Time Series Forecasting of Stock Market

Priyangshu Chandra Das +91-8132873151

pcdpcdjbx@gmail.com
GitHub Profile
LinkedIn Profile
Project Link : https://github.com/5PCD3/Time-Series-Forecasting-of-Stock-Market.git

Abstract

This project explores stock market data through Exploratory Data Analysis (EDA) and statistical tests, including the Dickey-Fuller test and ACF/PACF plots. The data's stationarity was assessed using the Augmented Dickey-Fuller (ADF) test. An ARIMA model was then applied, achieving a Mean Absolute Percentage Error (MAPE) of 3.41%, which reflects a prediction accuracy of 96.59% for the next 16 observations.

1. Introduction

In this analysis, we explore Tesla Inc. (TSLA) stock prices to uncover trends and forecast future movements. Using historical data, we perform exploratory data analysis (EDA) to visualize trends and distributions. We then test for stationarity and decompose the time series into its trend, seasonal, and residual components.

To fine-tune our forecasts, we analyze autocorrelation patterns and fit an ARIMA model to the data. Our goal is to identify the best model parameters and evaluate its accuracy, providing insights into Tesla's stock price behavior and potential future trends.

2. Dataset Description

The dataset used for this analysis is '**tsla.csv**', which contains 759 columns of data. This dataset includes various features related to Tesla's stock prices over time. For illustrative purposes, we present the first 5 data points from the dataset in tabular form below. This sample helps in understanding the structure and type of data used in the analysis.

Table 1. First 5 Data Points from the '**tsla.csv**' Dataset

Date	Open	High	Low	Close	Volume	Dividends	Stock Splits
2019-05-21	39.55	41.48	39.21	41.02	90019500	0	0.0
2019-05-22	39.82	40.79	38.36	38.55	93426000	0	0.0
2019-05-23	38.87	39.89	37.24	39.10	132735500	0	0.0
2019-05-24	39.97	39.99	37.75	38.13	70683000	0	0.0
2019-05-28	38.24	39.00	37.57	37.74	51564500	0	0.0

3. Data Analysis and Preparation

In this section, we will analyze and prepare the data by focusing on the time series. We will first assess whether the time series data is stationary. If we find that the data is non-stationary, we will apply the necessary transformations to achieve stationarity. Ensuring stationarity is crucial for accurate and reliable time series analysis.

3.1. Initial Steps of Data Analysis

Firstly, we extracted the univariate time series from the dataset, focusing on the 'Date' and 'Close' columns. We verified the data types and found that 'Close' was a float type, while 'Date' was an object type. To make the 'Date' column understandable by the machine, we converted it to datetime format and set it as the index.

Why Convert the 'Date' Column to an Index?

Converting the 'Date' column to a 'DatetimeIndex' optimizes time-based operations like resampling and rolling calculations. Many time series forecasting models require the 'Date' column to be the index to capture time-based dependencies correctly. This conversion also enhances data retrieval efficiency and simplifies visualization, as libraries automatically use the 'Date' index for the x-axis in time series plots.

Next, we visualized the closing price over time using a line graph.

Figure 1. Line graph of the closing price over time.

To understand the feature distribution, we plotted a histogram and a Kernel Density Estimate (KDE) curve.

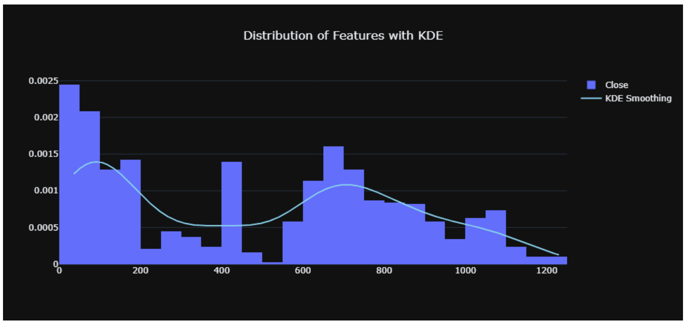

Figure 2. Histogram and Kernel Density Estimate (KDE) curve of the closing prices.

We also checked for the presence of outliers by visualizing a box-plot.

