School of Mathematics and Physical Science

Stock market forecasting (generic)

Candidate number:

277185

Project Supervisor:

Dr Peter Wijeratne

Final word count:

13,913

Abstract

This work presents a thorough analysis of the use of two well-known Long Short-Term Memory (LSTM) networks and Autoregressive Integrated Moving Average (ARIMA) time series forecasting models in the prediction of stock values, with a focus on Apple Inc. (AAPL) shares. By employing past data from January 1, 2016, to the present, this study aims to contribute to the body of information already available on stock price predictions. It also aims to offer helpful information about the relative benefits of current machine learning techniques versus conventional statistical methods in this area.

The research methodology described in this study is multifaceted and thorough, encompassing comprehensive data collection, rigorous preprocessing, feature engineering, model development, and robust performance evaluation. The LSTM model, renowned the ability to identify enduring relationships in sequential data, is meticulously compared to the classical ARIMA model, which leverages autoregressive and moving average components to forecast temporal patterns. This comparative analysis serves to elucidate the strengths and limitations of both approaches, ultimately enabling informed decision-making regarding the most suitable model for stock price prediction.

The study's findings show that the LSTM model performs better across a range of evaluation parameters, indicating its improved predictive accuracy. The LSTM model achieved a Mean Squared Error (MSE) of 1.8104e-04, Mean Absolute Error (MAE) of 0.0099, Root Mean Squared Error (RMSE) of 2.8233, and Mean Absolute Percentage Error (MAPE) of 1.330012. In comparison, the ARIMA model produced an MSE of 6.45e-02, MAE of 0.0625, RMSE of 0.080324, and MAPE of 1.2187. However, the research also acknowledges the ARIMA model's strength in interpretability, as its statistical components offer clear interpretations, and its computational efficiency, which may be a key consideration in real-time or frequent retraining scenarios.

Notably, both the LSTM and ARIMA models struggled to accurately predict extreme stock price movements, a challenge that is widely recognized in the domain of financial forecasting. The limitation highlights the inherent unpredictability of market extremes, which are often influenced by a myriad of external factors beyond the scope of historical price data alone.

The results of this study offer a thorough examination of the LSTM and ARIMA models, contributing to the growing collection of information on stock price prediction and providing helpful advice for academics, financial analysts, and decision-makers negotiating the intricate world of financial time series forecasting. The choice between these models should be carefully considered, considering factors such as the need for long-term dependency modelling, interpretability, and computational requirements, depending on the demands of the current issue.

Furthermore, this study paves the way for future research endeavours, which may investigate creating hybrid models that combine the advantages of ARIMA and LSTM techniques, the incorporation of additional features beyond historical stock prices, the evaluation of these models on a wider range of datasets, the investigation of alternative deep learning architectures, and the enhancement of techniques for addressing the challenge of accurately predicting extreme stock price movements. By pursuing these avenues of research, the scientific community can continue to push the boundaries of stock price prediction, ultimately providing more accurate, reliable, and actionable insights to the financial industry.

Table of Contents

1. Introduction

In the financial sector, time series analysis is becoming ever more important, especially when it comes to stock price prediction. Stock prices are a classic instance of time series data, where values are collected at defined intervals, as they are a time-dependent dataset. The development of information technology has greatly improved the ability to gather and process this kind of data, creating new opportunities for financial market predictive modelling.

This work focuses on the application of two prominent models for stock price prediction, Long Short-Term Memory (LSTM) networks and Autoregressive Integrated Moving Average (ARIMA) - to predict stock prices, with a specific focus on Apple Inc. (AAPL) stock. These models show how modern approaches to machine learning and conventional statistical methods have come together to solve one of the most difficult financial economics problems.

The research employs a comprehensive methodology that includes data collection, preprocessing, model development, and performance evaluation. We utilize historical stock data from Yahoo Finance, starting from January 1, 2016, to the current day, with daily price observations. The dataset comprises key financial metrics including Open, Closing, Adj Close, High, and Low prices as well as trading volumes.

Recurrent neural network (RNN) includes LSTM which is particularly suitable for sequential data analysis. Its architecture, consisting of output gates, input gates, and forget gates, allows it to get long-term dependencies in data. This capability is important in stock price prediction, where patterns may extend over significant periods. We explore various aspects of the data, including moving averages, daily returns, and correlations between different tech stocks, to provide a holistic view of the market dynamics. LSTM models can be single or multi-layered, each offering different functionalities but sharing the ability to retain information over extended periods.

On the other side, ARIMA (Autoregressive Integrated Moving Average) is a traditional statistical technique that's commonly applied to time series forecasting. It is part of the statsmodels library in Python, which provides a comprehensive set of tools for statistical analysis and econometrics. The auto_arima function, specifically found in the pmdarima library (also known as pyramid-arima), is utilized to determine optimal parameters. This process involves testing for stationarity using methods like the Augmented Dickey-Fuller test, applying differencing when necessary to remove trends and seasonality, and analysing autocorrelation and partial autocorrelation functions through ACF and PACF plots. The auto_arima function uses techniques like BIC (Bayesian Information Criterion) or AIC (Akaike Information Criterion) to select the best combination of p, d, and q parameters, after which the ARIMA model is fit to the data. ARIMA forecasts temporal dependencies using only historical values of the time series itself. It merges all three parts together: AutoRegressive (AR) which uses past values to predict future ones, Integrated (I) which represents the differencing of raw observations and Moving Average (MA) which incorporates the relationship between a lagged collection of observations and a residual error from a moving average model. This combination makes ARIMA a robust benchmark for comparison with more complex machine learning approaches, especially in scenarios where the time series exhibits clear trends or seasonality.

While both LSTM and ARIMA models have been applied used for predicting in a variety of environments, the aim of this study is to discuss the distinctions between the predictions made by both models, particularly about stock price prediction. By using identical datasets of stock price data for training, the objective is to look at the relative performance of both the models.

Our methodology involves several key steps:

1. Gathering and preparing historical stock price information.
2. Feature engineering to derive relevant indicators from the raw data.
3. Model development and training of both LSTM and ARIMA models.

4. Standard metrics include Root Mean Squared Error (RMSE), Mean Absolute Error (MAE), Mean Squared Error (MSE), and Mean Absolute Percentage Error (MAPE) are used to evaluate performance.
5. Comparative analysis of the models' predictive capabilities.

By carrying out this thorough study, we aim to add to the knowledge already available on predicting stock prices and eliminated some understanding of the relative advantages of machine learning vs more conventional time series methods in financial forecasting. The included graphs and detailed analysis will help determine which model is most appropriate for time series forecast in the context of stock price prediction. This research not only addresses the practical challenge of stock price prediction but also explores the intersection of cutting-edge machine learning techniques and established numerical methods in the realm of financial analysis. As such, it represents a step towards filling the space between traditional econometrics and modern data science approaches in tackling one of the most persistent challenges in finance.

2. Background and Related Work

Over the past three decades, time series analysis has attracted significant interest [3]. A collection of quantitative observations grouped chronologically is termed a time series [1]. There hasn't been much comparison of time dependent variables in the past because time was frequently considered as a continuous or discontinuous variable. The first time series econometric model was created by Jan Tinbergen in 1939 [5], which provided the groundwork for subsequent econometric research studies. Time series have been widely utilised in econometrics. When conventional linear regression models were first developed, it was generally accepted that the residuals of estimated equations of stock market were randomly generated and independent of each other, and that chronologically arranged observations might not depending on one another.

The main objective of creating a precision time series model is to produce values that closely match those in the series. The distribution of observations at a given point in time depends on the values of the series at previous times, according to statistics, and Stochastic processes that change over time are recorded in time series. These recordings fall into two categories: stationary and non-stationary patterns. They can also create discrete sets of values or continuous patterns. The statistical characteristics of non-stationary time series values fluctuate, whereas those of stationary time series values are consistent. In time series forecasting models, both kinds of series can be used [2][8].

The type of dataset utilised to train the models determines how time series forecasting models are constructed. Non-stationary datasets are frequently converted into stationary ones, as stationary datasets are less difficult to train for predictions than non-stationary datasets. This transformation can be performed by several techniques, including the Fourier transform, Dicky-Fuller test, and Hilbert-Huang transform. When analysing data that is not linear and non-stationary, the Hilbert-Huang approach works especially well [7][8][9].

Predicting the results of plans is a practice as old as human civilisation: forecasting. These days, one uses readily available datasets and built-in methods to train them for forecasting in Python. Today's popular technique, time series forecasting directly influences decision-making by offering more precise estimates. Time series forecasting is a common application for both machine learning and deep learning models, considering their different operating parameters [12]. Deep neural networks (DNNs) and supervised machine learning models are frequently used for prognostications. Without explicit programming, supervised learning allows systems to automatically get better by learning from data results [10], and DNNs' expressive power frequently results in better forecasts than shallow neural networks in many applications, including detection and prediction [13].

A. ARIMA

An expanded version of the Autoregressive Moving Average (ARMA) model, the ARIMA (Autoregressive Integrated Moving Average Model) combines the Moving Average (MA) and Autoregressive (AR) processes to create a composite model of the time series. The acronym "ARIMA" implies that it includes the essential components of the model [15].

- AR: (Autoregression) A regression model that makes use of the correlations between a few lagged observations (p) and an observation.

- I: (Integrated) To determine the differences in observations at different times to make the time series stationary (d).

-MA: (Moving Average) A method that considers the relationship between the residual error terms and the data when applying a moving average model to the lag observations (q).

An AR model of order p in its simplest form, or AR(p), can be expressed as the linear process provided by:

$$x_t = c + \sum_{i=1}^{p} \emptyset_i x_{t-i} + \in_t$$

In this case, the residuals ($\in_t$) are the Gaussian white noise series with mean zero and variance σ^2, the stationary variable (x_t), the constant (c), Autocorrelation coefficients at lags 1, 2...p are represented by the terms in $\emptyset_i$. An MA model of order q, or MA(q), has the following form:

$$x_t = \mu + \sum_{i=1}^{p} \emptyset_i \in_{t-i}$$

Where $\theta_0 = 1$, the $\emptyset_i$ represents the weights assigned to the current and old values of a stochastic term in the time series, and μ is the expectation of x_t (often assumed to equal zero). We consider $\in_t$to be a Gaussian white noise series with variance equal to σ_ϵ^2 and mean zero. By combining these two models together, we may create an ARIMA model of order (p, q):

$$x_t = c + \sum_{i=1}^{p} \emptyset_i x_{t-i} + \in_t + \sum_{i=1}^{p} \emptyset_i \in_{t-i}$$

In cases when $\theta_i \neq 0, \emptyset_i \neq 0$ and $\sigma_\epsilon^2 > 0$. The terms "AR" and "MA orders" refer to the parameters p and q respectively. The "integrate" stage in ARIMA forecasting commonly referred to as Box and Jenkins forecasting enables it to handle non-stationary time series data [8]. In actuality, the "integrate" part entails time series differencing to transform transforming a time series that is not stationary into one that is. ARIMA (p, d, q) represents an ARIMA model's general form.

Given seasonal time series data, the model is probably influenced by short-term non-seasonal components. For this reason, to include both seasonal and non-seasonal components in a multiplicative model, we must estimate the seasonal ARIMA model. A seasonal the general expression for the ARIMA model is represented as ARIMA (p, d, q) × (P, D, Q)S, where p represents the seasonal AR order, d the seasonal differencing, Q the seasonal MA order, S the time span of the recurrent seasonal pattern, and q the non-seasonal MA order [23].

The computation of the (p, d, q) and (P, D, Q) values is the most essential stage in the seasonal ARIMA model estimate process. For example, if the data's time plot shows that the variance increases with time, then differencing and variability-stabilizing treatments are required. Afterwards by using the inverse autocorrelation function (IACF) to identify over differencing and the partial autocorrelation function (PACF) to calculate the number of autoregressive terms (q) that are required, we can figure out the initial values of the autoregressive order (p), the order of differencing (d), the moving average order (q), and the seasonal parameters P, D, and Q that correspond with these values.

B. LSTM

Artificial Neural Networks (ANNs) have gained immense popularity in machine learning, especially in neural networks used for this purpose. These networks build on knowledge about the structure and functioning of the human brain to approximate its capacity for learning and decision-making [17]. An input, hidden, and output layer comprise a multilayer artificial neural network design. After the input layer collects the inputs and the hidden layers process and transform them, the output layer produces the predicted results. An artificial neural network's "synapses," or connections, are distinguished among its layers by a unique "weight" that determines the degree of the link.

While traditional ANNs are adept at processing individual inputs independently, a specialized type of network known as Recurrent Neural Networks (RNNs), is designed to work with sequential data such as text or time series information [17]. RNNs have the unique ability to utilize information from previous inputs to make predictions about the current input, essentially giving them a "memory" and the capacity to capture the context of the sequence. This feature is particularly valuable in applications where the understanding of past events is crucial for making accurate forecasts or decisions. However, a limitation of basic RNNs is their inability to effectively remember and utilize information from long sequences of data. This shortcoming is addressed by a more advanced form of RNN, called Long Short-Term Memory (LSTMs). LSTMs are equipped with additional "gates" that help them selectively remember and forget information, enabling them to capture long-term dependencies more effectively within the sequential data [19].

An LSTM has three main gates: the Forget Gate, Memory Gate, and Output Gate. The memory gate decides what new information from the current step should be added to the memory, the forget gate chooses which information from the previous step should be forgotten or preserved, and the output gate selects which data from the current memory state should be used to make the current prediction [14]. Language modelling, machine translation, and time series forecasting activities containing long-term dependencies—are among the tasks that LSTMs excel at because they can efficiently regulate information flow by utilising these gates (see figure 1). The comprehensive understanding of the inner workings of LSTMs, from their foundational ANN structure to the specialized mechanisms that enable them to handle sequential data, is crucial for researchers and developers working in the field of deep learning and neural network applications [11]. By mastering the nuances of these powerful models, they can unlock their full potential and deploy them in a wide range of real-world scenarios, driving advancements in various industries and domains.

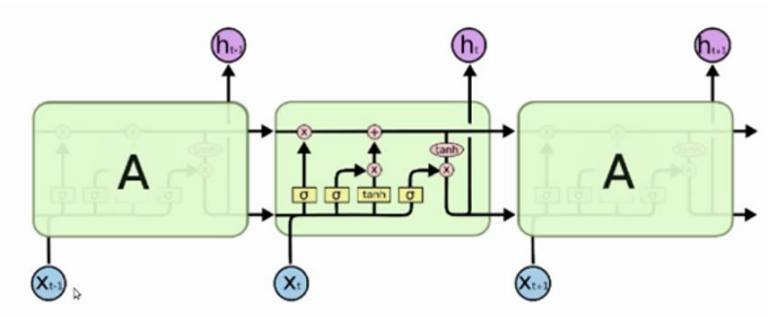


Figure 1: The internal structure of an LSTM

In the internal structure of an LSTM model, several key components work together to process and manage information flow. The 'x' typically represents the input at the current time step, which could be a feature vector or an element in a sequence. 'h' denotes the hidden state, which carries information from previous time steps and is updated at each step. The 'tanh' (hyperbolic tangent) function is an activation function used to introduce non-linearity, helping the model capture complex patterns. It squashes values to a range between -1 and 1, aiding in gradient flow during training.

Comparative Studies and Applications While both LSTM and ARIMA models have been extensively used in finance and commerce markets, their application in trading markets is less common. The literature indicates limited use of these models in stock exchange prediction, highlighting a gap in research comparing their performance on the same dataset in this context [15][16][23]. Research gap and current study focus this study aims to address the limited comparative research on LSTM and ARIMA models in stock exchange prediction. By implementing both models on the same dataset, we seek to evaluate their relative performance and determine their efficacy in this specific domain of financial forecasting.

3. Methodology

In this research, the data used for predicting stock price movements is sourced from publicly available financial datasets. These datasets typically include historical stock prices, trading volumes, and other relevant financial metrics for various companies. The datasets are pre-existing and publicly accessible, offering a rich repository of information necessary for effective time series forecasting. This study utilizes historical stock market data for four major technology companies: Apple Inc. (AAPL), Google (GOOG), Microsoft (MSFT), and Amazon (AMZN). The dataset spans from January 1, 2016, to the current date providing a comprehensive time series of daily stock prices and trading volumes. Data was sourced from Yahoo Finance, a reliable and widely used repository for financial market information.

The primary variables of interest include the daily closing prices, adjusted closing prices, and trading volumes for each stock. These variables were chosen for their significance in capturing market trends and investor sentiment. The adjusted closing price in particular accounts for corporate actions such as stock splits and dividend distributions, ensuring a consistent basis for analysis over time.

The data was retrieved using Python's pandas-datareader library, which allows for direct access to Yahoo Finance's API. The following key variables were collected:

1. Date: The trade date (that uses as the index).
2. Open: The stock's opening price at the start of each trading day.
3. High: The price that was achieved at the highest point of the trading day.
4. Low: The price that was the lowest during the trading day.
5. Close: The price at which a stock stops trading when the market closes during the trading day. This price is often used in stock price analysis and forecasting.
6. Adj Close: The closing price of the stock after adjustments for all applicable splits and dividend distributions. This value provides a more precise representation of the stock's value by considering corporate actions.
7. Volume: The overall number of shares traded for the stock during the trading day. High volume can indicate strong investor interest and activity.

