CS EE World
https://cseeworld.wixsite.com/home
May 2024
30/34
A

Submitter Info:
megatimate [at] gmail [dot] com
University of Toronto CS 2024-2028

COMPARISON OF PERFORMANCE OF ARIMA AND LSTM MODELS FOR STOCK PRICE PREDICTION

Which model, ARIMA or LSTM, demonstrates superior accuracy in predicting stock prices based on empirical evidence?

Computer Science

Word Count: 3,867

Table of Contents

Comparison of Performance of ARIMA and LSTM Models for Stock Price Prediction

1 Introduction

The prospect of making significant returns makes stock markets attractive to traders, investors and professionals alike. Investing in the stock market is risky owing to many factors ranging from macroeconomic to microeconomic factors, government policies, individual company's financial performance, conflicts, and natural disasters.

Therefore, predicting stock prices is of significant interest to market participants as it can assist in making predictable returns, deciding investment strategies, asset allocation and portfolio management.

Over the years, experts have developed numerous mathematical models to identify underlying patterns from market data and forecast stock prices. Auto-Regressive Integrated Moving Average (ARIMA), a popular statistical model for forecasting a time series that can be made *stationary*.[1]

[1] Fuqua School of Business. Introduction to ARIMA Models. https://people.duke.edu/~rnau/411arim.htm. Accessed 10 Dec 2023.

The recent advances in artificial intelligence brought to prominence Recurrent Neural Networks (RNNs) such as Long Short-Term Memory (LSTM), which employ deep learning to identify complex underlying data patterns and make predictions.[2]

However, despite their popularity, limited research is available which establishes the superiority of ARIMA or LSTM in making accurate predictions of stock prices. Findings of Kobiela et al. suggest that ARIMA is more accurate than LSTM, while findings of others, like Ma and Siami Namini et al., suggest the opposite.[3] [4]

This extended essay aims to compare the accuracy of both models in predicting stock prices to determine which model offers superior performance based on empirical evidence by answering the research question**:**

"Which model, ARIMA or LSTM, demonstrates superior accuracy in predicting stock prices based on empirical evidence?"

2 Methodology

The study involved the following steps:

[2] Kobiela, Dariusz, et al. "ARIMA Vs LSTM on NASDAQ Stock Exchange Data." Procedia Computer Science, vol. 207, Jan. 2022, pp. 3836–45. https://doi.org/10.1016/j.procs.2022.09.445. Accessed 10 Dec 2023.

[3] Kobiela, Dariusz, et al. "ARIMA Vs LSTM on NASDAQ Stock Exchange Data." Procedia Computer Science, vol. 207, Jan. 2022, pp. 3836–45. https://doi.org/10.1016/j.procs.2022.09.445. Accessed 10 Dec 2023.

[4] S. Siami-Namini, N. Tavakoli and A. Siami Namin, "A Comparison of ARIMA and LSTM in Forecasting Time Series," 2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA), Orlando, FL, USA, 2018, pp. 1394-1401, doi: 10.1109/ICMLA.2018.00227. PDF available at https://sci-hub.se/10.1109/ICMLA.2018.00227. Accessed 10 Dec 2023.

- Obtaining historical data for selected stocks and preprocessing it for use with ARIMA/LSTM models.
- Splitting the pre-processed data into training/testing sets.
- Estimating optimal ARIMA and LSTM models and fitting them to the training data.
- Making predictions for the test set.
- Evaluating the performance of both models using statistical metrics such as Root Mean Square Error (RMSE).
- Analyzing both models using the performance metrics to draw conclusions.

A more thorough discussion on Experimental methodology followed is given in Section 4.

3 Theoretical Background

A theoretical background of LSTM and ARIMA relevant to answering the research question is discussed here.

3.1 Time Series

A **time series** is an ordered sequence of data points indexed by time, such as stock's daily closing prices, product's monthly sales, and annual rainfall in a region. [5]

[5] Hayes, Adam. "What Is a Time Series and How Is It Used to Analyze Data?" Investopedia, 13 June 2022, www.investopedia.com/terms/t/timeseries.asp.