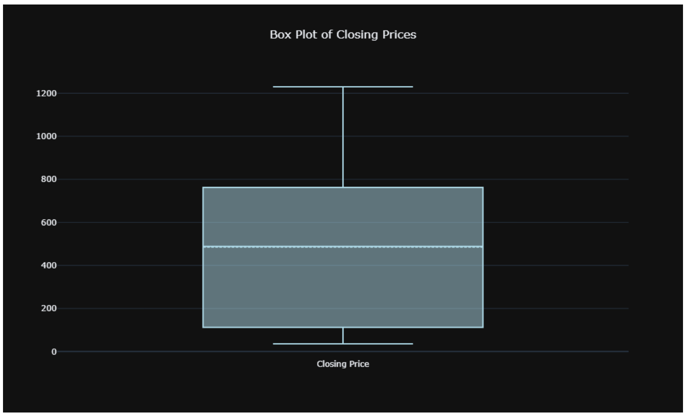

Figure 3. Boxplot showing the distribution of closing prices and potential outliers.

We performed all these tasks using mathematical computation libraries such as Numpy, Pandas, Plotly, and Scipy.

4. Stationarity Analysis and Trend Visualization

In this section, we verify the stationarity of the time series dataset using the Dickey-Fuller test. We explore various aspects of the time series by visualizing decomposition plots for both additive and multiplicative decompositions. Additionally, we examine the temporal dependencies in the time series data by plotting the Autocorrelation Function (ACF) and Partial Autocorrelation Function (PACF).

4.1. Understanding Stationarity

Stationarity is a key concept in time series analysis. A time series is stationary if its statistical properties do not change over time. In simpler terms, this means the data behaves in a consistent way throughout the time period you are analyzing.

For example, if you're looking at daily temperatures, a stationary time series would have a consistent pattern of high and low temperatures, without any noticeable long-term trend.

4.2. Why Check for Stationarity?

Many statistical methods, like forecasting models, assume that the data is stationary. If the data isn't stationary, these methods may not work well. So, we need to check and often transform the data to make it stationary.

4.3. Dickey-Fuller Test

The **Dickey-Fuller test** is a statistical test used to determine whether a given time series is stationary or not. It is especially useful for identifying whether a series contains a unit root, which is a sign of non-stationarity.

4.3.1. Mathematical Model

The test works by analyzing the following type of model:

$$y_t = \alpha + \beta t + \gamma y_{t-1} + \epsilon_t$$

Where:

- y_t is the value of the time series at time t.
- α is a constant (intercept).
- βt represents a trend component over time.
- γy_{t-1} indicates the relationship between y_t and its previous value y_{t-1}.
- ϵ_t is the error term, accounting for any randomness in the data.

4.3.2. Hypothesis Testing

The Dickey-Fuller test involves the following hypotheses:

- **Null Hypothesis (H0)**: The time series has a unit root ($\gamma = 0$), meaning it is non-stationary.
- **Alternative Hypothesis (H1)**: The time series does not have a unit root ($\gamma < 0$), meaning it is stationary.

4.3.3. Test Statistic

The test calculates a statistic, which is compared to critical values from a pre-determined table (P-Table). The decision rule is as follows:

- If the test statistic (P-Value) is smaller than the critical value (0.05), we reject the null hypothesis and conclude that the series is stationary.
- If the test statistic (P-Value) is larger than the critical value (0.05), we fail to reject the null hypothesis and conclude that the series is non-stationary.

4.3.4. Example

Imagine you are analyzing the number of goals scored in football matches over several seasons. You notice that the number of goals fluctuates randomly over time. To check if these fluctuations are random or follow a trend, you would use the Dickey-Fuller test. The model would look like this:

$$\text{Goals}_t = \alpha + \beta t + \gamma \text{Goals}_{t-1} + \epsilon_t$$

Where:

- Goals_t is the number of goals at time t.
- Goals_{t-1} is the number of goals in the previous match (time $t-1$).

After running the test, you would compare the resulting statistic to a critical value:

- If the test statistic is smaller than the critical value, the number of goals follows a consistent pattern (stationary).
- If the test statistic is larger, the number of goals shows a changing trend over time (non-stationary).