Data Preprocessing

Different preprocessing steps were used to prepare the data for analysis:

1. Date Conversion: The 'Date' column was converted to datetime format and set as the index of the DataFrame.

2. Missing Values: To find any missing values, the dataset was examined. but in this case, no missing data was identified.
3. Feature Engineering: Additional features were created, including:
    - Moving averages (10-day, 20-day, and 50-day) for the 'Adj Close' price
    - Daily return percentage calculated using the percentage change in 'Adj Close' prices.

Exploratory data analysis (EDA) was carried out to understand the underlying patterns and characteristics of the data. EDA involved several steps for LSTM:

1. **Time Series Visualization**: Plotted the 'Adj Close' prices over time to visualize trends and patterns. Created subplots for each stock (AAPL, GOOG, MSFT, AMZN) to compare their price movements.
2. **Moving Averages:** Calculated and plotted 10, 20, and 50th day moving averages alongside the actual prices. This helps in understanding long-term trends and possible buying or selling indications.
3. **Volume Analysis:** Plotted daily trading volumes to identify periods of high and low trading activity. This can reveal information about the general state of the market and future price changes.
4. **Daily Returns:** Calculated and plotted daily return percentages. Created histograms of daily returns to understand the distribution of price changes.
5. **Correlation Analysis:** Computed correlation matrices between different stocks. Created heatmaps to visualize these correlations, which can inform feature selection for the LSTM model.
6. **Data Scaling:** Applied MinMaxScaler to normalize the data between 0 and 1, which is crucial for LSTM performance.
7. **Learning Performance:** These loss function values suggest how well LSTM model has learned, with relatively low error rates. However, it also hints that there might be some overfitting or underfitting since the loss on the test data is provided separately and is slightly different (usually higher) than the training loss.

EDA for ARIMA:

1. **Stationarity Test:** Performed Augmented Dickey-Fuller test on the 'Adj Close' prices to check for stationarity. If non-stationary, applied differencing to achieve stationarity.
2. **Time Series Decomposition:** Divided the time series into components which included trending, seasonal, and residual. This improves understanding of the data's underlying patterns.
3. **Autocorrelation and Partial Autocorrelation Analysis:** Created ACF and PACF plots to identify possible AR and MA values for the ARIMA model. This informs the selection of p, d, and q parameters for the ARIMA model.
4. **Log Transformation:** Applied log transformation to the 'Adj Close' prices to stabilize variance.
5. **Rolling Statistics:** Calculated and plotted rolling mean and standard deviation to visually assess stationarity.

90% of the data was implemented for training, and the other 10% was set aside for out-of-sample testing and model evaluation in the LSTM model, and 80% was used for training and the other 20% for testing in the ARIMA model. This division makes it possible to evaluate each model's predicted effectiveness on hypothetical data with a high level of confidence.

By thoroughly analysing the dataset before applying the predictive models, we ensured that the models were well-informed and tailored to the specific characteristics of the data. This approach not only improved the accuracy of the predictions but also provided a greater comprehension of the factors influencing stock price movements.

The methodology for predicting stock price movements using Long Short-Term Memory (LSTM) and Autoregressive Integrated Moving Average (ARIMA) models involves a series of steps designed to ensure accurate and reliable predictions. The steps include understanding the problem, data gathering and pre-processing, exploratory analysis, choosing libraries and training models, and evaluating models. Each step plays an important role in building and refining the models for optimal performance.

The general methodological steps are explained and shown in Fig. 2.

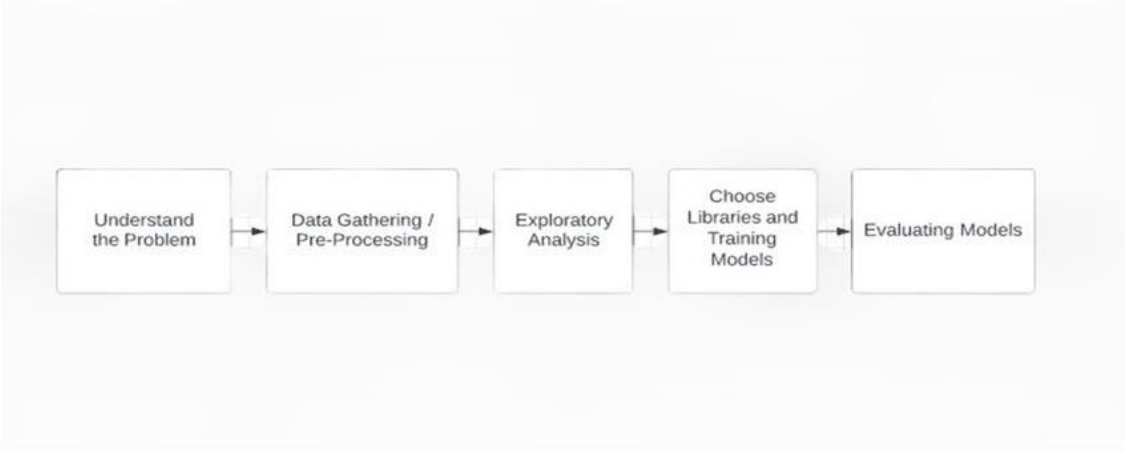


Figure 2. Flowchart of general methods.

The first step is to thoroughly understand the problem. Time series forecasting involves predicting future values based on historical data. In this research, the goal is to predict stock prices, a problem that requires careful consideration of the models to be used. The study employs two prominent models: Long Short-Term Memory (LSTM) and Autoregressive Integrated Moving Average (ARIMA). These models are chosen due to their proven efficacy in handling time series data, particularly in financial contexts.

3.1 Data Collection and Preprocessing:

For stock price prediction, this typically includes daily data, such as opening, closing, high, and low prices, as well as trading volumes. In this study, the dataset used comes from yahoo finance, ensuring it spans a sufficiently long period to capture various market conditions.

Once the data is gathered, the preprocessing phase begins. This phase is crucial because ARIMA models requires the data to be stationary. The statistical features of a time series, such as its mean and variance, are constant across time when it is stationary. Non-stationary data can lead to inaccurate forecasts and misleading results.

The preprocessing steps for ARIMA include:

1. **Testing for Stationarity**: To use ARIMA, the data must be stationary. To verify stationarity, statistical tests such as the Augmented Dickey-Fuller (ADF) are run. The null hypothesis of this test is supported by the non-stationary and unit root nature of the data in Figure 3. When the p-value exceeds the specified significance level of 0.05, Figure 4 null hypothesis does not reject, indicating that the data is not stationary.

$$y_t = c + \beta t + \alpha y_{t-1} + \emptyset \Delta y_{t-1} + e_t$$

where,

- y(t-1) = lag 1 in the time series
- delta Y(t-1) = first variation in the series at time (t-1)

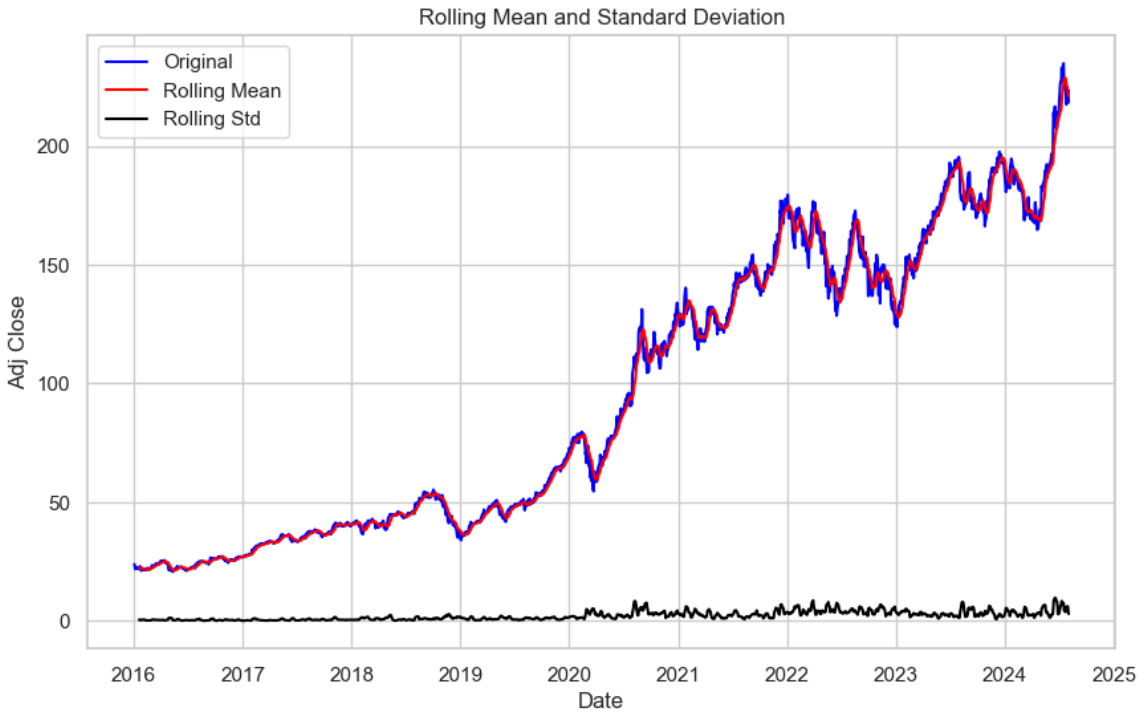


Figure 3: Rolling Mean and Standard Deviation

```
Results of Dickey-Fuller Test
Test Statistic                    0.232925
p-value                           0.974049
No. of lags used                  0.000000
Number of observations used    2159.000000
Critical Value (1%)              -3.433382
Critical Value (5%)              -2.862880
Critical Value (10%)             -2.567483
dtype: float64
```

Figure 4: Results of Dicky-Fuller Test

The Dickey-Fuller test results provided offer insights into the stationarity of time series data. With a high p-value of 0.9740, which is well above the conventional significance levels (0.01, 0.05, or 0.1), we fail to reject the null hypothesis of a unit root. This suggests that time series is likely non-stationary. The test statistic of 0.2329 is greater than all the critical values at 1% (-3.433), 5% (-2.8628), and 10% (-2.567) levels, further confirming the non-stationarity. The test used 2159 observations without any lags (no. of lags used: 0.00). In practical terms, this indicates that time series data exhibits trends or patterns that change over time, rather than fluctuating around a constant mean. For ARIMA modelling or other time series analyses that assume stationarity, it would need to consider differencing or other transformations to make the series stationary before proceeding with further analysis. This non-

stationarity could be due to various factors such as long-term trends, seasonal patterns, or structural changes in the underlying process generating the data.

2. Differencing: By using differencing, the non-stationary data is transformed into stationarity. To stabilise and make the data steady, differencing involves reducing the prior data point from the present one. This helps to eliminate underlying trends and seasonality. The number of times differencing is applied whether once, twice, or more is determined by analysing the results of a stationarity test, as depicted in Figure 5. This test helps decide how many differences are needed to achieve a stable, stationary time series suitable for the ARIMA model.

```
ADF Statistic: -0.9067908279644271
p-value: 0.7857278750618841
Data is not stationary, applying differencing
ADF Statistic: -9.931440601207287
p-value: 2.8159691156484266e-17
Data is stationary after first differencing
```

Figure 5: ADF result after differencing.

1. **Seasonal Adjustment**: The data exhibits seasonality, seasonal differencing may be necessary. This involves subtracting the season specific value of the time series in the previous cycle (e.g. same month last year) shown in figure 6.

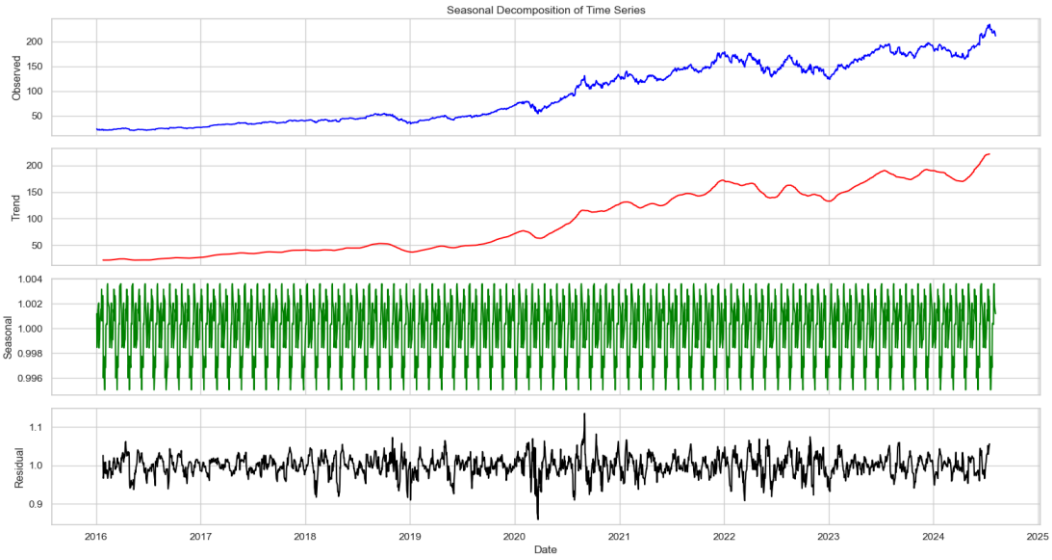


Figure 6: Seasonal Decomposition of Time Series

2. **Data Transformation**: In some cases, applying transformations like logarithms or square roots can stabilize variance and improve model performance. This is particularly useful if the time series exhibits heteroscedasticity.

3. **Divide the dataset into Train and Test Sets**: The dataset is frequently divided into training and test sets to evaluate the model's performance. The test set is used to evaluate the prediction reliability of the ARIMA model, while the training set is used to fit the model.

The preprocessing steps for LSTM include:

1. **Feature scaling:** Feature scaling is vital for the performance of LSTM models as it helps in faster convergence and improves the model's accuracy by ensuring all input features are on a similar scale. We applied the MinMaxScaler from Scikit-Learn to normalize all features. This scaling transforms features to a range between 0 and 1, which is essential for the LSTM.

$$x_{scaled} = \frac{\mathrm{x} - x_{min}}{x_{max} - x_{min}}$$

2. **Calculation of Moving Averages (10, 20, and 50 Days)**: To better understand stock price trends, we calculate moving averages for different periods 10 days, 20 days, and 50 days. These averages help smooth out daily fluctuations and highlight both short-term and long-term trends. The 10-day moving average reacts quickly to recent price changes, the 20-day average balances short-term and long-term insights, and the 50-day average reveals more sustained trends. By applying these moving averages to the 'Close' prices, as shown in Figure 7, we can get clearer insights into the stock's performance and underlying trends.

Figure 7: Moving Average of all the stocks (10,20 and 50 Days)

3. **Sequence creation:** LSTM models require data to be in sequence format. To represent the temporal dependencies in the data, we built sequences. We chose a sequence length of 60 days for input features (X) and the next day's price as the output (y). This means each training example consists of 60 days of data points, predicting the value for the 61st day.

4. **Data splitting:** The dataset was divided into training and testing sets so that we could evaluate the model's performance on unseen data. We allocate 20% of the sequences in the data for testing and the remaining 80% for training. By doing this, it is ensured that the model is trained on a significant amount of the data and is assess on untested data.

5. **Data Shaping for LSTM:** LSTM models expect the input data to be in the shape (samples, time steps, features). We reshaped our data accordingly to fit this requirement.

Once pre-processed, the data is ready for ARIMA modelling. The next steps are to select the appropriate ARIMA model parameters (p, d, and q), where q is the moving average window size (also known as the moving average order), d is the degree of differencing (also referred to as the frequency of difference between the raw data points), and p is the number of observations with lags (also known as the lag order) in the model. Plots of autocorrelation and partial autocorrelation, which help identify the structure of the model, were used to select these values. The ARIMA model's flow diagram is displayed in Figure 8.

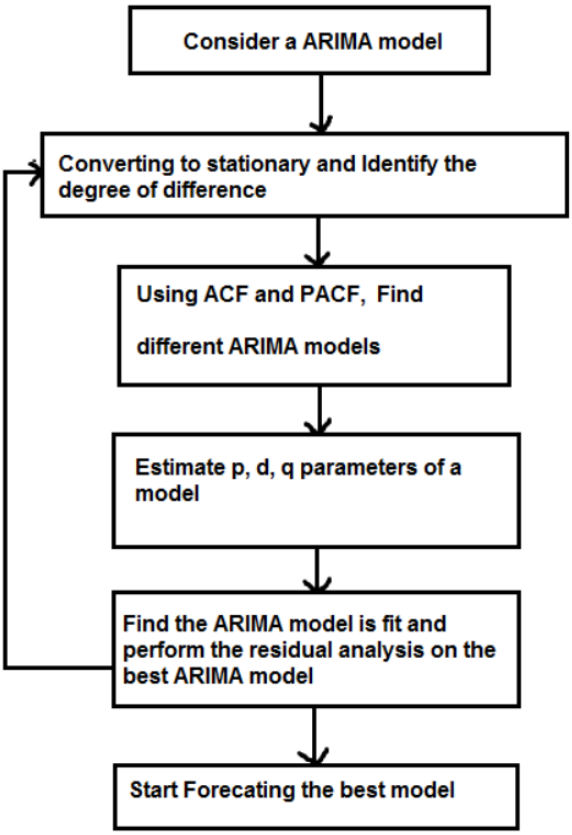


Figure 8: Flow diagram of ARIMA model

3.2 Exploratory Data Analysis (EDA)

The Exploratory Data Analysis (EDA) phase is important for both ARIMA and LSTM models, as it provides insights into the data characteristics and guides the modelling process. While many EDA steps are common for both models, there are some specific analyses that are particularly relevant for each. Here's a detailed explanation of the EDA steps for both ARIMA and LSTM models:

Common EDA Steps for Both Models:

3.2.1 Descriptive Statistics

We begin with calculating summary statistics to get an overall sense of the central tendency and dispersion of the data. This includes the mean, median, minimum, maximum, and standard deviation

values of the closing prices and returns. Additionally, we analysed the distribution of returns using histograms and Q-Q plots to assess normality and identify any skewness or kurtosis in the data.

