413)
     plt.plot(seasonal, label='Seasonality')
     plt.legend(loc='upper left')
     plt.subplot(414)
     plt.plot(residual, label='Residuals')
     plt.legend(loc='upper left')
     plt.tight_layout()
     plt.show()
```



Original
60
50
40
30
2020-01
2020-03
2020-05
2020-07
2020-09
2020-11
2021-01
Trend
50
40
30
2020-02
2020-03
2020-04
2020-05
2020-06
2020-07
2020-08
2020-09
2020-10
2020-11
2020-12
Seasonality
1.04
1.02
1.00
0.98
2020-01
2020-03
2020-05
2020-07
2020-09
2020-11
2021-01
Residuals
1.05
1.00
0.95
0.90
2020-02
2020-03
2020-04
2020-05
2020-06
2020-07
2020-08
2020-09
2020-10
2020-11
2020-12

```
[58] # Split the data into train and test sets
     train_data = data_diff[:-60]
     test_data = data_diff[-60:]

     # Plotting
     plt.figure(figsize=(18, 8))
     plt.grid(True)
     plt.xlabel('Dates', fontsize=20)
     plt.ylabel('Closing Prices', fontsize=20)
     plt.xticks(fontsize=15)
     plt.plot(train_data, 'g', label='Train data', linewidth=2)
     plt.plot(test_data, 'b', label='Test data', linewidth=2)
     plt.legend(fontsize=20, shadow=True, facecolor='lightpink', edgecolor='k')
     plt.show()
```


Train data
Test data
Closing Prices
Dates
4
3
2
1
0
−1
−2
−3
−4
2020-01
2020-03
2020-05
2020-07
2020-09
2020-11
2021-01

GETTING THE BEST ARIMA ORDER

```
import itertools
import statsmodels.api as sm

# Define the p, d, and q parameters to take any value between 0 and 2
p = d = q = range(0, 3)

# Generate all different combinations of p, d, and q triplets
pdq = list(itertools.product(p, d, q))

# Grid search for the optimal ARIMA parameters
best_aic = float("inf")
best_order = None

for order in pdq:
    try:
        model = sm.tsa.ARIMA(train['Close'], order=order)
        results = model.fit()
        aic = results.aic
        if aic < best_aic:
            best_aic = aic
            best_order = order
    except:
        continue

print("Best ARIMA order:", best_order)
print("Best AIC:", best_aic)
```

```
Best ARIMA order: (0, 1, 0)
Best AIC: 587.0052552346217
```

```python
history = [x for x in train_data]
# Creating and fitting the ARIMA model
model = ARIMA(history, order=(0, 1, 0))
model_fit = model.fit()
# Displaying the model summary
print(model_fit.summary())
```

```python
[61] def train_arima_model(X, y, arima_order):
        # prepare training dataset
        history = [x for x in X]
        predictions = list()

        for t in range(len(y)):
            model = ARIMA(history, order=arima_order)
            model_fit = model.fit()
            yhat = model_fit.forecast()[0]  # Get the forecast value
            predictions.append(yhat)
            history.append(y[t])

        # calculate out-of-sample error
        rmse = np.sqrt(mean_squared_error(y, predictions))
        return rmse
```

```python
] #model evaluation
def evaluate_models(dataset, test, p_values, d_values, q_values):
    dataset = dataset.astype('float32')
    best_score, best_cfg = float("inf"), None

    for p in p_values:
        for d in d_values:
            for q in q_values:
                order = (p, d, q)
                try:
                    rmse = train_arima_model(dataset, test, order)
                    if rmse < best_score:
                        best_score, best_cfg = rmse, order
                    print('ARIMA%s RMSE=%.3f' % (order, rmse))
                except Exception as e:
                    print(f"Error with order {order}: {e}")
                    continue

    print('Best ARIMA%s RMSE=%.3f' % (best_cfg, best_score))
```

```python
# Prepare the training data
history = [x for x in train_data]
predictions = list()
conf_list = list()

for i in range(len(test_data)):
    model = ARIMA(history, order=(0,1,0))
    model_fit = model.fit()
    fc = model_fit.forecast(alpha=0.05)  # Get the forecast
    predictions.append(fc[0])
    history.append(test_data[i])