4.4. Result of Performing Dickey-Fuller Test on Time Series Dataset

After applying the Dickey-Fuller test to the time series dataset, we obtained a p-value of 0.5999, which is greater than the significance level of 0.05. As a result, we fail to reject the null hypothesis and conclude that the time series is non-stationary.

To further illustrate this, we plotted a line graph of the closing prices over time, along with the rolling mean and standard deviation. For the rolling calculations, we used a window of 183 days to observe how the mean and variability evolve over time.

Figure 4. Line graph of closing prices with rolling mean and standard deviation (window: 183 days) for non-stationary series.

4.5. Time Series Decomposition

Time series data consists of observations taken at consecutive points in time. These data can often be decomposed into multiple components to better understand the underlying patterns and trends. Time series decomposition refers to the process of separating a time series into its constituent components, such as trend, seasonality, and residual (noise). In this section, we will explore various time series decomposition techniques, their types, and provide code samples for each.

4.5.1. Components of Time Series Decomposition

Time series decomposition helps to break down a time series dataset into three main components:

- **Trend**: The trend component represents the long-term movement in the data, showing the underlying pattern over time.
- **Seasonality**: The seasonality component captures repeating, short-term fluctuations caused by factors such as seasons, cycles, or events.
- **Residual (Noise)**: The residual component represents the random variability that remains after removing the trend and seasonality.

By separating these components, we can gain insights into the behavior of the data and make better forecasts.

4.5.2. Types of Time Series Decomposition Techniques

Additive Decomposition In additive decomposition, the time series is expressed as the sum of its components:

$$Y(t) = Trend(t) + Seasonal(t) + Residual(t) \tag{1}$$

This method is suitable when the magnitude of the seasonality does not vary with the magnitude of the time series.

Multiplicative Decomposition In multiplicative decomposition, the time series is expressed as the product of its components:

$$Y(t) = Trend(t) \times Seasonal(t) \times Residual(t) \tag{2}$$

This method is suitable when the magnitude of the seasonality scales with the magnitude of the time series.

4.5.3. Decomposition of the Non-Stationary Time Series

We applied both additive and multiplicative decomposition techniques to the non-stationary time series dataset. This allows us to visually understand the trend and seasonality present in the data. By plotting these decompositions, we can examine how the different components (trend, seasonality, and residuals) behave over time.

Additive Decomposition The additive decomposition separates the time series into components where the seasonality remains constant regardless of the magnitude of the trend.

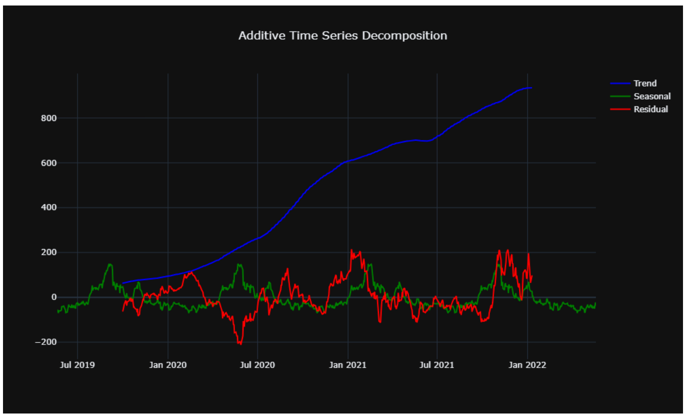

Figure 5. Additive Decomposition of the Time Series Data) for non-stationary series.

Multiplicative Decomposition The multiplicative decomposition separates the time series into components where the seasonality changes in proportion to the magnitude of the trend.