Summary Statistics and Distribution Analysis:

- Calculated mean, median, and standard deviation for closing prices and returns.
- Analysed the distribution of returns using histograms.
- Used Q-Q plots to check for normality in the return distribution.

3.2.2 Time Series Visualization

Visualizing time series data is crucial to identify seasonality, trends, and any irregular patterns. We plotted the historical adjusted closing prices for each stock, including AAPL, GOOG, MSFT, and AMZN. Additionally, we created subplots to compare price movements across all four stocks simultaneously, which helps in visualizing the comparative performance and correlation.

Visualization of Historical Prices:

- Plotted the adjusted closing prices for each stock.
- Created subplots for side-by-side comparison of price movements across different stocks AAPL, GOOG, MSFT, and AMZN.

3.2.3 Volatility Analysis

Understanding the volatility of stock prices is important for modelling and forecasting. We computed daily returns and plotted them to assess volatility. Furthermore, we calculated the rolling standard deviation of returns, which provides a time-varying measure of volatility and helps in identifying periods of high and low volatility.

Volatility Assessment:

- Computed daily returns to evaluate day-to-day price changes.
- Plotted daily returns to visualize volatility.
- Calculated and visualized the rolling standard deviation of returns to observe changes in volatility over time.

3.2.4 Correlation Analysis

Correlation analysis helps in understanding the relationships between the stock prices and returns. We created correlation matrices and heatmaps to visualize the relationships between the stocks. Additionally, pair plots were performed to examine bivariate relationships between stock returns, which can reveal potential co-movements and dependencies.

Correlation and Relationship Analysis:

- Created correlation matrices to quantify the relationships between stock prices.
- Used heatmaps to visualize these correlations.
- Performed pair plots to examine the bivariate relationships between stock returns.

Specific EDA Steps for ARIMA:

For ARIMA modelling, certain additional steps are necessary to ensure the data meets the assumptions of stationarity:

3.2.5 Stationarity Check

The time series must be stationary for ARIMA models to work. To look for unit roots in the data, we used stationarity tests such the Augmented Dickey-Fuller (ADF) test. Differentiating was used to make the series stationary if it turned out to be non-stationary. Until stationarity was reached, these steps were repeated.

Stationarity Assessment:

- To ensure stationarity, the Augmented Dickey-Fuller (ADF) test was performed.
- Applied differencing to the time series if non-stationarity was detected.

3.2.6 Autocorrelation and Partial Autocorrelation

The proper ARIMA model parameters were identified by utilising the Autocorrelation Function (ACF) and Partial Autocorrelation Function (PACF). Based on these charts, one may determine the correct sequence for the moving average (q) and autoregressive (p) components of the ARIMA model.

ACF and PACF Analysis:

- Based on the plot in figure 9, Plotted Autocorrelation Function (ACF) and Partial Autocorrelation Function (PACF) graphs.
- Used these plots to identify potential AR and MA terms for the ARIMA model.

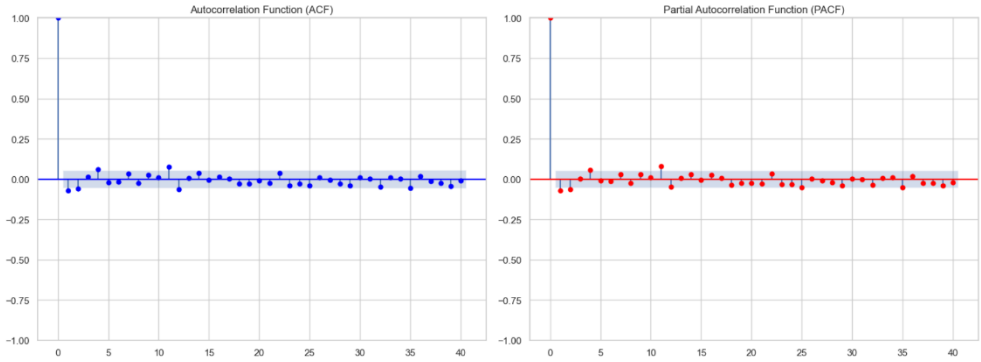


Figure 9: ACF and PACF for SARIMA

3.2.7 Seasonal Decomposition

Seasonal decomposition helps in identifying and isolating the seasonal, trend, and residual components of the time series. This step is particularly useful for determining if a seasonal ARIMA model (SARIMA) will be more appropriate.

Seasonal Analysis:

- **Trend Component**: The results seem to be generally trending upward, based on the plot shown in Figure 10. We may nevertheless include an autoregressive (AR) component into our model to adjust for pricing even in the absence of knowledge about the precise long-term patterns that affect it.

- **Seasonal Component:** We can see that there is definite seasonality in the data, causing prices to fluctuate by 1.004 over the course of a year.
- **Residual Component:** The randomness in the data makes this plot to appear. We know our model will benefit from the addition of a moving average (MA) piece to smooth out these shocks because prices are susceptible to unpredictable "shocks."

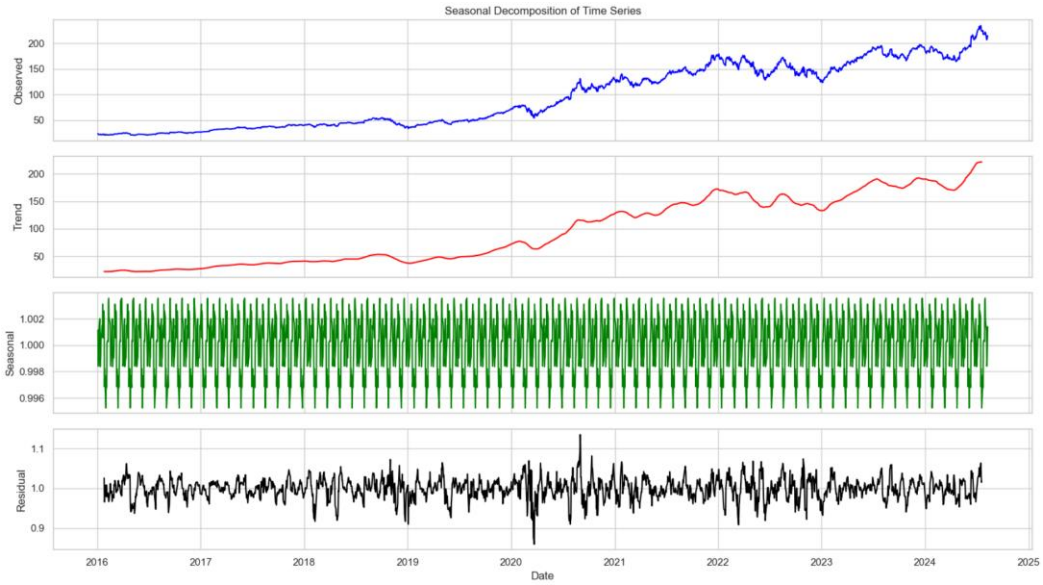


Figure 10: Seasonal Decomposition of Time Series

Specific EDA Steps for LSTM:

3.2.8 Sequence Visualization

To prepare for training the LSTM model, we created visualizations of the input sequences over a specified window, typically 60 days. This step involves plotting segments of the time series data that will be applied as inputs to the LSTM model. By visualizing these sequences, we acquire understanding into the historical trends and pattern present in the data. This helps in understanding the patterns that the LSTM model will be learning from.

Sequence Visualization:

- Created visualizations of 60-day input sequences for LSTM training.
- Observed seasonal trends, spikes, drops, and overall movement patterns.

3.2.9 Feature Correlation

Analysing correlations between different features such as Open, High, Volume, Low, and Close is critical for identifying which features are most relevant for the LSTM model across different stocks like AAPL, GOOG, MSFT and AMZN. This analysis typically involves computing the correlation matrix, which quantifies the degree of linear relationship between each pair of features shown in figure 11. By creating heatmaps or scatter plots of these correlations, we can visualize how closely related each feature is to the target variable and decide which features to include in the model to improve its predictive power.

Feature Correlation:

- Computed correlation matrix for features (Open, High, Low, Close, Volume).
- Created heatmaps to visualize relationships between features.
- Identified features most relevant to the LSTM model.

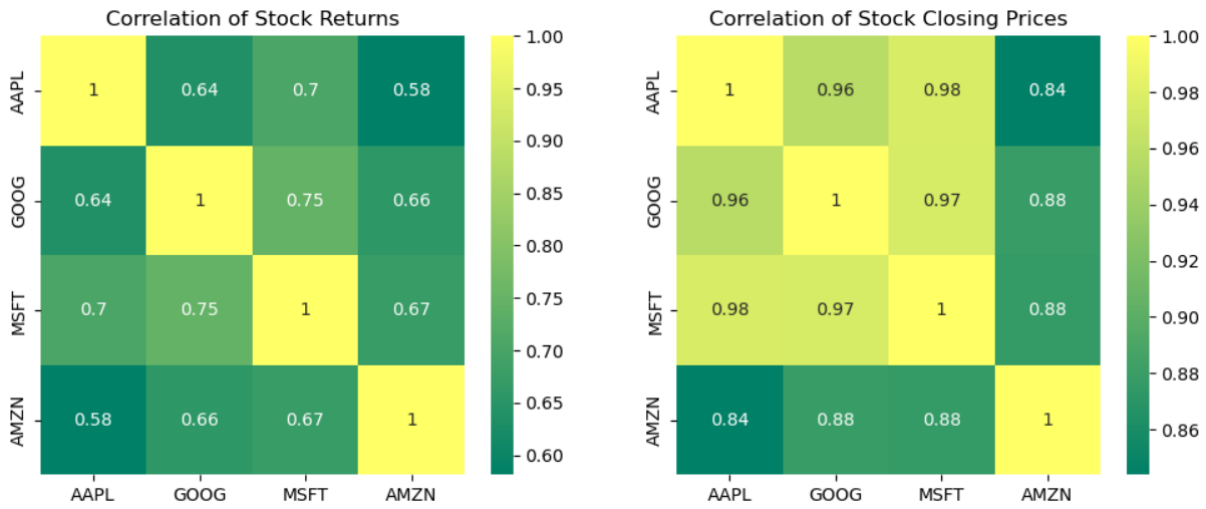


Figure 11: Heatmap of correlation

3.2.10 Lag Analysis

Examining the connection between the current price and its lag values at different time steps is known as lag analysis. Understanding the temporal dependencies in the data requires completing this stage. By plotting these correlations, we can identify significant lag periods that the LSTM model should consider. If certain lagged prices show strong correlations with the current price, it indicates that past prices have a substantial influence on the present price shown in figure 12.

Lag Analysis:

- Examined correlation between current price and lagged prices at various time steps.
- Plotted correlations to identify significant lag periods.
- Determined lagged values to incorporate into the LSTM model.

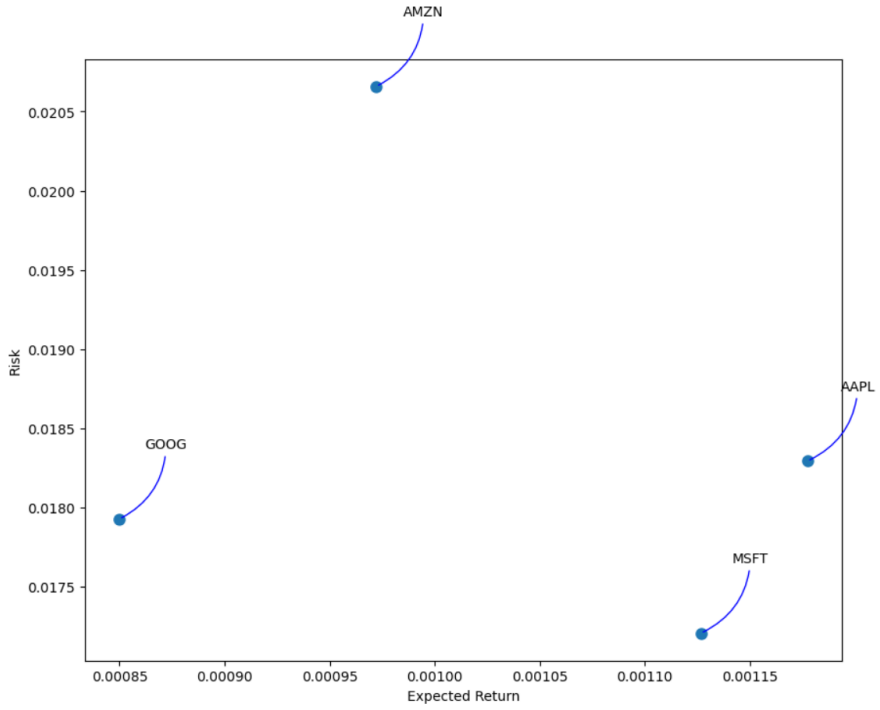


Figure 12: Scatter Plot of Return vs Risk

3.2.11 Normalization Visualization

For LSTM models, normalisation is an important preprocessing step because scaled inputs improve their performance. We plot the stock price data before and after normalisation to see how it affects the data. Z-score normalisation and Min-Max scaling are two popular normalisation methods. We can make sure that all input characteristics are on the same scale by visualising the normalised data, which will help in the LSTM model's faster convergence and improved performance.

Normalization Visualization:

- Visualized stock price data before and after normalization.
- Applied Min-Max scaling and Z-score normalization techniques.
- Ensured for better model performance, all input features are scaled similarly.

These EDA steps provide a comprehensive understanding of the stock price data, helping to inform the modelling decisions for both ARIMA and LSTM approaches. The insights gained from this analysis guide the choice of model parameters, feature selection, and data preprocessing steps, ultimately contributing to more accurate and reliable stock price predictions.

3.3 Model Implementation and Libraries Description

In this part, overview description of the libraries has used and the implementation steps for both the ARIMA and LSTM models.

3.3.1 ARIMA Model

Libraries Used: The ARIMA model implementation relied heavily on the statsmodels library, which provides classes and functions for predicting a wide range of statistical models. Additionally, the pmdarima library was used to facilitate automatic ARIMA order selection, making the process of model specification more efficient.

Model Training: To determine the best option for ARIMA model, the auto_arima function from the pmdarima library was employed. This function automates the process of fitting the ARIMA model by iterating through different combinations of p, d, and q parameters and choosing the ideal pair according to a chosen criterion, such as AIC (Akaike Information Criterion).

Once the suitable p, d, and q values were determined, the ARIMA model was fitted using the ARIMA class from the statsmodels library. The model fitting process involved estimating the parameters of the ARIMA model that is most effective describe the different patterns in the training data.

Forecasting: After fitting model, forecasts for the test period were generated. The ARIMA model produced point forecasts as well as confidence intervals, providing insights into the expected future values along with the uncertainty associated with the predictions.

3.3.2 LSTM Model

Libraries Used: The implementation of the LSTM model utilized several powerful libraries:

- **TensorFlow** and **Keras** were utilised for developing and training the LSTM neural network. These libraries are widely used for deep learning tasks and provide high-level APIs for defining and training models.
- **Scikit-learn** was employed for data preprocessing tasks such as normalization and for calculating evaluation metrics.

Data Preparation: Data preparation involved normalizing the stock price data to ensure that all input features are on a similar parameter scale, which is crucial for the effective training of neural networks. The MinMaxScaler from scikit-learn was used for this purpose, scaling the data to a range between 0 and 1.

Sequence of 60 days were created to function as input data for the LSTM model. Each sequence contained 60 days of historical data and the corresponding target value was the stock price on the 61st day. This approach helped the LSTM model understand patterns and temporal connections in the data.

Model Architecture: The LSTM model was implemented as an organised model using the Sequential API from Keras. The architecture consisted of:

- Two LSTM layers: The goal of these layers is to fetch patterns and long-term dependencies from the sequential data. Sequences were returned by the first LSTM layer, allowing the second LSTM layer to handle them.
- Two Dense layers: The first Dense layer with ReLU activation function helped in learning non-linear combinations of the input features. The final Dense layer with a single neuron and linear activation function produced the output, which is the predicted stock price.

The architecture described combines the strengths of LSTM layers with Dense layers to create a powerful model for stock price prediction. When analysing sequential data, the two LSTM layers interact to capture complex temporal patterns and long-term dependencies. This is essential for understanding stock price fluctuations that could be impacted by past trends and occurrences. By returning sequences from the first LSTM layer, the model allows for deeper processing of temporal information in the second LSTM layer. The subsequent Dense layers serve to interpret and transform the LSTM outputs. The model can develop complex correlations between features due to the non-linearity created by the first Dense layer with ReLU activation. The output layer, which is the last Dense layer with linear activation, generates a single value forecast for the stock price. The model's ability to capture the sequential nature of stock data as well as the complex, non-linear interactions between different elements influencing stock prices is made possible by the combination of layers, which may result in more accurate predictions.