3.2 Time Series Analysis and Forecasting

Time Series Analysis involves analyses to gain meaningful insights from time series data.[6]

Time Series Forecasting is the process of predicting future values of a time series based on historical data.[7]

3.3 Stationarity

Time series data is often *non-stationary,* meaning that statistical properties like the mean and variance change over time due to inherent *trends*, *seasonality*, *cyclical fluctuations*, and *random noise*. Modelling and predicting *non-stationary time series data* is challenging needing specialized techniques.[8]

3.4 Auto Regressive Integrated Moving Average (ARIMA) Model

ARIMA is a statistical model for forecasting time series data which can be made *stationary* by employing techniques such as *differencing* or *nonlinear transformations*. A *stationary* time series has no *trend*, constant *amplitude,* and consistent *short-term random time patterns*.[9]

---

[6] Gupta, Sakshi. “What Is Time Series Forecasting? Overview, Models &Amp; Methods.” Springboard Blog, 28 Sept. 2023, https://www.springboard.com/blog/data-science/time-series-forecasting/. Accessed 10 Dec 2023.

[7] Gupta, Sakshi. “What Is Time Series Forecasting? Overview, Models &Amp; Methods.” Springboard Blog, 28 Sept. 2023, https://www.springboard.com/blog/data-science/time-series-forecasting/. Accessed 10 Dec 2023.

[8] Gupta, Sakshi. “What Is Time Series Forecasting? Overview, Models &Amp; Methods.” Springboard Blog, 28 Sept. 2023, https://www.springboard.com/blog/data-science/time-series-forecasting/. Accessed 10 Dec 2023.

[9] Fuqua School of Business. Introduction to ARIMA Models. https://people.duke.edu/~rnau/411arim.htm. Accessed 10 Dec 2023.

3.4.1 ARIMA Forecasting Equation

A forecasted value of a stationary time series can be expressed as a weighted sum of previous observations (referred as the *lagged observations* or the *Autoregressive terms*) and/or a weighted sum of previous forecast errors (referred as the *lagged errors* or the *Moving Average terms*) and a constant.[10]

The *ARIMA forecasting* equation can be expressed as:[11]

$$\hat{y}_t = \mu + \phi_1 y_{t-1} + \cdots + \phi_p y_{t-p} - \theta_1 e_{t-1} - \cdots - \theta_q e_{t-q}$$

Where,

- $\hat{y}_t$ is the predicted value at time t
- μ is the constant term representing the mean of the series.
- $y_{t-1}, \ldots, y_{t-p}$ are the past values of the series at times $t-1, \ldots, t-p$, also called lags of the series or the *AR* terms.
- $e_{t-1}, \ldots, e_{t-q}$ are the past forecast errors at times $t-1, \ldots, t-q$, also called the *MA* terms.
- $\phi_1, \ldots, \phi_p$ are the parameters of the *AR* terms.
- $\theta_1, \ldots, \theta_q$ are the parameters of the *MA* terms.
- p is the number of *AR* terms, also called the *AR* order.

[10] Fuqua School of Business. Introduction to ARIMA Models. https://people.duke.edu/~rnau/411arim.htm. Accessed 10 Dec 2023.

[11] Fuqua School of Business. Introduction to ARIMA Models. https://people.duke.edu/~rnau/411arim.htm. Accessed 10 Dec 2023.

- *q* is number of *MA* terms, also called the *MA* order.

3.4.2 ARIMA Order

ARIMA, denoted as ARIMA*(p,d,q*), is thus a combination of the Autoregressive model (*AR* terms) and the Moving Average model (*MA* terms). The time series that needs to be differenced to be made stationary is the Integrated version of the stationary time series. The differencing is done by subtracting the previous observation from the current observation. The parameter *d* refers to the number of times the integrated version of the equation needs to be differenced to make the time series stationary.[12]

The parameters *p, d and q*, collectively referred to as the *order* of the ARIMA model, must be tuned prior for optimal results.