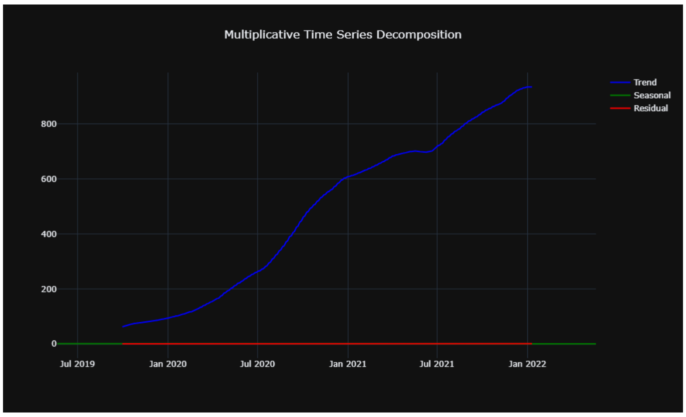

Figure 6. Multiplicative Decomposition of the Time Series Data) for non-stationary series.

4.6. Autocorrelation and Partial Autocorrelation Functions

In time series analysis, it's important to understand how current values are related to past values. Two key tools for this are the Autocorrelation Function (ACF) and the Partial Autocorrelation Function (PACF). These functions help identify patterns and relationships over time, making them essential for time series forecasting models like ARIMA.

4.6.1. Autocorrelation Function (ACF)

The Autocorrelation Function (ACF) measures how similar the time series is to itself at different time lags. Essentially, it checks if the value of the series at one point in time is related to the value of the series at a different point in time.

The formula for calculating the ACF at a lag k is:

$$\mathrm{ACF}(k) = \frac{\sum_{t=k+1}^{T} (y_t - \bar{y})(y_{t-k} - \bar{y})}{\sum_{t=1}^{T} (y_t - \bar{y})^2}$$

Where:

- y_t is the value of the time series at time t,
- y_{t-k} is the value of the time series at time $t-k$,
- $\bar{y}$ is the mean of the series,
- T is the total number of observations,
- k is the lag.

4.6.2. Partial Autocorrelation Function (PACF)

The Partial Autocorrelation Function (PACF) tells us the direct relationship between an observation and its lag k, while controlling for the effects of the values in between. It removes the influence of intermediate lags, allowing us to see the pure effect of each lag.

The formula for the PACF at a lag k is:

$$\mathrm{PACF}(k) = \mathrm{corr}(y_t, y_{t-k} \mid y_{t-1}, y_{t-2}, \ldots, y_{t-(k-1)})$$

This shows the correlation between y_t and y_{t-k}, while removing the effects of the values between t and $t-k$.

4.6.3. Application of ACF and PACF

I applied both the ACF and PACF functions to the stationary time series dataset to visualize and understand the temporal dependencies in the data. The ACF plot helps in identifying patterns and lags where the data exhibits autocorrelation, while the PACF plot helps in determining the lag order for models like ARIMA by showing the direct influence of specific lags.