Model Training: The Mean Squared Error (MSE) loss function and the Adam optimiser were used to train the model. Because of its effectiveness and versatility, the Adam optimiser was selected to be used in deep learning model training. The average squared difference between the actual and forecasted stock prices was calculated using the MSE loss function.

With a batch size of 1, the model was trained over 9 epochs. Through backpropagation, the model was trained to minimise the loss function by modifying the weights and biases. To increase prediction accuracy, the model's parameters had to be updated when the input sequences were put into it after the loss was calculated.

3.4 Evaluation of Models

A key initial phase to assessing the precision and dependability of the ARIMA and LSTM models for stock price forecasting is to look at their performance. The measurements and evaluation techniques for both models are explained in this section.

3.4.1 ARIMA Model Evaluation

Metrics Used: The performance of the ARIMA model was assessed using several standard metrics:

- **Mean Absolute Error (MAE):** This provides a clear indication of the accuracy of the model by measuring the average magnitude of the errors between the anticipated and actual values.

$$\text{MAE} = \frac{\sum_{i=1}^{n} |x_i - \hat{x}_i|}{n}$$

Where.

MAE = mean absolute error
x_i = The real value of the i^{th} observation.
$\hat{x}_i$ = The predicted value of the i^{th} observation produced by the ARIMA model.
n = The number of observations or samples in the dataset.

- **Mean Squared Error (MSE):** This measure highlights any significant variations between expected and actual values by squaring the errors before averaging, which gives greater weight to larger errors.

$$\text{MSE} = \frac{1}{N}\sum_{i=1}^{n}(x_i - \hat{x}_i)^2$$

Where,

MSE = mean squared error
$n =$ The number of observations or samples in the dataset.
x_i = The real value of the i^{th} observation.
$\hat{x}_i$ = The predicted value of the i^{th} observation produced by the ARIMA model.

Root Mean Squared Error (RMSE): The RMSE provides a more comprehensible estimate of average error magnitude because it is the square root of the MSE and is expressed in the same units as the original data.

$$\text{RMSE} = \sqrt{MSE}$$

- **Mean Absolute Percentage Error (MAPE):** This represents the average error as a percentage of actual values, offering a normalized measure of prediction accuracy.

$$\text{MAPE} = \frac{1}{n}\sum_{i=1}^{n}\left|\frac{x_i - \hat{x}_i}{x_t}\right| x\ 100$$

Where,

n = number of times the summation iteration happens
x_i = The real value of the i^{th} observation.
$\hat{x}_i$ = The predicted value of the i^{th} observation produced by the ARIMA model.

Evaluation Process: After generating forecasts for the test period using the ARIMA model, the predicted values were compared to the real stock prices. The evaluation metrics were then computed to quantify the model's performance. These metrics helped identify how closely the model's predictions matched the observed data and highlighted areas where the model might need improvement.

3.4.2 LSTM Model Evaluation

Metrics Used: The LSTM model was determined using the same set of metrics as the ARIMA model to allow for a direct comparison:

- **Mean Squared Error (MSE)**
- **Mean Absolute Error (MAE)**
- **Root Mean Squared Error (RMSE)**
- **Mean Absolute Percentage Error (MAPE)**

Evaluation Process: Predictions for the test period were produced once the LSTM model had been trained. The actual stock prices were then contrasted with these projected values. The evaluation metrics were computed to evaluate the LSTM model's accuracy. A visual examination of the predictions was also carried out due to the complex structure of neural networks to better understand the model's performance and spot any possible overfitting or underfitting problems.

When evaluating how well the model predicts the target values, the loss function is a key factor. Our model's loss function was Mean Squared Error (MSE). Since MSE determines the squared difference between the average of the two actual and shown values, it is especially useful for regression assignments as it assigns more weight to larger errors. Reducing forecast errors is the reason for making this choice.

Evaluation Process: To visually evaluate the performance of the model over the training period, we plotted the loss function's behaviour across epochs. The loss curve, which tracks the MSE values as the model trains, provides valuable insights into the model's learning process and effectiveness.

Diagram Summary:

The loss curve diagram illustrates the following:

1. **Initial High Loss:** At the start of training, the loss value is typically high. This is due to the initial randomness in the model's parameter initialization, which results in less accurate predictions.
2. **Decreasing Loss:** As training progresses to reduce the loss. The optimiser modifies the model's parameters. This is reflected in the downward slope of the loss curve, indicating that the model is improving its predictions and learning from the data.
3. **Plateauing Loss:** Eventually, the loss curve begins to flatten, showing that further training produces minimal reductions in loss. This plateau indicates that the model has reached a point of convergence where it has effectively learned from the data, and additional training may not significantly improve performance.

We can verify that the model is learning efficiently and generalising successfully by looking at the loss curve. The loss numbers show a gradual drop, which indicates that the model can adjust and improve its predictions with time. The plateau indicates that the model is almost operating at peak efficiency with the available data. As a crucial evaluation tool, this diagram makes sure that the implementation follows expected performance standards and gives an easily understood visual representation of the model's training dynamics.

Comparative Analysis

By using the same set of evaluation metrics, we can directly compare the results of the ARIMA and LSTM models. This comparative analysis provided insights into which model better captured the underlying patterns in the stock price data and offered more reliable forecasts.

- **ARIMA Model:**
  a. **Strengths:** Effectively models linear time series with trends and seasonal patterns. Handles both stationary and non-stationary data through differencing. Provides interpretable results for forecasting and analysis.
  b. **Weaknesses:** Limited ability to get non-linear relationships in data. Assumes constant variance over time, which may not hold for all datasets. Requires careful parameter selection and can be sensitive to outliers.

- **LSTM Model:**
  a. **Strengths:** Excels at getting long-term dependencies in sequential data. Capable of learning complex non-linear patterns. Effective for handling vanishing gradient problems in traditional RNNs. Versatile for various time series tasks, including prediction and classification.
  b. **Weaknesses:** Requires large amounts of data for optimal performance. Computationally intensive, leading to longer training times. Limited interpretability due to its complex architecture. Prone to overfitting, especially with smaller datasets. Hyperparameter tuning can be challenging and time-consuming.

The evaluation of both models provided a comprehensive understanding of their performance in stock price forecasting. The ARIMA model, with its focus on linear patterns, served as a robust baseline. In contrast, the LSTM model's ability to learn from complex, sequential data offered potentially higher accuracy in capturing the nuances of stock price movements. The comparative analysis based on standard evaluation metrics enabled us to select the most suitable model for our forecasting objectives.

4. Implementation

Both the LSTM and ARIMA implementations made extensive use of pandas for data manipulation and numpy for numerical operations. These libraries provided efficient data structures and operations necessary for handling large time series datasets. The visualization libraries matplotlib and seaborn were crucial in creating informative and visually appealing plots, allowing for better interpretation of the results. The statsmodels library played a central role in the ARIMA implementation, providing the necessary functions for statistical tests, model fitting, and diagnostics. For the LSTM implementation, Keras and TensorFlow were indispensable, offering a high-level API for developing and train deep learning models while leveraging efficient computational backends.

4.1 LSTM Model Implementation

The LSTM model is a type of recurrent neural network particularly adept at learning long-term dependencies, was implemented using the Keras with TensorFlow library as the backend. This choice was made due to Keras's user-friendly API and TensorFlow's robust computational capabilities, making it well-suited for deep learning tasks. This model was a multi-step process involving data preparation, model architecture design, training, and evaluation. Below is a detailed explanation of each phase of the LSTM implementation, complemented by placeholders for relevant figures.

4.1.1 Data Preparation

a. Historical Data Collection

First, we collected historical stock data from Yahoo Finance for the four largest technical companies: Apple (AAPL), Microsoft (MSFT), Amazon (AMZN), and Google (GOOG). Figure 13 shows how this data source was chosen due to its large historical coverage and dependability. 'Adjusted Close' prices

were selected as the objective variable because they give a more accurate evaluation of the stock's value over time by reflecting company actions like stock splits and dividends.

Figure 13: Timeseries plot of adjusted closing prices

b. Normalization

Using the MinMaxScaler function from the scikit-learn library, normalisation was applied to make sure the data scale was not skewing the model's performance. By changing each feature value to a specified range, usually between 0 and 1, this scaler modifies the data. The MinMaxScaler prevents variables with bigger magnitudes from excessively affecting the model by rescaling the data in this method. This kind of scaling is essential because, in the absence of it features with higher values might control the learning process, potentially leading to biased or ineffective model performance. Additionally, normalizing the data facilitates faster convergence during training by ensuring that the optimization algorithms operate more efficiently. This standardization of data inputs helps the model learn more effectively and achieve better performance by treating all features with equal importance and enhancing the training process's general stability and speed.

c. Data Splitting

90% of the dataset was allocated for train and 10% for test after it was divided into train and test dataset. This split ratio was chosen to provide the model with ample historical data for training while preserving a sufficient portion of the data for evaluation from 2162 rows data split into 1946 to training data and 216 to testing data.

d. Sequence Creation

One important task was to create input sequences. Sequences of 60 days were used as input characteristics for the LSTM model, with the 61st day acting as the prediction objective. This choice assumed that two months of historical data would provide enough context to predict the next day's stock

price. These sequences were generated by sliding a 60-day window over the dataset, resulting in a three-dimensional array with dimensions corresponding to the number of samples, time steps, and features shown in figure 14.

```
[array([0.01768088, 0.01457112, 0.01220345, 0.00719721, 0.00779796,
       0.00964733, 0.01133178, 0.00830448, 0.01081348, 0.00799821,
       0.00744459, 0.00759771, 0.00702053, 0.01305157, 0.01071926,
       0.01136712, 0.00362806, 0.00441727, 0.00824558, 0.00717366,
       0.00487668, 0.00707942, 0.00737391, 0.00433482, 0.00550099,
       0.00547742, 0.0046293 , 0.00395788, 0.00429948, 0.00742102,
       0.00916438, 0.00697341, 0.00671426, 0.00770372, 0.00512405,
       0.00678494, 0.00756238, 0.00773907, 0.00747993, 0.01200321,
       0.01226235, 0.01314581, 0.0149245 , 0.01358165, 0.01259218,
       0.0126982 , 0.01275709, 0.01404105, 0.0143473 , 0.01677386,
       0.0184112 , 0.01821095, 0.0183523 , 0.01834053, 0.01929465,
       0.01859966, 0.01805782, 0.01749241, 0.02042548, 0.02264   ])]
[0.02196857174477078]

[array([0.01768088, 0.01457112, 0.01220345, 0.00719721, 0.00779796,
       0.00964733, 0.01133178, 0.00830448, 0.01081348, 0.00799821,
       0.00744459, 0.00759771, 0.00702053, 0.01305157, 0.01071926,
       0.01136712, 0.00362806, 0.00441727, 0.00824558, 0.00717366,
```

Figure 14: Sequence data for every 60 days

4.1.2 Model Architecture

With TensorFlow working as the backend, the LSTM model was built using the Keras package. Keeping complexity and generalisation in check, the architecture was created to recognise the data's temporal dependencies. The components of the model were:

1. **LSTM Layer (128 units, returning sequences):** This initial LSTM layer is responsible for processing the input sequences of stock price data. The "128 units" refers to the number of LSTM cells in this layer, which determines the layer's capacity to learn and store information from the sequences. By "returning sequences," this layer outputs the full sequence of processed data for each time step, rather than just the final output. This allows subsequent layers to get temporal dependencies and relationships with entire sequence of inputs, which is crucial for understanding complex patterns in time series data.
2. **LSTM Layer (64 units, not returning sequences):** The second LSTM layer further refines the features extracted by the first LSTM layer. With "64 units," this layer has fewer cells compared to the first layer, which helps in condensing and abstracting the sequence information into a more compact form. By "not returning sequences," this layer outputs only the final state of the processed sequence, summarizing the temporal dependencies into a single vector for each input sequence. This output serves as a compressed representation of the sequence, capturing the most relevant information for prediction.
3. **Dense Layer (25 units):** Following the LSTM layers, this fully connected Dense layer with "25 units" is designed to learn higher-level representations from the features provided by the previous layers. Each unit in this Dense layer processes inputs from all units in the preceding layer, allowing it to combine and refine the information into more abstract features. This layer helps in identifying complex patterns and relationships which are essential for making accurate predictions.
4. **Dense Layer (1 unit):** The final Dense layer having "1 unit" produces the model's output prediction. The single unit provides the forecasted stock price based on the higher-level features extracted and processed by the previous layers. The output is a continuous value that represents the predicted stock price for the next time step or sequence, depending on the model's design.

This layered approach allows the model to progressively improves its understanding of the input data, which result in a more accurate prediction shown in figure 15. This architecture, with its decreasing number of units in subsequent layers, allows the model to provide a single prediction value by continually enhancing its representations of the input data

Model: "sequential"

Layer (type)	Output Shape	Param #
lstm (LSTM)	(None, 60, 128)	66,560
lstm_1 (LSTM)	(None, 64)	49,408
dense (Dense)	(None, 25)	1,625
dense_1 (Dense)	(None, 1)	26

Total params: 117,619 (459.45 KB)
Trainable params: 117,619 (459.45 KB)
Non-trainable params: 0 (0.00 B)

Figure 15: LSTM Model Architecture

4.1.3 Model Compilation and Training

The Adam optimiser, which is popular for its flexible learning rate capabilities, was used to construct the model. The choice the loss function is Mean Squared Error (MSE) was made because of its suitability for regression issues such as stock price prediction. Mean Absolute Error (MAE) has been included as a metric to provide another approach to assess the model's performance.

The training process involved 9 epochs and a batch size of 1. This configuration allows the model to update its weights after each training example, potentially leading to faster convergence, especially given the sequential format of the data. The choice of 9 epochs shown in figure 16 was made to balance between sufficient learning and avoiding overfitting, though in practice, this number could be adjusted in understanding the model's learning curve.

```
Epoch 2/9
1889/1889 ━━━━━━━━━━━━━━━━━━━━ 14s 8ms/step - MAE: 0.0103 - loss: 1.9185e-04
Epoch 3/9
1889/1889 ━━━━━━━━━━━━━━━━━━━━ 14s 8ms/step - MAE: 0.0101 - loss: 1.8594e-04
Epoch 4/9
1889/1889 ━━━━━━━━━━━━━━━━━━━━ 14s 8ms/step - MAE: 0.0104 - loss: 1.9689e-04
Epoch 5/9
1889/1889 ━━━━━━━━━━━━━━━━━━━━ 14s 7ms/step - MAE: 0.0110 - loss: 2.1952e-04
Epoch 6/9
1889/1889 ━━━━━━━━━━━━━━━━━━━━ 14s 7ms/step - MAE: 0.0100 - loss: 1.8470e-04
Epoch 7/9
1889/1889 ━━━━━━━━━━━━━━━━━━━━ 14s 7ms/step - MAE: 0.0093 - loss: 1.6636e-04
Epoch 8/9
1889/1889 ━━━━━━━━━━━━━━━━━━━━ 14s 7ms/step - MAE: 0.0098 - loss: 1.7983e-04
Epoch 9/9
1889/1889 ━━━━━━━━━━━━━━━━━━━━ 14s 7ms/step - MAE: 0.0093 - loss: 1.6507e-04

<keras.src.callbacks.history.History at 0x22affcc67d0>
```

Figure 16: LSTM Model Training Loss and MAE Over Epochs

The LSTM model's training process was closely observed due to the behaviour of the loss function, a crucial sign of the model's learning performance. The loss function used during the model's training phases was the Mean Squared Error (MSE), a popular choice for regression tasks that penalises higher prediction errors more severely. The loss at the beginning of training was naturally high because the

model's parameters were first created at random. The Adam optimiser changed the model's parameters during training to minimise the loss, which caused the loss value to gradually drop over the period of the epochs. This drop in loss showed that the model was improving its predictions over time by effectively learning from the data.

Dropout layers have been built into the architecture to prevent against overfitting and ensure the model functioned well when used with fresh data. These dropout layers randomly deactivated a portion of neurones forcing the model to learn during training to develop more robust and universal features. In the context of time series forecasting, the model's ability to consistently lower the loss function during training a sign that it had identified underlying patterns in the data was the most significant outcome.

As the training continued, the loss curve began to plateau, indicating that the model was approaching its optimal performance shown in figure 17. This plateau suggested that the model had learned the most it could from the data, and further training would likely lead to diminishing returns. The observed behaviour of the loss function, from its initial high value to its gradual decline and eventual plateau, provided clear evidence of the model's effective learning process and its convergence towards an optimal solution. This behaviour was visually represented in the loss curve, which demonstrated the model's training dynamics and served as a key validation of the implementation's success.