# Convert predictions to a Pandas Series
fc_series = pd.Series(predictions, index=test_data.index)

# Calculate and print RMSE
rmse = np.sqrt(mean_squared_error(test_data, fc_series))
print(f"RMSE is {rmse}")
```

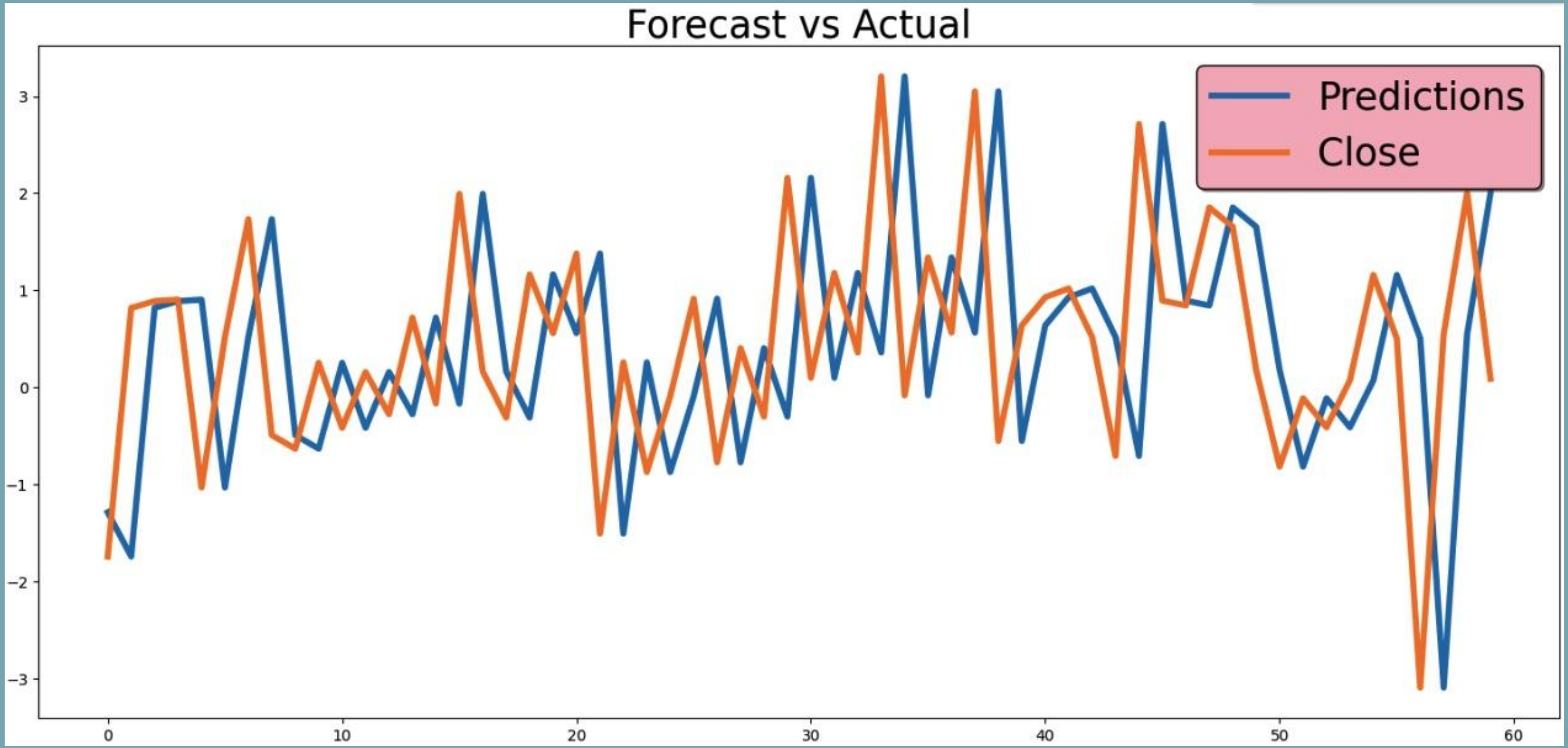
```
RMSE is 1.166065973154891
```

```python
[65] # Iterate through the test data
     for t in range(len(test_data)):
         model = SARIMAX(history, order=(0, 1, 0), seasonal_order=(0, 0, 0, 0))
         model_fit = model.fit(disp=False)
         fc = model_fit.forecast()
         predictions.append(fc[0])
         history.append(test_data[t])

     # Convert predictions to a Pandas Series
     fc_series = pd.Series(predictions, index=test_data.index)

     # Calculate and print RMSE
     rmse = np.sqrt(mean_squared_error(test_data, fc_series))
     print('RMSE of SARIMA Model:', rmse)
```

```
RMSE of SARIMA Model: 1.6484481633429833
```

ACTUAL VS PREDICTED GRAPH

```
plt.figure(figsize=(18, 8))
plt.title('Forecast vs Actual', fontsize=25)
plt.plot(range(60), predictions, label='Predictions', linewidth=4)
plt.plot(range(60), test_data, label='Close', linewidth=4)
plt.legend(fontsize=25, shadow=True, facecolor='lightpink', edgecolor='k')
plt.show()
```


Forecast vs Actual
Predictions
Close
3
2
1
0
−1
−2
−3
0
10
20
30
40
50
60

THANK YOU!!