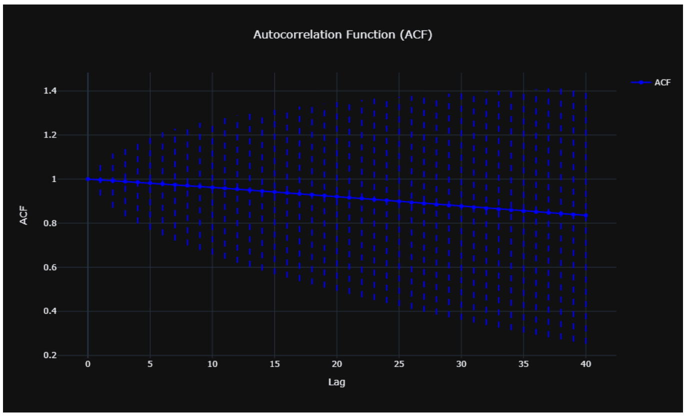

Figure 7. ACF plot for the stationary time series dataset

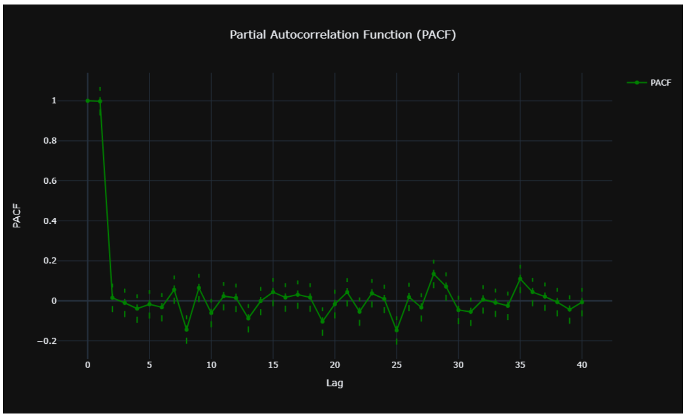

Figure 8. PACF plot for the stationary time series dataset

4.7. Converting Time Series from Non-Stationary to Stationary

To build a robust time series model, it is crucial to convert the non-stationary series into a stationary one. We applied the differencing method to remove trends and stabilize the mean of the series. After applying differencing, we verified stationarity using the Dickey-Fuller test. Additionally, for visual confirmation, we plotted the rolling mean and standard deviation over time to observe if the series has become stationary.

5. Differencing Method for Converting Non-Stationary to Stationary Time Series

Time series data often exhibit trends or patterns that change over time. To make such data stationary, which is necessary for many time series models, we use a technique called differencing. Here's a simple explanation of how differencing works and why it is useful.

5.1. What is Differencing?

Differencing is a method used to transform a time series into a stationary series. The main idea is to subtract the previous observation from the current observation. This process helps to remove trends and make the data more stable over time.

5.2. How to Apply Differencing

1. **First Differencing:**
   - Subtract the value of the time series at time $t-1$ from the value at time t.
   - Mathematically, if y_t is the value at time t, the differenced value is:
     $$\Delta y_t = y_t - y_{t-1}$$
   - This step helps to remove linear trends.
2. **Second Differencing (if needed):**
   - If the first differencing is not sufficient, apply differencing again.
   - This involves differencing the already differenced series:
     $$\Delta^2 y_t = \Delta y_t - \Delta y_{t-1}$$
   - This method helps to remove more complex patterns.

5.3. Why Differencing Works

Differencing works by removing systematic changes in the mean of the time series. If the data has a trend, differencing helps to stabilize the mean and make the data more consistent over time.

5.4. Verifying Stationarity

After applying differencing, you need to verify that the time series is now stationary. Two common methods for verification are:

- **Dickey-Fuller Test:**
  - This statistical test checks whether a time series has a unit root, indicating non-stationarity. If the p-value from the test is less than a threshold (usually 0.05), the series is considered stationary.
- **Visual Inspection:**
  - Plot the rolling mean and standard deviation of the differenced time series. If these plots show constant mean and variance over time, the series is likely stationary.

5.5. Example

For example, consider monthly sales data showing an increasing trend. By applying differencing (subtracting each month's sales from the previous month's), you can remove this trend, making the series stationary and more suitable for modeling.

5.6. Verifying the Stationarity after Applying Differencing

After applying differencing, we performed the Dickey-Fuller test and obtained a p-value of 3.498786×10^{-13}, which is less than 0.05. By rejecting the null hypothesis, we can conclude that the time series has become stationary.

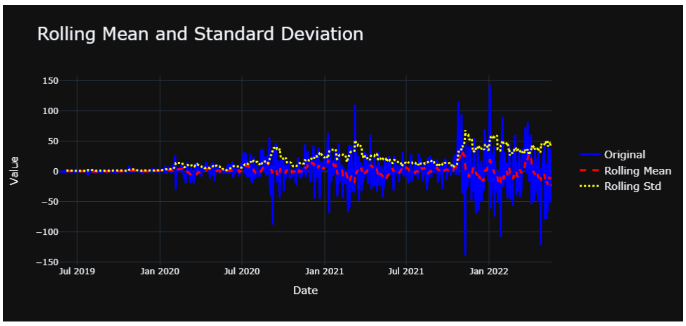

Figure 9. Rolling Mean and Standard Deviation Plot After Differencing

6. Splitting Training and Test Data

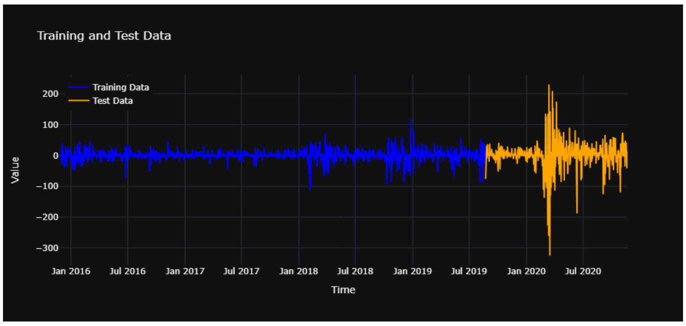

Figure 10. Training and Test Data Plot

We split the data into two parts: we use the last 60 data points for testing the model and use the remaining data for training purposes.