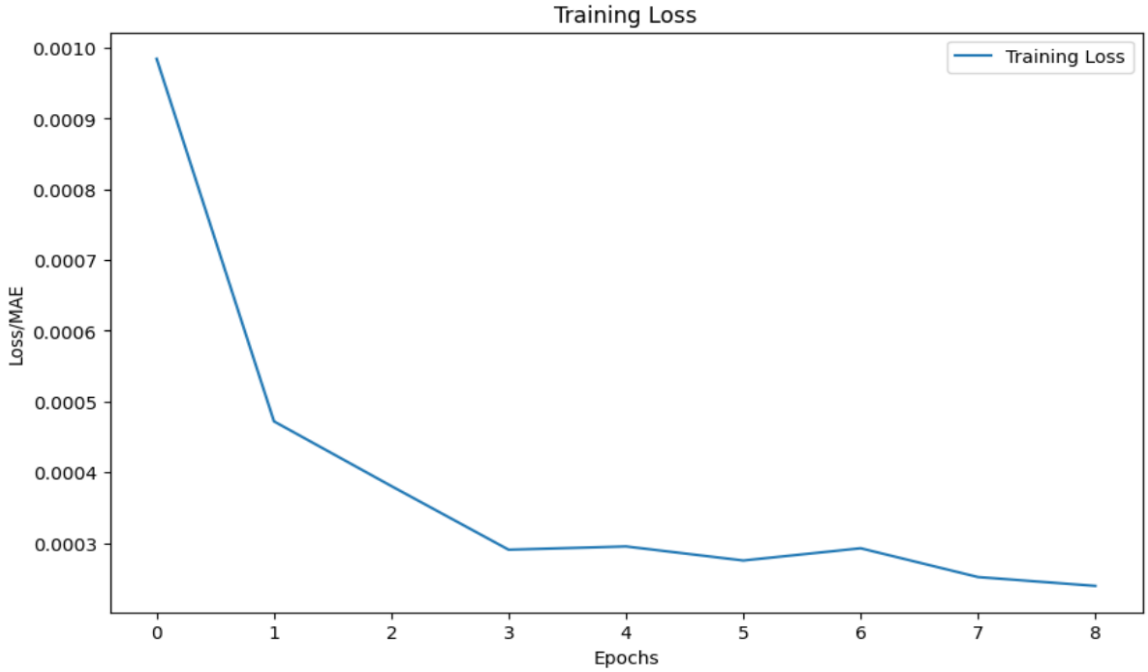


Figure 17: Training loss curve for LSTM Model

4.1.4 Prediction and Evaluation

The model was used to generate predictions on the test dataset once it had been trained. The MinMaxScaler fitted on the training data was then used to reverse translate these predictions back to the original scale. To interpret the findings in relation to actual stock values, this step is essential.

The model's effectiveness was assessed using the Root Mean Squared Error (RMSE) between the actual and projected stock prices. The Root Mean Square Error (RMSE) quantifies the average size of prediction mistakes in the same unit as the target variable, which in this case is dollars. A lower RMSE is indicative of better model performance.

Result	LSTM Model
MSE	1.8104e-04

MAE	0.0099
RMSE	2.823308
MAPE	1.330123

Table 1: LSTM Model Performance Metrics

4.1.5 Visualization

Matplotlib was used to generate a plot in figure 18 that compares the actual stock prices with the model's predictions to provide an accurate visual representation of the model's performance. This plot included the training data, actual test data, and predicted values, allowing an extensive evaluation of how accurately the model represented the changes in the stock's price.

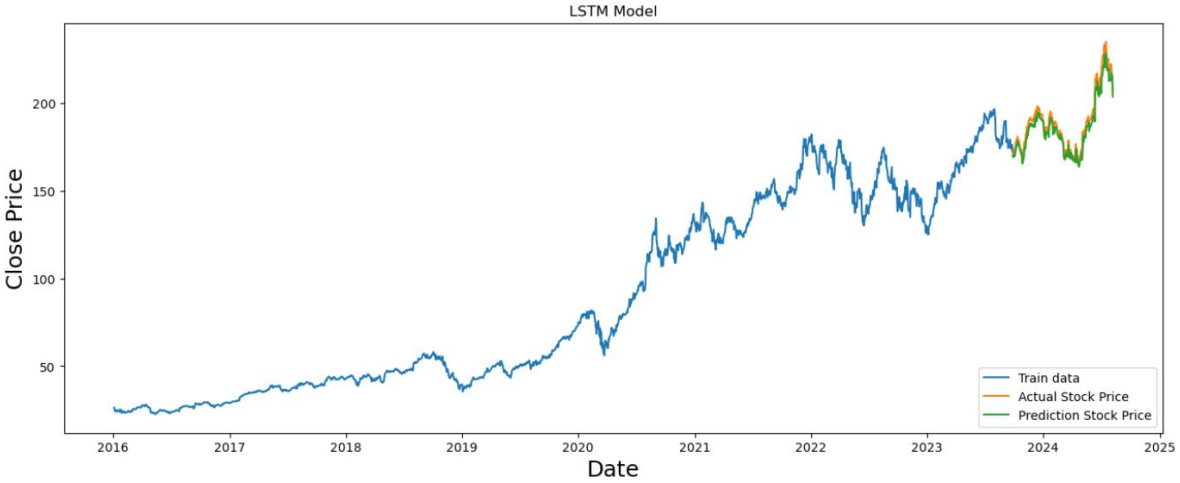


Figure 18: LSTM Model Predictions vs Actual Stock Prices

4.2 ARIMA Model Implementation

To time series forecasting, the ARIMA (Autoregressive Integrated Moving Average) model is a helpful technique that integrates differencing, moving average (MA), and autoregressive (AR) components. This method was used with the help of the statsmodels package, which provides several statistical modelling tools. Finding the ideal model configuration was made easier by using the pmdarima library to automate the ARIMA parameter selection process.

4.2.1 Data Preprocessing

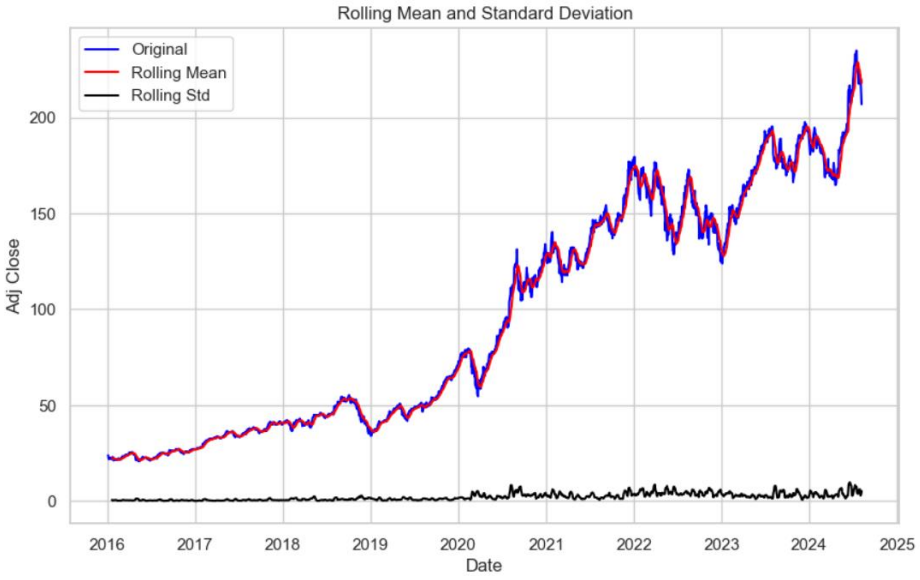
The ARIMA model's implementation began with the extraction of the 'Adjusted Close' prices from historical stock data for the selected technology company name Apple. The 'Adjusted Close' price was chosen as it accounts for corporate actions like stock splits and dividend payments, providing a more accurate reflection of the stock's value shown in figure 19.

To stabilize variance and prepare the data for ARIMA modelling, a logarithmic transformation was applied to the 'Adjusted Close' prices. This transformation is particularly useful in financial time series analysis as it helps manage heteroscedasticity, this is the tendencies of the time sequence' variance to vary over time.

Figure 19: Adj Close vs Date plot of Apple company

4.2.2 Stationarity Testing and Differencing

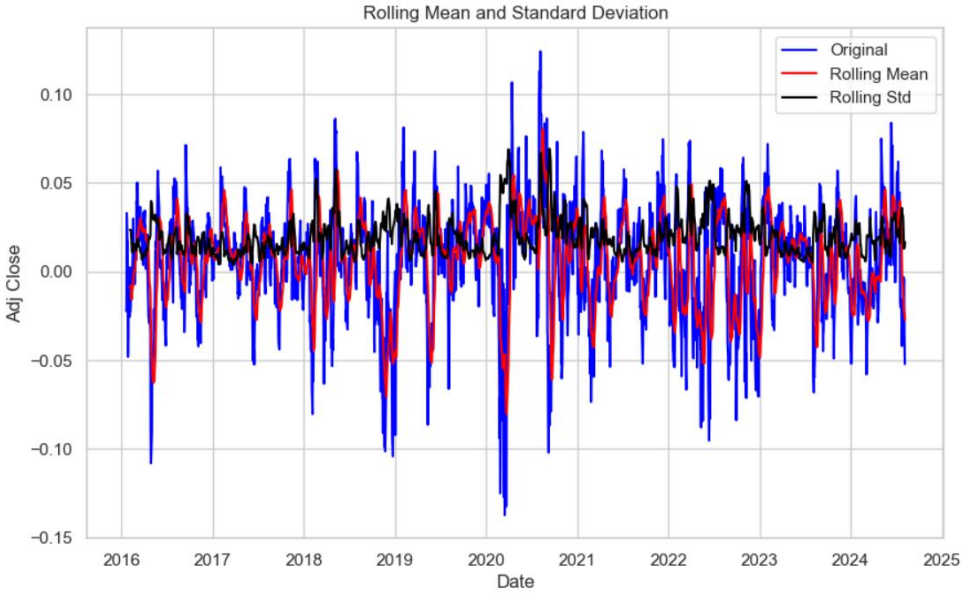
It is essential to confirm that the time series data is stationary. A stationary time series is one in which the mean and variance, together with other statistical parameters, remain constant throughout the duration of the record. Using the Augmented Dickey-Fuller (ADF) test, stationarity was ensured. This test tests a null hypothesis related to non-stationarity in the time series, as shown by the Figure 20, unit root.




```
Results of Dickey-Fuller Test
Test Statistic                  -0.133128
p-value                          0.946037
No. of lags used                 0.000000
Number of observations used    2161.000000
Critical Value (1%)             -3.433380
Critical Value (5%)             -2.862878
Critical Value (10%)            -2.567482
dtype: float64
```

Figure 20: Result of Dicky-Fuller Test

The ADF test results indicated non-stationarity, differencing was applied. To eliminate the patterns and seasonality shown in figure 21, differencing involves deducting the prior data from the present observation. This process can be repeated if necessary and is essential for transforming the series into a stationary one.




```
Results of Dickey-Fuller Test
Test Statistic                -9.659636e+00
p-value                        1.364438e-16
No. of lags used               1.100000e+01
Number of observations used    2.139000e+03
Critical Value (1%)           -3.433411e+00
Critical Value (5%)           -2.862892e+00
Critical Value (10%)          -2.567490e+00
dtype: float64
```

Figure 21: Result of ADF test after differencing

The figure 21 above illustrates the time series data before and after applying differencing, highlighting how the transformation helps in achieving stationarity.

4.2.3 Data Selection

There are two distinct subsets of the dataset: the train and test datasets shown in figure 22. More specifically, 80 percent of the data were reserved for training the model, and the rest 20 percent were reserved for evaluation of performance. This division was purposefully intended to achieve a balance between the necessity for a large training set and the need for enough data to evaluate the model's effectiveness. From the 2,162 rows, 1,729 rows were allocated to the training set, providing the model with a solid historical dataset to enhance its prediction abilities.

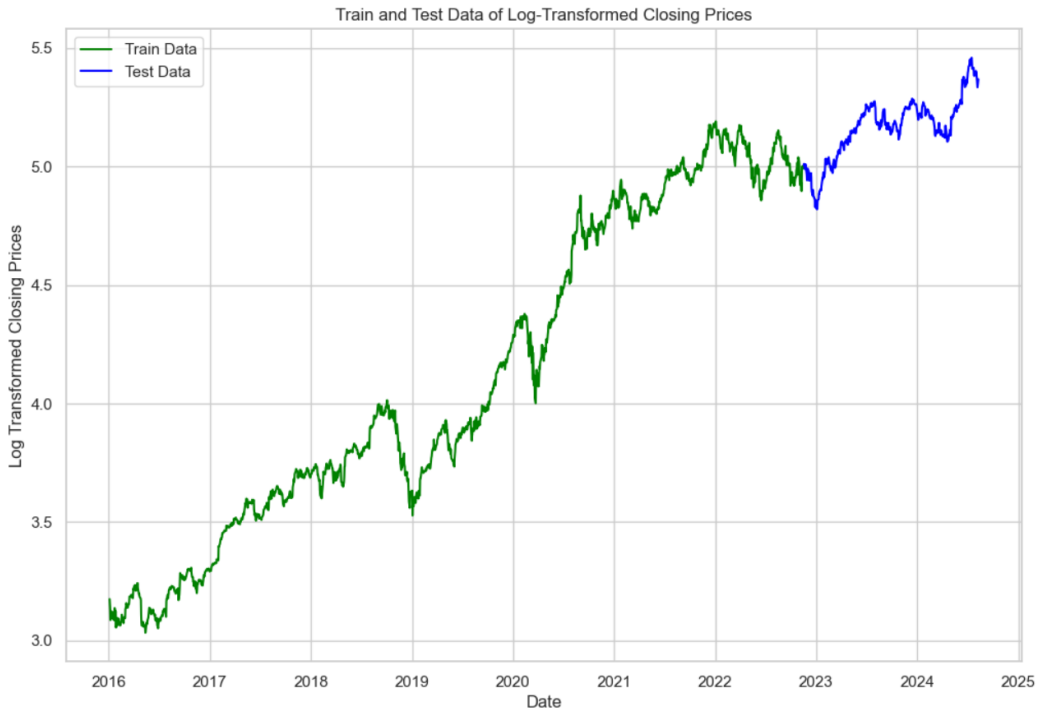


Figure 22: Train and test dataset

4.2.4 Model Selection

Choosing the best values for the three important parameters in ARIMA modelling p, d, and q is one of the biggest issues. Here, d indicates the time series must be stationary for differencing to be used, q represents the moving average component's order, and p refers the autoregressive term's order. This procedure was made simpler by using the auto_arima function from the pmdarima package, as Figure 23 shows. This method automates parameter selection by doing a grid search over a range of possible values for p, d, and q, respectively. The parameters that were selected for our model were (7, 2, 7), as it was found to have the best fit after trying out other combinations. However, the grid search also revealed that the smallest values for these parameters were (0, 2, 0), suggesting that while the best-performing model used higher values for p and q, the model could still perform adequately with simpler parameter settings.

In configuring the model function, several parameters were carefully selected to guide the ARIMA model selection process. The function starts with initial values for the autoregressive and moving average terms and explores a specified range for these parameters to identify the optimal configuration. The continuity of the time series is set to a particular value, indicating the periodic cycle of the data. Differencing is used to convert the series stationary, with the function also assessing if alternative differencing values might be more effective. Since the data does not exhibit seasonal patterns, the model is set to ignore seasonality. Options are included to show the progress of the model fitting process and to handle any errors or warnings that may arise during the fitting. A stepwise search method is employed to incrementally explore different parameter combinations efficiently. This configuration helps in automating the process of finding the best ARIMA model by streamlining parameter selection and ensuring a thorough and effective search.