7. Model Building and Fine-Tuning

We build and train our model using the ARIMA approach. To fine-tune the model, we use the Root Mean Squared Error (RMSE) as our evaluation metric. The RMSE formula is given by:

$$\text{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}$$

where y_i represents the observed values, $\hat{y}_i$ represents the predicted values, and n is the number of observations. We use RMSE to assess the model's accuracy and make adjustments to improve its performance.

7.1. ARIMA Model and Parameter Estimation

The ARIMA model helps us predict future values in a time series. Imagine you have a list of numbers that change over time, like the monthly temperature of a city. The ARIMA model uses patterns in these numbers to guess what the next number will be.

7.1.1. ARIMA Components

The ARIMA model has three parts:

- **Auto-Regressive (AR) Part:**

$$y_t = \phi_1 y_{t-1} + \phi_2 y_{t-2} + \cdots + \phi_p y_{t-p} + \epsilon_t \tag{3}$$

  This part looks at how previous numbers affect the current number. Here, y_t is the current value, ϕ represents the effect of past values, and ϵ_t is the error.
- **Integrated (I) Part:**

$$\Delta y_t = y_t - y_{t-1} \tag{4}$$

  This part calculates the difference between the current and previous values to make the data more stable.
- **Moving Average (MA) Part:**

$$y_t = \mu + \theta_1 \epsilon_{t-1} + \theta_2 \epsilon_{t-2} + \cdots + \theta_q \epsilon_{t-q} + \epsilon_t \tag{5}$$

  This part looks at past errors to make better predictions. Here, μ is the mean, θ represents past errors, and ϵ_t is the current error.

7.1.2. Parameter Estimation

To estimate the parameters (like ϕ, θ, and μ), we use historical data to find the best values that make the model fit the data well. This process involves using mathematical techniques to minimize the difference between the predicted values and the actual values.

7.1.3. Automated Training and Fine-Tuning

To efficiently build and fine-tune ARIMA models, we follow an automated process that involves several key steps. This approach helps us systematically evaluate multiple models to find the best fit for our data. Here's a step-by-step breakdown:

1. **Prepare Training and Test Data:** We start by dividing our time series data into two parts:
   - **Training Data:** Used to build and train the ARIMA models.
   - **Test Data:** Used to evaluate the performance of the models.
2. **Define ARIMA Parameters:** We select a range of values for the ARIMA parameters, which include:
   - p: The number of lag observations included in the model (Auto-Regressive part).
   - d: The number of times the raw observations are differenced (Integrated part).
   - q: The size of the moving average window (Moving Average part).
3. **Train and Evaluate Models:** We then train and evaluate ARIMA models for each combination of (p, d, q) values:
   - Build an ARIMA model using the current set of parameters.
   - Fit the model to the training data.
   - Generate forecasts and calculate the Root Mean Squared Error (RMSE) to assess the model's performance.
4. **Select the Best Model:** We compare the RMSE values for all the models and select the one with the lowest RMSE as the best model. The RMSE measures how well the model's forecasts match the actual data, with a lower value indicating a better fit.

The automated process evaluated various ARIMA models with different parameter combinations. The model ARIMA(2, 0, 0) was found to have the lowest RMSE of 39.440, making it the best model for our data.

Here's the pseudocode used in this automated process:

```
def train_arima_model(X, y, arima_order):
    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]
        predictions.append(yhat)
        history.append(y[t])
    rmse = np.sqrt(mean_squared_error(y, predictions))
    return rmse

def evaluate_models(train, test, p_values,
                    d_values, q_values):
    train = train.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(train,
                                             test, order)
                    if rmse < best_score:
                        best_score, best_cfg =rmse,order
                    print(
                        f'ARIMA{order} '
                        f'MAPE={rmse:.3f}'
```

```
                    )
                except:
                    continue
    print(f'Best ARIMA{best_cfg} RMSE={best_score:.3f}')
```

7.1.4. Explanation of Pseudocode

1. Define the Train ARIMA Model Function

```
def train_arima_model(X, y, arima_order):
```

Explanation: This line defines a function called `train_arima_model` that takes three inputs:

- `X`: The training data.
- `y`: The actual values to predict.
- `arima_order`: A tuple specifying the parameters of the ARIMA model.

2. Initialize History

```
    history = [x for x in X]
```

Explanation: Creates a list called `history` which starts as a copy of the `X` list. This list is used to train the model.

Example: If `X = [10, 20, 30]`, then `history` will also be `[10, 20, 30]`.

3. Initialize Predictions

```
    predictions = list()
```

Explanation: Initializes an empty list called `predictions` that will store forecasted values.

4. Loop Through Each Time Step

```
    for t in range(len(y)):
```

Explanation: Starts a loop that runs for each value in `y`.

Example: If `y = [5, 6, 7]`, the loop will run 3 times.

5. Create ARIMA Model

```
        model = ARIMA(history, order=arima_order)
```

Explanation: Creates an ARIMA model object with the current `history` data and the specified `arima_order`.

Example: If `arima_order = (1, 0, 1)`, the ARIMA model is configured with these parameters.

6. Fit the Model

```
        model_fit = model.fit()
```

Explanation: Trains the ARIMA model on the `history` data.

7. Forecast Next Value

```
        yhat = model_fit.forecast()[0]
```

Explanation: The `forecast()` method predicts the next value. `[0]` retrieves the first prediction.

Example: If the forecasted value is 25, then `yhat` is 25.

8. Append Prediction

```
        predictions.append(yhat)
```

Explanation: Adds the predicted value (`yhat`) to the `predictions` list.

Example: If `predictions` was `[22]` and `yhat` is 25, then `predictions` becomes `[22, 25]`.

9. Update History

```
        history.append(y[t])
```

Explanation: Adds the actual value from `y` at position `t` to the `history` list.

Example: If `history` was `[10, 20, 30, 25]` and `y[t]` is 26, then `history` becomes `[10, 20, 30, 25, 26]`.

10. Calculate RMSE

```
    rmse = np.sqrt(mean_squared_error(y, predictions))
```

Explanation: Calculates the RMSE (Root Mean Squared Error), which measures how well the model's predictions match the actual values. RMSE is calculated by:

- Finding the squared differences between actual and predicted values.
- Taking the average of these squared differences.
- Taking the square root of this average.

Example Calculation: If `y = [5, 6, 7]` and `predictions = [5.1, 6.1, 6.8]`:

- Differences: `[0.1, 0.1, 0.2]`
- Squared Differences: `[0.01, 0.01, 0.04]`
- Mean Squared Error: `(0.01 + 0.01 + 0.04) / 3 = 0.02`
- RMSE: `sqrt(0.02) = 0.14`

11. Return RMSE

```
    return rmse
```

Explanation: The function returns the RMSE value, indicating how well the model performed.

12. Define the Evaluate Models Function

```
def evaluate_models(train, test, p_values,
                    d_values, q_values):
```

Explanation: Defines the `evaluate_models` function which takes:

- `train`: Training data.
- `test`: Test data.
- `p_values`, `d_values`, `q_values`: Lists of possible values for ARIMA parameters.

13. Convert Training Data

```
    train = train.astype('float32')
```

Explanation: Converts the training data to a specific format (`float32`) for consistency in calculations.

14. Initialize Best Score and Configuration

```
    best_score, best_cfg = float("inf"), None
```

Explanation: Initializes `best_score` to infinity and `best_cfg` to `None`. These will store the best (lowest) RMSE and the corresponding ARIMA configuration.

15. Loop Through p Values

```
    for p in p_values:
```

Explanation: Starts a loop over all possible values for the `p` parameter of the ARIMA model.

16. Loop Through d Values

```
    for d in d_values:
```

Explanation: Starts a nested loop over all possible values for the `d` parameter.