The auto_arima function fits multiple ARIMA models by conducting a grid search over possible parameter combinations, then choosing the finest model based on the Akaike Information Criterion (AIC). By finding a model that achieves a balance among complexity and goodness of fit, this method reduces the chance of overfitting.

```
Performing stepwise search to minimize aic
 ARIMA(0,2,0)(0,0,0)[0] intercept   : AIC=-4409.684, Time=0.29 sec
 ARIMA(1,2,0)(0,0,0)[0] intercept   : AIC=-5177.873, Time=0.07 sec
 ARIMA(0,2,1)(0,0,0)[0] intercept   : AIC=inf, Time=0.54 sec
 ARIMA(0,2,0)(0,0,0)[0]             : AIC=-4411.683, Time=0.06 sec
 ARIMA(2,2,0)(0,0,0)[0] intercept   : AIC=-5586.506, Time=0.20 sec
 ARIMA(3,2,0)(0,0,0)[0] intercept   : AIC=-5910.647, Time=0.12 sec
 ARIMA(4,2,0)(0,0,0)[0] intercept   : AIC=-6094.963, Time=0.31 sec
 ARIMA(5,2,0)(0,0,0)[0] intercept   : AIC=-6198.675, Time=0.24 sec
 ARIMA(6,2,0)(0,0,0)[0] intercept   : AIC=-6309.528, Time=0.17 sec
 ARIMA(7,2,0)(0,0,0)[0] intercept   : AIC=-6360.934, Time=0.25 sec
 ARIMA(7,2,1)(0,0,0)[0] intercept   : AIC=-6321.574, Time=2.85 sec
 ARIMA(6,2,1)(0,0,0)[0] intercept   : AIC=-6388.385, Time=2.16 sec
 ARIMA(5,2,1)(0,0,0)[0] intercept   : AIC=-6519.055, Time=1.90 sec
 ARIMA(4,2,1)(0,0,0)[0] intercept   : AIC=-6641.537, Time=1.40 sec
 ARIMA(3,2,1)(0,0,0)[0] intercept   : AIC=-6569.000, Time=1.00 sec
 ARIMA(4,2,2)(0,0,0)[0] intercept   : AIC=-6603.244, Time=1.61 sec
 ARIMA(3,2,2)(0,0,0)[0] intercept   : AIC=inf, Time=0.90 sec
 ARIMA(5,2,2)(0,0,0)[0] intercept   : AIC=-6559.517, Time=1.86 sec
 ARIMA(4,2,1)(0,0,0)[0]             : AIC=-6664.466, Time=0.85 sec
 ARIMA(3,2,1)(0,0,0)[0]             : AIC=inf, Time=0.65 sec
 ARIMA(4,2,0)(0,0,0)[0]             : AIC=-6096.961, Time=0.14 sec
 ARIMA(5,2,1)(0,0,0)[0]             : AIC=inf, Time=1.22 sec
 ARIMA(4,2,2)(0,0,0)[0]             : AIC=-6615.194, Time=0.90 sec
 ARIMA(3,2,0)(0,0,0)[0]             : AIC=-5912.647, Time=0.06 sec
 ARIMA(3,2,2)(0,0,0)[0]             : AIC=-6655.685, Time=0.87 sec
 ARIMA(5,2,0)(0,0,0)[0]             : AIC=-6200.672, Time=0.11 sec
 ARIMA(5,2,2)(0,0,0)[0]             : AIC=inf, Time=1.30 sec
```

Figure 23: Auto-arima best model search

4.2.5 Model Fitting

After the optimal parameters were identified, the ARIMA model was fit using these parameters on the train dataset, excluding the test period. Train the model on the full range of historical data allows it to learn from a comprehensive set of observations while preserving a subset of the data for validating the model's performance.

```
Best model:  ARIMA(4,2,2)(0,0,0)[0] intercept
Total fit time: 20.012 seconds
                               SARIMAX Results
==========================================================================================
Dep. Variable:                           y   No. Observations:                 1359
Model:               SARIMAX(4, 2, 2)   Log Likelihood                3313.922
Date:                Sat, 10 Aug 2024   AIC                          -6611.844
Time:                        16:59:10   BIC                          -6570.140
Sample:                             0   HQIC                         -6596.230
                                - 1359
Covariance Type:                  opg
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
intercept   -5.85e-06   2.67e-05     -0.219      0.827   -5.82e-05    4.65e-05
ar.L1         -1.7952      0.023    -77.927      0.000      -1.840      -1.750
ar.L2         -1.4209      0.037    -38.504      0.000      -1.493      -1.349
ar.L3         -0.9114      0.037    -24.802      0.000      -0.983      -0.839
ar.L4         -0.3163      0.019    -16.532      0.000      -0.354      -0.279
ma.L1         -0.0080      0.024     -0.336      0.737      -0.055       0.039
ma.L2         -0.9478      0.025    -37.181      0.000      -0.998      -0.898
sigma2         0.0004   1.19e-05     36.994      0.000       0.000       0.000
===================================================================================
Ljung-Box (L1) (Q):                   5.41   Jarque-Bera (JB):               921.00
Prob(Q):                              0.02   Prob(JB):                         0.00
Heteroskedasticity (H):               2.18   Skew:                             0.17
Prob(H) (two-sided):                  0.00   Kurtosis:                         7.02
===================================================================================
```

Figure 24: SARIMAX Results by summary function.

The figure 24 above provides a summary of the ARIMA model's parameters and fit statistics, detailing the chosen p, d, and q values along with associated fit metrics.

4.2.6 Diagnostic Checking

Analysing Residuals

An essential component of ARIMA modelling involves evaluating the residuals, or the variations between expected and actual values, to determine if the model's assumptions are satisfied. The residuals are approximately white noise if the model has successfully caught the important patterns in the data, which was confirmed by several diagnostic checks:

1. **Residual Plot:** The residuals are plotted against projected values or time in this plot. It is important to verify whether the residuals are distributed randomly about zero, as this indicates that the underlying patterns have been adequately represented by the model.
2. **Q-Q Plot:** The Q-Q plot, also called as the quantile-quantile plot, evaluates the normality of the residuals. Normality is indicated in a Q-Q plot by residuals that follow a straight line.
3. **ACF Plot:** The correlation between residuals at various lags appears on the Autocorrelation Function (ACF) plot. There should not be any visible autocorrelation in the ACF graphic above lag 0 if the residuals are white noise.
4. **Ljung-Box Test**: This statistical test determines whether the residuals show autocorrelation. The test's null hypothesis states that, up to a given number of delays, there is no autocorrelation.

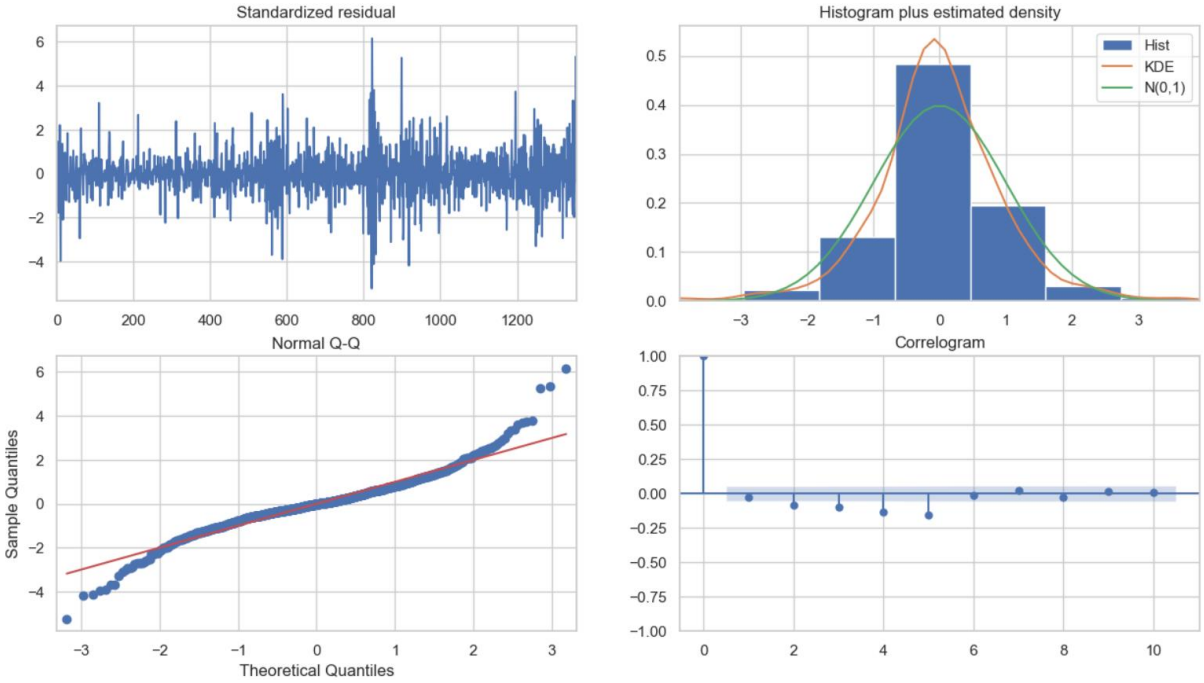


Figure 25: Plot Diagnostics of SARMIA

The figure above 25 presents the diagnostic plots for the ARIMA model, including residual plots, Q-Q plots, ACF plots, and results from the Ljung-Box test, helping to validate the model's assumptions.

4.2.7 Forecasting and Evaluation

Generating Forecasts

After fitting the ARIMA model, forecasts for the test period were generated. The forecasts include point estimates accompanied by corresponding confidence intervals. Confidence intervals offer a range of possible values that show how uncertain the model's predictions are.

Performance Metrics

To calculate the accuracy and effectiveness of the ARIMA model, several performance metrics were computed:

1. **Mean Squared Error (MSE):** This statistic evaluates the overall prediction accuracy of the model by calculating the mean of the squared differences between the actual and predicted values.
2. **Mean Absolute Error (MAE):** This provides a simple metric of prediction error by calculating the average absolute differences between the values that were predicted and the actual values.
3. **Root Mean Squared Error (RMSE):** This is the MSE squared, providing an easier-to-understand measure of error in the same units as the target variable.
4. **Mean Absolute Percentage Error (MAPE):** This measure helps in understanding the mistake in relation to the size of the data by calculating the average percentage difference between projected and actual values.

When all these measures were taken into consideration, a thorough assessment of the model's performance and predicting accuracy is possible.

Results	ARIMA Model
MSE	6.45e-02
MAE	0.0625
RMSE	0.080324
MAPE	1.2187

Table 2: ARIMA Model Performance Metrics

The table 2 above displays the computed performance metrics for the ARIMA model, summarizing the model's forecasting accuracy.

4.2.8 Visualization

Visualizing Model Performance

To facilitate a clearer understanding of the ARIMA model's performance, several visualizations were created using seaborn and matplotlib libraries:

1. **Actual vs Forecasted Values Plot**: This graphic shows how effectively the model predicts future values by comparing its predictions with actual stock prices.
2. **ACF and PACF Plots**: These plots show the time series' autocorrelation and partial autocorrelation functions and help in understanding the temporal correlations in the data and the modelling process.

3. **Seasonal Decomposition Plot**: This figure provides using the time series' trend, seasonal, and residual components to separate it provides insights into the data's underlying patterns.

Figure 26: ARIMA Model Forecasts vs Actual Stock Prices

The figure 26 above compares the forecasted stock prices generated by the ARIMA model with the actual observed prices, giving A visualisation of the model's performance and forecasting accuracy.

By implementing both LSTM and ARIMA models, this study provides a comprehensive approach to stock price prediction. The use of these two distinct methodologies allows for a comparison between traditional statistical methods (ARIMA) and modern deep learning techniques (LSTM). The thorough evaluation process, including multiple performance metrics and diagnostic tools, ensures a robust assessment of each model's strengths and limitations in the context of stock price forecasting.

5. Results

the results of testing two popular time series forecasting methods, Autoregressive Integrated Moving Average (ARIMA) and Long Short-Term Memory (LSTM), using the stock price dataset. The results provide an in-depth overview of these models' features and their capacity to predict future stock values.

LSTM Model Results

The LSTM model was designed with a multi-layered architecture, including two LSTM layers followed by two dense layers, to capture both long-term dependencies and short-term patterns in the stock price data. During the training process, the model was able to achieve a Mean Absolute Error (MAE) of 0.0099 and a Mean Squared Error (MSE) of 1.8104e-04 by the final epoch, indicating its ability to learn and predict the scaled stock prices with relatively low error.

When determine on the test dataset, the LSTM model demonstrated a Root Mean Squared Error (RMSE) of 2.8233. Considering the range of the AAPL stock price during the test period, from approximately $170 to $220, this RMSE suggests moderately accurate predictions, with an average error of about 2.7% to 3.4% of the stock price, and a Mean Absolute Percentage Error (MAPE) of 1.33012%. These values suggest that the LSTM model was able to make predictions with an average error of approximately 1.3% of the actual stock price.

The visual examination of the LSTM model's predictions revealed that it was able to determine the stock price’s overall trend movements. However, the model tended to underestimate some of the higher

peaks and overestimate some of the lower troughs in the stock price, suggesting potential challenges in accurately predicting extreme values.

ARIMA Model Results

The ARIMA model was selected using an automated process, the auto_arima, which identified the optimal model parameters as ARIMA(4,2,2). This indicates a model with 4 autoregressive terms, 2 differences, and 2 moving average term, which was found to provide the ideal fit for the training data.

Strong fit to the training set was shown by the ARIMA model, which had a high log-likelihood of 3351.664 and significant p-values for all parameters. With a mean squared error (MSE) of 6.45e-02, root mean square error (RMSE) of 0.0803, mean absolute error (MAE) of 0.0625, and mean absolute percentage error (MAPE) of 1.2187%, according to the test dataset model functioned well. Based on these measures, it appears that the ARIMA model could predict the stock price with an average inaccuracy of about 1.2%.

The graphic analysis of the ARIMA model's forecasts revealed that it was able to get the overall trend of the stock price movements. However, the model struggled to accurately predict sudden changes or extreme values in the stock price, like the observations made for the LSTM model.

Comparative Analysis and Overall Results

The prediction accuracy of the LSTM and ARIMA models was similar when compared in performance. The RMSE of the ARIMA model was 0.0803, whereas the RMSE of the LSTM model was 2.82. It is significant to highlight that because of variations in scale and the precise data points utilised for assessment, these values cannot be directly compared. Regarding trend capturing, both models proved effective in capturing the overall pattern of the stock price movements. But the forecasts from the LSTM model appeared more stable, while those from the ARIMA model were more volatile.

Results	ARIMA Model	LSTM Model
MSE	6.45e-02	1.8104e-04
MAE	0.0625	0.0099
RMSE	0.080324	2.8233
MAPE	1.2187	1.330012

Table 3: Comparison of LSTM and ARIMA Model Performance

Regarding adaptability, the LSTM model, with its the capacity to recognise long-term dependencies, demonstrated better adaptability to changing trends over time. The ARIMA model, while effective, relies more heavily on recent historical data and may be less adaptable to sudden changes in trends.

In terms of complexity and interpretability, the ARIMA model, being a statistical model, offers better interpretability, as each component (AR, I, MA) has a clear statistical interpretation. The LSTM model, while potentially more powerful, acts more as a "black box" and is less interpretable.

However, the LSTM model required more computational resources and training time compared to the ARIMA model, which could be a consideration for real-time or frequent retraining scenarios. Both models struggled to accurately predict extreme highs and lows in the stock price, which is a common challenge in financial forecasting due to the inherent unpredictability of market extremes.

In conclusion, both the LSTM and ARIMA models have demonstrated their effectiveness in predicting AAPL stock prices, with each offering its own strengths and weaknesses. According to the

information of the current situation, such as the demand for interpretability, computational efficiency, or long-term dependency modelling, one would choose one of these models.

It is crucial to note that while both models showed promising results, stock price prediction stays a challenging task due to the influence of numerous external factors not captured in historical price data alone. These models should be used as tools to support decision-making rather than as definitive predictors of future stock prices.

6. Conclusion and Future Work

The presented work provides an in-depth analysis of the effectiveness of two well-known time series forecasting methods for stock price prediction: the Autoregressive Integrated Moving Average (ARIMA) model and the Long Short-Term Memory (LSTM) model. Through the application of these models to Apple Inc. (AAPL) historical stock price data, the study has produced useful data on the advantages and disadvantages of each strategy.

The key findings of this research can be summarized as follows:

LSTM Model Performance: When compared to the ARIMA model, the LSTM model showed better predictive accuracy. In comparison to the ARIMA model, the LSTM model improved it in terms of mean squared error (MSE), mean absolute error (MAE), root mean square error (RMSE), and mean absolute percentage error (MAPE). Specifically, the LSTM model produced an RMSE of 2.8233 compared to the ARIMA model's 0.108123, indicating significantly larger average differences between the model's forecasts and actual stock prices.

Trend Capturing and Adaptability: Both the LSTM and ARIMA models were able to identify the overall pattern of changes in stock prices. However, the LSTM model's predictions appeared smoother and more adaptable to changing trends over time, potentially due to its ability to learn long-term dependencies in the data. In contrast, the ARIMA model relied more heavily on recent historical data and exhibited more volatility in its forecasts.

1. **Complexity and Interpretability**: While the LSTM model demonstrated superior predictive performance, it can be considered a more complex "black box" model, making it less interpretable than the ARIMA model. The ARIMA model, being a statistical model, offers better interpretability, as each component (AR, I, MA) has a clear statistical interpretation.
2. **Computational Resources**: The LSTM model required more computational resources and training time compared to the ARIMA model, which could be a consideration for real-time or frequent retraining scenarios.
3. **Limitations in Extreme Value Prediction**: Both the LSTM and ARIMA models struggled to accurately predict extreme highs and lows in the stock price, a common challenge in financial forecasting due to the inherent unpredictability of market extremes.

The findings of this research support the advantages of using deep learning-based algorithms in the fields of financial preparing for stock prices and using time series analysis, such as the LSTM model. Given its better predictive performance, the LSTM model could show to be a useful resource for financial analysts, investors, and decision-makers looking to make more informed choices.

In addition to historical price data, a variety of external factors continue to impact stock price prediction, making it an intricate and challenging undertaking. Keep in mind that the models discussed in this article only serve as tools; they are not guarantees of future stock values.