17. Loop Through q Values

```
for q in q_values:
```

Explanation: Starts another nested loop over all possible values for the q parameter.

18. Create Order Tuple

```
order = (p, d, q)
```

Explanation: Creates a tuple for the current combination of `p`, `d`, and `q` values.

19. Try to Train Model

```
try:
```

Explanation: Starts a block where we attempt to execute the code and catch any errors that might occur.

20. Train Model and Calculate RMSE

```
rmse = train_arima_model(train, test, order)
```

Explanation: Uses the `train_arima_model` function to train the model with the current `order` and calculate the RMSE.

21. Check if RMSE is Best

```
if rmse < best_score:
```

Explanation: Checks if the current RMSE is better (lower) than the previous best score.

22. Update Best Score and Configuration

```
best_score, best_cfg = rmse, order
```

Explanation: If the current RMSE is better, updates `best_score` and `best_cfg` with the current RMSE and `order`.

23. Print ARIMA Configuration and RMSE

```
print(f'ARIMA{order} MAPE={rmse:.3f}')
```

Explanation: Prints the ARIMA model configuration and its RMSE value in a formatted way.

24. Handle Errors

```
except:
```

Explanation: Catches any errors that occur in the `try` block and continues to the next iteration.

25. Print Best ARIMA Model

```
print(f'Best ARIMA{best_cfg} RMSE={best_score:.3f}')
```

Explanation: Prints the best ARIMA model configuration and its RMSE value.

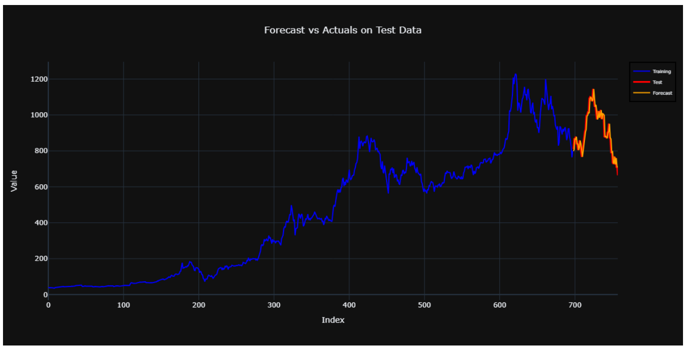

Figure 11. Comparing forecast data on actual test data by this plot

8. Result & Conclusion

After identifying the best model, ARIMA(2, 0, 0), we forecasted 16 future values. The performance of the model is evaluated as follows:

- **Mean Absolute Percentage Error (MAPE)**: 3.41%
- **Accuracy**: 96.59%

The Mean Absolute Percentage Error (MAPE) is calculated using the formula:

$$\text{MAPE} = \frac{1}{n}\sum_{i=1}^{n}\left|\frac{A_i - F_i}{A_i}\right| \times 100\% \tag{6}$$

where:

- n is the number of observations,
- A_i is the actual value at time i,
- F_i is the forecasted value at time i.

To validate our model's predictions, we compared the forecasted values with the actual test data. The graph below shows the true test values versus the next 16 predicted values for visual verification.

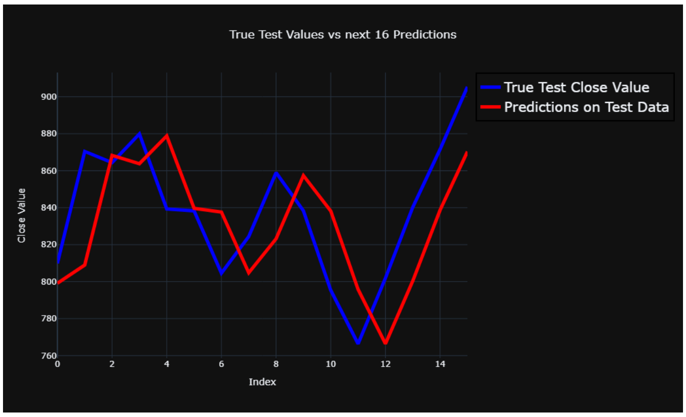

Figure 12. True Test Values vs. Next 16 Predictions