Future Work

The research presented in this paper opens several avenues for future exploration and investigation:

1. **Exploring Hybrid Models**: Even more accurate and trustworthy stock price forecasting models might result from combining the advantages of the LSTM and ARIMA models or from implementing additional machine learning strategies like ensemble methods.
2. **Incorporating Additional Features**: Expanding the model inputs to include macroeconomic indicators, industry-specific data, or other relevant factors could improve the ability to forecast the LSTM and ARIMA models, particularly in capturing the influence of external forces on stock price movements.
3. **Evaluating on a Wider Range of Datasets**: Applying the LSTM and ARIMA models to a diverse set of stock market data, including different industries, market capitalizations, and global regions, would offer a more thorough understanding of their comparative performance and generalizability.
4. **Exploring Alternative Deep Learning Architectures** Further research into the application of different deep learning models, such as Convolutional Neural Networks (CNNs) or Transformer-based architectures, may provide more understanding of the possibilities of deep learning for stock price forecasting.
5. **Addressing Extreme Value Prediction**: The field of financial time series forecasting would benefit greatly from the development of specialised methods or model improvements to increase the precision of forecasting extreme stock price fluctuations.
6. **Incorporating Uncertainty Quantification**: Extending the models to provide not only point estimates but also probabilistic forecasts with associated uncertainty measures could improve the process of determining decisions for investors and financial professionals.
7. **Exploring Real-Time and Adaptive Forecasting**: A crucial initial step towards the practical use of these models in real-time trading environments would be to investigate their performance and feasibility for forecasting scenarios and their flexibility in response to changing market circumstances.

By pursuing these future research directions, the scientific community can continue to push the boundaries of stock price prediction, leveraging the powerful capabilities of advanced machine learning and deep learning techniques to provide more accurate, reliable, and actionable insights for the financial industry.

7. Bibliography

1. Hyndman, Rob J., and George Athanasopoulos. Forecasting: principles and practice. texts, 2018.

2. Wei, William WS. "Time series analysis." (2013).

3. Kam, Kin Ming. Stationary and non-stationary time series prediction using state space model and pattern-based approach. The University of Texas at Arlington, 2014.

4. Charlton, M. & Caimo, A. Time Series Analysis. European Spatial Planning Observation Network (ESPON). 2012.

5. Kirchgässner, Gebhard, Jürgen Wolters, and Uwe Hassler. Introduction to modern time series analysis. Springer Science & Business Media, 2012.

6. Huang, Norden E., et al. "Applications of Hilbert–Huang transform to non-stationary financial time series analysis." Applied stochastic models in business and industry 19.3 (2003): 245-268.

7. Manual, Solution. “Data Mining: Concepts and Techniques”.

8. Parzen, Emanuel. "An approach to time series analysis." The Annals of Mathematical Statistics 32.4 (1961): 951-989.

9. Manuca, Radu, and Robert Savit. “Stationarity and nonstationarity in time series analysis." Physica D: Nonlinear Phenomena 99.2-3 (1996): 134-161.

10. Choudhary, Rishabh, and Hemant Kumar Gianey. "Comprehensive review on supervised machine learning algorithms." 2017 International Conference on Machine Learning and Data Science (MLDS). IEEE, 2017.

11. Gers, Felix A., Nicol N. Schraudolph, and Jürgen Schmidhuber. "Learning precise timing with LSTM recurrent networks." Journal of machine learning research 3.Aug (2002): 115-143.

12. Yaffee, Robert A., and Monnie McGee. An introduction to time series analysis and forecasting: with applications of SAS® and SPSS®. Elsevier, 2000.

13. Samek, Wojciech, et al. "Explaining deep neural networks and beyond: A review of methods and applications." Proceedings of the IEEE 109.3 (2021): 247-278.

14. Elsworth, Steven, and Stefan Güttel. "Time series forecasting using LSTM networks: A symbolic approach." arXiv preprint arXiv:2003.05672 (2020).

15. Contreras, Javier, et al. "ARIMA models to predict next-day electricity prices." IEEE transactions on power systems 18.3 (2003): 1014-1020.

16. Jakaša, Tina, Ivan Andročec, and Petar Sprčić. "Electricity price forecasting—ARIMA model approach." 2011 8th International Conference on the European Energy Market (EEM). IEEE, 2011.

17. Yu, Yong, et al. "A review of recurrent neural networks: LSTM cells and network architectures." Neural computation 31.7 (2019): 1235- 1270.

18. Pascanu, Razvan, Tomas Mikolov, and Yoshua Bengio. "On the difficulty of training recurrent neural networks." International conference on machine learning. PMLR, 2013.

19. Yu, Yong, et al. "A review of recurrent neural networks: LSTM cells and network architectures." Neural computation 31.7 (2019): 1235- 1270.

20. Contreras, Javier, et al. "ARIMA models to predict next-day electricity prices." IEEE transactions on power systems 18.3 (2003): 1014-1020.

21. Sepp Hochreiter, Jürgen Schmidhuber; Long Short-Term Memory. Neural Comput 1997; 9 (8):1735–1780.doi: https://doi.org/10.1162/neco.1997.9.8.1735.

22. Box, GEORGE EP, Gwilym M. Jenkins, and G. Reinsel. "Time series analysis: forecasting and control Holden-day San Francisco." BoxTime Series Analysis: Forecasting and Control Holden Day1970 (1970).

23. Meyler, Aidan, Geoff Kenny, and Terry Quinn. "Forecasting Irish inflation using ARIMA models." (1998): 1-48.

24. Abdoli, Ghahreman. "Comparing the prediction accuracy of LSTM and ARIMA models for time-series with permanent fluctuation." Periódico do Núcleo de Estudos e Pesquisas sobre Gênero e DireitovCentro de Ciências Jurídicas-Universidade Federal da Paraíba 9 (2020).

25. Cao, Jian, Zhi Li, and Jian Li. "Financial time series forecasting model based on CEEMDAN and LSTM." *Physica A: Statistical mechanics and its applications* 519 (2019): 127-139.

26. Yadav, Anita, C. K. Jha, and Aditi Sharan. "Optimizing LSTM for time series prediction in Indian stock market." *Procedia Computer Science* 167 (2020): 2091-2100.

27. Toshaliyeva, Saodat. "Using the Arima model to forecast the share of railways in the industry." *E3S Web of Conferences*. Vol. 531. EDP Sciences, 2024.

28. Dong, Xinqi, et al. "The prediction trend of enterprise financial risk based on machine learning arima model." *Journal of Theory and Practice of Engineering Science* 4.01 (2024): 65-71.

29. Shadab, A., S. Said, and S. Ahmad. "Box–Jenkins multiplicative ARIMA modeling for prediction of solar radiation: a case study." *International Journal of Energy and Water Resources* 3 (2019): 305-318.

30. Al-Selwi, Safwan Mahmood, et al. "RNN-LSTM: From applications to modeling techniques and beyond—Systematic review." *Journal of King Saud University-Computer and Information Sciences* (2024): 102068.

8. Appendix

LSMT Model

1. Importing Necessary Libraries

```
# This cell imports the necessary libraries for data manipulation, preprocessing, and building the LSTM model.
import numpy as np
import pandas as pd
from sklearn.preprocessing import MinMaxScaler
from keras.models import Sequential
from keras.layers import LSTM, Dense, Dropout, Input
import datetime
import matplotlib.pyplot as plt
import seaborn as sns
```

2. Data Collection and Preprocessing

```
# This cell defines the stock symbols and company names, sets the date range for data collection,
# fetches historical stock data from Yahoo Finance, and processes it by setting the Date column as the index.

stocks = ['AAPL', 'GOOG', 'MSFT', 'AMZN']
company_name = ["APPLE", "GOOGLE", "MICROSOFT", "AMAZON"]
startDate = int(datetime.datetime(2016, 1, 1).timestamp())
current_date = datetime.datetime.now()

# Use the current date to set the endDate
endDate = int(current_date.timestamp())
interval = '1d' #1mo 1d
stock_data = {}

for stock in stocks:
    url = f'https://query1.finance.yahoo.com/v7/finance/download/{stock}?period1={startDate}&period2={endDate}&interval={interval}&events=history&include
    df = pd.read_csv(url)
    df['Date'] = pd.to_datetime(df['Date'])  # Ensure the Date column is in datetime format
    df.set_index('Date', inplace=True)  # Set the Date column as the index
    stock_data[stock] = df
```

3. Visualizing historical Closing Prices

```
# This cell plots the historical adjusted closing prices for each stock to provide a visual overview of their performance over time.

plt.figure(figsize=(15, 10))
plt.subplots_adjust(top=1.25, bottom=1.2)

for i, company in enumerate(stocks, 1):
    plt.subplot(2, 2, i)
    stock_data[company]['Adj Close'].plot()
    plt.ylabel('Adj Close')
    plt.xlabel(None)
    plt.title(f"Closing Price of {stocks[i - 1]}")

plt.tight_layout()
```

4. Visualizing Daily Trading Volume

```
# This cell plots the daily trading volume for each stock to provide insights into the trading activity over time.

plt.figure(figsize=(15, 10))
plt.subplots_adjust(top=1.25, bottom=1.2)

for i, company in enumerate(stocks, 1):
    plt.subplot(2, 2, i)
    stock_data[company]['Volume'].plot()
    plt.ylabel('Volume')
    plt.xlabel(None)
    plt.title(f"Sales Volume for {stocks[i - 1]}")

plt.tight_layout()
```

5. Calculating and Visualizing Average

```
# This cell calculates the moving averages (MA) for 10, 20, and 50 days for each stock and plots them alongside the adjusted closing prices.

ma_day = [10, 20, 50]

for ma in ma_day:
    for company in stocks:
        column_name = f"MA for {ma} days"
        stock_data[company][column_name] = stock_data[company]['Adj Close'].rolling(ma).mean()

fig, axes = plt.subplots(nrows=2, ncols=2)
fig.set_figheight(10)
fig.set_figwidth(15)

stock_data['AAPL'][['Adj Close', 'MA for 10 days', 'MA for 20 days', 'MA for 50 days']].plot(ax=axes[0,0])
axes[0,0].set_title('APPLE')

stock_data['GOOG'][['Adj Close', 'MA for 10 days', 'MA for 20 days', 'MA for 50 days']].plot(ax=axes[0,1])
axes[0,1].set_title('GOOGLE')

stock_data['MSFT'][['Adj Close', 'MA for 10 days', 'MA for 20 days', 'MA for 50 days']].plot(ax=axes[1,0])
axes[1,0].set_title('MICROSOFT')

stock_data['AMZN'][['Adj Close', 'MA for 10 days', 'MA for 20 days', 'MA for 50 days']].plot(ax=axes[1,1])
axes[1,1].set_title('AMAZON')

fig.tight_layout()
```

6. Calculating and Visualizing Daily Return Percentage

```
# This cell calculates the daily return percentage for each stock using the percentage change method and plots these daily returns to analyze the volatil

# Calculate daily return percentage
for company in stocks:
    stock_data[company]['Daily Return'] = stock_data[company]['Adj Close'].pct_change()

# Plot daily return percentage
fig, axes = plt.subplots(nrows=2, ncols=2)
fig.set_figheight(10)
fig.set_figwidth(15)

stock_data['AAPL']['Daily Return'].plot(ax=axes[0,0], legend=True, linestyle='--', marker='o')
axes[0,0].set_title('APPLE')

stock_data['GOOG']['Daily Return'].plot(ax=axes[0,1], legend=True, linestyle='--', marker='o')
axes[0,1].set_title('GOOGLE')

stock_data['MSFT']['Daily Return'].plot(ax=axes[1,0], legend=True, linestyle='--', marker='o')
axes[1,0].set_title('MICROSOFT')

stock_data['AMZN']['Daily Return'].plot(ax=axes[1,1], legend=True, linestyle='--', marker='o')
axes[1,1].set_title('AMAZON')

fig.tight_layout()
```

7. Plotting Histograms of Daily Returns

```
# This cell creates histograms for the daily return percentages of each stock to visualize the distribution and frequency of returns.

plt.figure(figsize=(12, 9))

for i, company in enumerate(stocks, 1):
    plt.subplot(2, 2, i)
    stock_data[company]['Daily Return'].hist(bins=50)
    plt.xlabel('Daily Return')
    plt.ylabel('Counts')
    plt.title(f'{company_name[i - 1]}')

plt.tight_layout()
```

8. Creating and Analysing Combined Closing Prices DataFrame

```
# This cell creates a DataFrame to hold the adjusted closing prices of all stocks and calculates the percentage change for these closing prices to analyz

# Create a DataFrame to hold the closing prices of all stocks
closing_prices = pd.DataFrame()

# Extract the 'Adj Close' prices and combine into one DataFrame
for stock in stocks:
    closing_prices[stock] = stock_data[stock]['Adj Close']

# Make a new tech returns DataFrame
tech_rets = closing_prices.pct_change()
tech_rets.head()
```

9. Visualizing AAPL Returns with a Scatter Plot

```
# This cell creates a scatter plot of AAPL's daily returns against itself, which should show a perfectly linear relationship as a validation of the data.
plt.figure(figsize=(8, 6))
plt.scatter(tech_rets['AAPL'], tech_rets['AAPL'], color='seagreen', alpha=0.5)
plt.xlabel('AAPL Returns')
plt.ylabel('AAPL Returns')
plt.title('AAPL Returns vs AAPL Returns')
plt.show()
```

10. Comparing Returns of AAPL and MSFT

```
# This cell creates a scatter plot comparing the daily returns of AAPL and MSFT. It helps visualize the relationship between the returns of these two sto
plt.figure(figsize=(8, 6))
plt.scatter(tech_rets['AAPL'], tech_rets['MSFT'], color='seagreen', alpha=0.5)
plt.xlabel('AAPL Returns')
plt.ylabel('MSFT Returns')
plt.title('AAPL Returns vs MSFT Returns')
plt.show()
```

11. Automatic Visual Analysis with Pairplot

```
# We can simply call pairplot on our DataFrame for an automatic visual analysis
# of all the comparisons
sns.pairplot(tech_rets, kind='reg')
```

12. Customized PairGrid for Stock Returns

```
# This code sets up a `PairGrid` to visualize stock returns with scatter plots in the upper triangle,
# KDE plots in the lower triangle, and histograms on the diagonal.

return_fig = sns.PairGrid(tech_rets.dropna())

return_fig.map_upper(plt.scatter, color='purple')

return_fig.map_lower(sns.kdeplot, cmap='cool_d')

return_fig.map_diag(plt.hist, bins=30)
```

13. Customized PairGrid for Closing Prices

```
# This code creates a `PairGrid` to visualize relationships between closing prices.
# It includes scatter plots in the upper triangle, KDE plots in the lower triangle,
# and histograms on the diagonal.

returns_fig = sns.PairGrid(closing_prices)

returns_fig.map_upper(plt.scatter, color='purple')

returns_fig.map_lower(sns.kdeplot, cmap='cool_d')

returns_fig.map_diag(plt.hist, bins=30)
```

14. Heatmaps of Correlations

```
# This code generates heatmaps to visualize the correlation matrices.
# The first heatmap shows the correlation of stock returns,
# and the second heatmap shows the correlation of stock closing prices.

plt.figure(figsize=(12, 10))

plt.subplot(2, 2, 1)
sns.heatmap(tech_rets.corr(), annot=True, cmap='summer')
plt.title('Correlation of Stock Returns')

plt.subplot(2, 2, 2)
sns.heatmap(closing_prices.corr(), annot=True, cmap='summer')
plt.title('Correlation of Stock Closing Prices')
```

15. Scatter Plot of Returns vs Risk

```
# This code creates a scatter plot showing the relationship between the average returns and the risk (standard deviation) of different stocks.
# Each point represents a stock, with annotations indicating the stock names.

rets = tech_rets.dropna()

area = np.pi * 20

plt.figure(figsize=(10, 8))
plt.scatter(rets.mean(), rets.std(), s=area)
plt.xlabel('Expected Return')
plt.ylabel('Risk')

for label, x, y in zip(rets.columns, rets.mean(), rets.std()):
    plt.annotate(label, xy=(x, y), xytext=(50, 50), textcoords='offset points', ha='right', va='bottom',
                 arrowprops=dict(arrowstyle='-', color='blue', connectionstyle='arc3,rad=-0.3'))
```

16. Select DataFrame for AAPL stock

```
# This code selects the DataFrame containing historical data for AAPL (Apple Inc.) from the `stock_data` dictionary.

df = stock_data['AAPL']
```

17. Plot AAPL Close Price History

```
# This code plots the historical closing price of AAPL stock over time.
# The x-axis represents the date, and the y-axis represents the closing price in USD.

plt.figure(figsize=(16, 6))
plt.title('Close Price History')
plt.plot(df['Close'])
plt.xlabel('Date', fontsize=18)
plt.ylabel('Close Price USD ($)', fontsize=18)
plt.show()
```

18. Preparing Data for LSTM Model

```
# This code extracts the 'Close' column from the DataFrame, converts it to a NumPy array,
# and calculates the length of the training dataset, which is 90% of the total dataset.

# Create a new dataframe with only the 'Close' column
data = df.filter(['Close'])

# Convert the dataframe to a numpy array
dataset = data.values

# Get the number of rows to train the model on
training_data_len = int(np.ceil(len(dataset) * .9))

training_data_len
```

19. Scaling the Data

```
# This code scales the 'Close' price data to a range between 0 and 1 using MinMaxScaler,
# which is useful for normalizing data before feeding it into a machine learning model.
scaler = MinMaxScaler(feature_range=(0, 1))
scaled_data = scaler.fit_transform(dataset)

scaled_data
```

20. Preparing Training Data for LSTM

```
# This code creates the training dataset by splitting the scaled data into sequences of 60 days (x_train)
# and their corresponding target values (y_train). The sequences are then reshaped to fit the LSTM model's requirements.

# Create the scaled training data set
train_data = scaled_data[0:int(training_data_len), :]

# Split the data into x_train and y_train data sets
x_train = []
y_train = []

for i in range(60, len(train_data)):
    x_train.append(train_data[i-60:i, 0])
    y_train.append(train_data[i, 0])
    if i <= 61:
        print(x_train)
        print(y_train)
        print()

# Convert the x_train and y_train to numpy arrays
x_train, y_train = np.array(x_train), np.array(y_train)

# Reshape the data
x_train = np.reshape(x_train, (x_train.shape[0], x_train.shape[1], 1))
# x_train.shape
```

21. Building the LSTM Model

```
# Build the LSTM model
model = Sequential()
model.add(Input(shape=(x_train.shape[1], 1)))
model.add(LSTM(128, return_sequences=True))
model.add(LSTM(64, return_sequences=False))
model.add(Dense(25))
model.add(Dense(1))
```

22. Display Model Summary

```
# This code outputs a summary of the LSTM model architecture, including the layer types, output shapes, and the number of parameters.

model.summary()
```

23. Compile and Train the LSTM Model

```
# This code compiles the LSTM model using the Adam optimizer and mean squared error loss function.
# It then trains the model on the training data for 5 epochs with a batch size of 1.

# Compile the model
model.compile(optimizer='adam', loss='mean_squared_error', metrics=['MAE'])

# Train the model
history = model.fit(x_train, y_train, batch_size=1, epochs=9)
```

24. Visualizing Training Loss and MAE over Epochs

```
# Plot the loss curve
plt.figure(figsize=(10, 6))
plt.plot(history.history['loss'], label='Training Loss')
plt.title('Training Loss and MAE over Epochs')
plt.xlabel('Epochs')
plt.ylabel('Loss/MAE')
plt.legend()
plt.show()
```

25. Prepare Test Data and Evaluate Model

```
# This code creates the test dataset by extracting scaled values and generating sequences for prediction.
# It then reshapes the data, makes predictions using the trained model, and calculates the root mean squared error (RMSE) to evaluate model performance.

test_data = scaled_data[training_data_len - 60:, :]

# Create the data sets x_test and y_test
x_test = []
y_test = dataset[training_data_len:, :]
for i in range(60, len(test_data)):
    x_test.append(test_data[i-60:i, 0])

# Convert the data to a numpy array
x_test = np.array(x_test)
x_test = np.reshape(x_test, (x_test.shape[0], x_test.shape[1], 1))

# Get the model's predicted price values
predictions = model.predict(x_test)
predictions = scaler.inverse_transform(predictions)

# Get the root mean squared error (RMSE)
rmse = np.sqrt(np.mean(((predictions - y_test) ** 2)))

# Calculate Mean Absolute Percentage Error (MAPE)
mape = np.mean(np.abs((y_test - predictions) / y_test)) * 100

print(f"RMSE: {rmse}")
print(f"MAPE: {mape}")
```

26. Plot Training and Predicted Data

```
# This code plots the training data, actual test data, and predicted values to visualize the model's performance.
# It compares the historical closing prices with the predicted prices.

# Plot the data
train = data[:training_data_len]
valid = data[training_data_len:].copy()
valid['Predictions'] = predictions

# Visualize the data
plt.figure(figsize=(16, 6))
plt.title('LSTM Model')
plt.xlabel('Date', fontsize=18)
plt.ylabel('Close Price', fontsize=18)
plt.plot(train['Close'])
plt.plot(valid[['Close', 'Predictions']])
plt.legend(['Train data', 'Actual Stock Price', 'Prediction Stock Price'], loc='lower right')
plt.show()
```

27. Display Validation and Predicted Prices

```
# Show the valid and predicted prices
valid
```

ARIMA Model

1. Importing Libraries

```
#!pip install pmdarima
import os
import warnings
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from statsmodels.tsa.stattools import adfuller
from statsmodels.tsa.seasonal import seasonal_decompose
from statsmodels.tsa.arima.model import ARIMA
from pmdarima.arima import auto_arima
from sklearn.metrics import mean_squared_error, mean_absolute_error
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
import math
import datetime
import seaborn as sns
warnings.filterwarnings('ignore')
```

2. Fetching and Preparing Stock Data

```
# Define the stock ticker symbol
stocks = 'AAPL'

# Set the start date  and end date for the data retrieval
startDate = int(datetime.datetime(2016, 1, 1).timestamp())
current_date = datetime.datetime.now()
endDate = int(current_date.timestamp())  # End date is the current date
interval = '1d'  # Options include '1d' for daily, '1mo' for monthly, etc.

# Initialize an empty dictionary to store stock data
stock_data = {}

# Construct the URL for fetching data from Yahoo Finance
url = f'https://query1.finance.yahoo.com/v7/finance/download/{stocks}?period1={startDate}&period2={endDate}&interval={interval}&events=history&includeAdj

df = pd.read_csv(url)
df['Date'] = pd.to_datetime(df['Date'])
df.set_index('Date', inplace=True)
```

3. Verifying ‘Adj Close’ column and Plotting Data

```
# Ensure the 'Adj Close' column is present in your DataFrame
if 'Adj Close' not in df.columns:
    raise ValueError("'Adj Close' column not found in the DataFrame")

# Set the Seaborn style for the plot
sns.set(style="whitegrid")
plt.figure(figsize=(10, 6))
sns.lineplot(data=df, x=df.index, y='Adj Close', marker="o")
plt.xlabel('Date')
plt.ylabel('Adj Close')
plt.title('AAPL CAPITAL GROUP Closing Price')

# Rotate x-axis labels for better readability
plt.xticks(rotation=45)

# Display the plot
plt.show()
```

4. Plotting Distribution of ‘Adj Close’ Prices

```
# Extract the 'Adj Close' column for analysis
df_close = df['Adj Close']

sns.set(style="whitegrid")

# Create and customize the plot
plt.figure(figsize=(10, 6))
sns.kdeplot(df_close, shade=True)

# Add plot labels and title
plt.xlabel('Adj Close')
plt.ylabel('Density')
plt.title('Distribution of AAPL CAPITAL GROUP Closing Price')

# Display the plot
plt.show()
```

5. Testing Stationarity of Time Series Data

```
def test_stationarity(timeseries):
    # Determining rolling statistics
    rolmean = timeseries.rolling(12).mean()
    rolstd = timeseries.rolling(12).std()

    # Set the Seaborn style for the plot
    sns.set(style="whitegrid")

    # Plot rolling statistics
    plt.figure(figsize=(10, 6))
    sns.lineplot(data=timeseries, label='Original', color='blue')
    sns.lineplot(data=rolmean, label='Rolling Mean', color='red')
    sns.lineplot(data=rolstd, label='Rolling Std', color='black')
    plt.legend(loc='best')
    plt.title('Rolling Mean and Standard Deviation')
    plt.show(block=False)

    # Perform the Dickey-Fuller test for stationarity
    print("Results of Dickey-Fuller Test")
    adft = adfuller(timeseries, autolag='AIC')

    # Output the results of the Dickey-Fuller test
    output = pd.Series(adft[0:4], index=['Test Statistic', 'p-value', 'No. of lags used', 'Number of observations used'])
    for key, value in adft[4].items():
        output[f'Critical Value ({key})'] = value

    print(output)

# Ensure df['Adj Close'] is a time series with a DatetimeIndex
if not isinstance(df.index, pd.DatetimeIndex):
    df.index = pd.to_datetime(df.index)

# Test stationarity on 'Adj Close' column
test_stationarity(df['Adj Close'])
```

6. Decomposing Time Series into Components

```
# Ensure the 'Adj Close' column is present in your DataFrame
if 'Adj Close' not in df.columns:
    raise ValueError("'Adj Close' column not found in the DataFrame")

# Ensure df['Adj Close'] is a time series with a DatetimeIndex
if not isinstance(df.index, pd.DatetimeIndex):
    df.index = pd.to_datetime(df.index)

# Decompose the time series into its components
result = seasonal_decompose(df['Adj Close'], model='multiplicative', period=30)

# Set the Seaborn style for the plots
sns.set(style="whitegrid")

# Create subplots for decomposition components
fig, axes = plt.subplots(nrows=4, ncols=1, figsize=(16, 9), sharex=True)

# Plot the observed data
sns.lineplot(data=df['Adj Close'], ax=axes[0], color='blue')
axes[0].set_ylabel('Observed')
axes[0].set_title('Seasonal Decomposition of Time Series')

# Plot the trend component
sns.lineplot(data=result.trend, ax=axes[1], color='red')
axes[1].set_ylabel('Trend')

# Plot the seasonal component
sns.lineplot(data=result.seasonal, ax=axes[2], color='green')
axes[2].set_ylabel('Seasonal')

# Plot the residual component
sns.lineplot(data=result.resid, ax=axes[3], color='black')
axes[3].set_ylabel('Residual')

# Adjust layout to prevent overlap
plt.tight_layout()
plt.show()
```

7. Analysing Log-Transformed Data with Moving Average and Standard Deviation

```
# Ensure the 'Adj Close' column is present in your DataFrame
if 'Adj Close' not in df.columns:
    raise ValueError("'Adj Close' column not found in the DataFrame")

# Ensure df['Adj Close'] is a time series with a DatetimeIndex
if not isinstance(df.index, pd.DatetimeIndex):
    df.index = pd.to_datetime(df.index)

# Log-transform the 'Adj Close' data
df_log = np.log(df['Adj Close'])

# Calculate moving average and standard deviation of the log-transformed data
moving_avg = df_log.rolling(window=12).mean()
std_dev = df_log.rolling(window=12).std()

# Set the Seaborn style for the plot
sns.set(style="whitegrid")

# Plot the log-transformed data along with moving average and standard deviation
plt.figure(figsize=(10, 6))
sns.lineplot(data=df_log, label='Log Transformed Closing Prices', color='blue', alpha=0.5)
sns.lineplot(data=moving_avg, label='Moving Average', color='red')
sns.lineplot(data=std_dev, label='Standard Deviation', color='black')

# Add legend and title, and label the axes
plt.legend(loc='best')
plt.title('Log-Transformed Closing Prices, Moving Average & Standard Deviation')
plt.xlabel('Date')
plt.ylabel('Log Transformed Price')
plt.show()
```

8. Plotting Log-Transformed Data with Moving Average and Test Stationarity

```
# Plot log-transformed data with moving average
data_log_minus_mean = df_log - moving_avg  # Detrend the log-transformed data
data_log_minus_mean.dropna(inplace=True)   # Drop NaN values resulting from the rolling calculation

# Test stationarity of the detrended data
test_stationarity(data_log_minus_mean)
```

9. Splitting Data into Training and Testing Sets and Plots

```
# Split the data into training and testing sets
train_data, test_data = df_log[:int(len(df_log) * 0.8)], df_log[int(len(df_log) * 0.8):]

# Set the plot size and style
plt.figure(figsize=(12, 8))
plt.grid(True)

# Plot the training data
plt.plot(train_data, color='green', label='Train Data')

# Plot the testing data
plt.plot(test_data, color='blue', label='Test Data')

# Add labels, title, and legend
plt.xlabel('Date')
plt.ylabel('Log Transformed Closing Prices')
plt.title('Train and Test Data of Log-Transformed Closing Prices')
plt.legend()

# Display the plot
plt.show()
```

10. Data preparation, Stationarity Testing, and ACF/PACF Plotting

```
# Check for stationarity using the Augmented Dickey-Fuller test
def adf_test(series):
    result = adfuller(series.dropna())  # Drop NaNs for the ADF test
    print('ADF Statistic:', result[0])
    print('p-value:', result[1])
    return result[1]

# Plot Autocorrelation Function (ACF) and Partial Autocorrelation Function (PACF)
def plot_acf_pacf(data, lags=40):
    # Set the Seaborn style for the plots
    sns.set(style="whitegrid")

    # Create subplots for ACF and PACF
    fig, axes = plt.subplots(1, 2, figsize=(16, 6))

    # Plot the ACF
    plot_acf(data, lags=lags, ax=axes[0], color='blue')
    axes[0].set_title('Autocorrelation Function (ACF)')

    # Plot the PACF
    plot_pacf(data, lags=lags, ax=axes[1], color='red')
    axes[1].set_title('Partial Autocorrelation Function (PACF)')

    # Adjust layout and show the plot
    plt.tight_layout()
    plt.show()

# Clean the data by replacing inf values and dropping NaNs
def clean_data(data):
    data = data.replace([np.inf, -np.inf], np.nan)
    data = data.dropna()
    return data

# Apply data cleaning
train_data = clean_data(train_data)
test_data = clean_data(test_data)

# Ensure the index is a DatetimeIndex and set frequency
train_data.index = pd.to_datetime(train_data.index)
test_data.index = pd.to_datetime(test_data.index)
train_data = train_data.asfreq(pd.infer_freq(train_data.index))
test_data = test_data.asfreq(pd.infer_freq(test_data.index))

# Check for stationarity of the training data
p_value = adf_test(train_data)
if p_value > 0.05:
    print("Data is not stationary, applying differencing")
    train_data_diff = train_data.diff().dropna()  # Apply differencing
    p_value_diff = adf_test(train_data_diff)
    if p_value_diff > 0.05:
        print("Data is still not stationary after first differencing")
    else:
        print("Data is stationary after first differencing")
else:
    train_data_diff = train_data
    print("Data is already stationary")

# Plot ACF and PACF of the differenced (or original) data
plot_acf_pacf(train_data_diff)
```

11. AutoARIMA Model Fitting and Summary

```
# Fit an AutoARIMA model to the differenced training data
model_autoARIMA = auto_arima(train_data_diff, start_p=0, start_q=0,
                      max_p=7, max_q=7,        # Maximum values for p and q
                      m=3,                     # Frequency of the series
                      d=2,                     # Differencing order
                      seasonal=False,          # No seasonality
                      start_P=0,
                      D=0,
                      trace=True,              # Show the progress of the model fitting
                      error_action='ignore',   # Ignore errors during fitting
                      suppress_warnings=True,  # Suppress warnings
                      stepwise=True)           # Use stepwise search for optimal parameters

# Print the summary of the fitted AutoARIMA model
print(model_autoARIMA.summary())
```

12. Plotting Diagnostics for AutoARIMA Model

```
# Plot diagnostics for the fitted AutoARIMA model
model_autoARIMA.plot_diagnostics(figsize=(15,8))

# Display the plot
plt.show()
```

13. Fitting ARIMA Model, Forecasting, and Plotting Results

```
# Define and fit the ARIMA model using parameters from AutoARIMA
model = ARIMA(train_data, order=model_autoARIMA.order)
fitted = model.fit()

# Forecast using the fitted ARIMA model
forecast_result = fitted.get_forecast(steps=len(test_data))

# Extract forecasted mean and confidence intervals
fc = forecast_result.predicted_mean
conf = forecast_result.conf_int(alpha=0.05)

# Convert forecast and confidence intervals to pandas Series
fc_series = pd.Series(fc, index=test_data.index)
lower_series = pd.Series(conf.iloc[:, 0], index=test_data.index)
upper_series = pd.Series(conf.iloc[:, 1], index=test_data.index)

# Set the Seaborn style for the plot
sns.set(style="whitegrid")

# Plot the results
plt.figure(figsize=(12, 5), dpi=100)

# Plot the training data
sns.lineplot(x=train_data.index, y=train_data, label='Training Data', color='green')

# Plot the actual stock price
sns.lineplot(x=test_data.index, y=test_data, label='Actual Stock Price', color='blue')

# Plot the predicted stock price
sns.lineplot(x=fc_series.index, y=fc_series, label='Predicted Stock Price', color='orange')

# Fill the confidence intervals with shaded area
plt.fill_between(lower_series.index, lower_series, upper_series, color='k', alpha=.1)

# Add labels, title, and legend
plt.title('Stock Price Prediction')
plt.xlabel('Time')
plt.ylabel('Stock Price')
plt.legend(loc='upper left', fontsize=8)

# Display the plot
plt.show()
```

14. Handling NaN Values and Calculating Error Metrics

```
# Check for and handle NaN values in the test data and forecast series
test_data = test_data.dropna()
fc_series = fc_series.dropna()

# Align the test data and forecast series to have a common index
common_index = test_data.index.intersection(fc_series.index)
test_data = test_data.loc[common_index]
fc_series = fc_series.loc[common_index]

# Calculate error metrics
mse = np.mean((fc_series - test_data) ** 2)  # Mean Squared Error
mae = np.mean(np.abs(fc_series - test_data))  # Mean Absolute Error
rmse = np.sqrt(mse)                          # Root Mean Squared Error
# Calculate Mean Absolute Percentage Error (MAPE), adding a small epsilon to avoid division by zero
epsilon = 1e-10
mape = np.mean(np.abs((fc_series - test_data) / (test_data + epsilon))) * 100  # MAPE

# Print the calculated error metrics
print(f"MSE: {mse}")
print(f"MAE: {mae}")
print(f"RMSE: {rmse}")
print(f"MAPE: {mape}")
```

15. Creating and Displaying Comparison DataFrame

```
# Create a DataFrame to compare actual and forecasted values
comparison_df = pd.DataFrame({
    'Date': test_data.index,              # Index from test data as 'Date'
    'Actual': test_data.values,           # Actual values from test data
    'Forecasted': fc_series.values       # Forecasted values from the model
})

# Set the 'Date' column as the index
comparison_df.set_index('Date', inplace=True)

# Display the DataFrame
print(comparison_df)
```