3.4.3 Forecasting with ARIMA Models

Forecasting with ARIMA using the Box-Jenkins Methodology[13] involves the following steps:

3.4.3.1 Identification.

Identification step entails estimating the order (values of *p, d,* and *q*). This is achieved by:

Differencing the data iteratively until stationarity is achieved, which can be confirmed by statistical tests such as the Augmented Dickey-Fuller (ADF) test and the Kwiatkowski-Phillips-

[12] Fuqua School of Business. Introduction to ARIMA Models. https://people.duke.edu/~rnau/411arim.htm. Accessed 10 Dec 2023.

[13] "Box-Jenkins Methodology." Columbia University Mailman School of Public Health, 3 Oct. 2022, www.publichealth.columbia.edu/research/population-health-methods/box-jenkins-methodology.

Schmidt-Shin (KPSS) test. The number of times differencing is applied to make the data stationary gives an estimate of the order of differencing, the *d* parameter in *ARIMA(p,d,q)*.[14]

Visualizing data through *decomposition*, *autocorrelation* (ACF), and *partial autocorrelation* (PACF) plots. The order of autoregression (*p)* and order of moving average (*q)* can be determined by observing the lags in these plots.[15]

3.4.3.2 Estimation

The estimation step involves configuring and optimizing the ARIMA model with the estimated values of *p, d*, and *q* from the previous step and the training data. The performance of the model is evaluated using metrics such as the Akaike Information Criterion (AIC) and Bayesian Information Criterion (BIC), and the model which minimizes both AIC and BIC values is chosen for forecasting.[16]

3.4.3.3 Validation.

The performance of the chosen ARIMA model is then evaluated on testing data. It involves:

- Forecasting.

[14] Stationarity and Detrending (ADF/KPSS) - Statsmodels 0.15.0 (+200). www.statsmodels.org/dev/examples/notebooks/generated/stationarity_detrending_adf_kpss.html.

[15] Iamleonie. "Time Series: Interpreting ACF and PACF." Kaggle, 15 Mar. 2022, http://www.kaggle.com/code/iamleonie/time-series-interpreting-acf-and-pacf. Accessed 12 Dec 2023.

[16] Brownlee, Jason. "Probabilistic Model Selection With AIC, BIC, and MDL." MachineLearningMastery.com, 27 Aug. 2020, https://machinelearningmastery.com/probabilistic-model-selection-measures .20 Dec 2023.

- Evaluating the model's accuracy using metrics such as Root Mean Squared Error (RMSE), Mean Absolute Error (MAE), and Mean Absolute Percentage Error (MAPE).[17]

3.4.4 Employing Search Methods

Alternately, the order of the ARIMA model can also be estimated employing search algorithms like *grid search* or *random search*, seeking the optimal combination of parameters that minimizes AIC and BIC values or offers better performance on training data using metrics such as RMSE, MAE, and MAPE.

3.4.5 Software Libraries

The *statsmodels* library in Python provides tools for plotting ACF, PACF, and decomposition plots, performing statistical tests (ADF, KPSS, AIC, and BIC), fitting the ARIMA model, and performing forecasting and validation steps.[18]

The *scikit-learn* library offers methods for Grid Search, and Random Search etc.[19]

However, libraries like *pmdarima* provide specialized functions for finding the optimal ARIMA model, eliminating the need for pre-processing and custom implementation, saving time, and reducing the scope for errors.[20]

---

[17] Sumi. "Understand ARIMA and Tune P, D, Q." Kaggle, 20 Aug. 2018, www.kaggle.com/code/sumi25/understand-arima-and-tune-p-d-q. Accessed 22 Dec 2023.

[18] Time Series Analysis Tsa - Statsmodels 0.14.1. www.statsmodels.org/stable/tsa.html#descriptive-statistics-and-tests. Accessed 27 Dec 2023.

[19] Rendyk. "Tuning the Hyperparameters and Layers of Neural Network Deep Learning." Analytics Vidhya, 12 Jan. 2024, www.analyticsvidhya.com/blog/2021/05/tuning-the-hyperparameters-and-layers-of-neural-network-deep-learning. Accessed 21 Dec 2023.

[20] pmdarima.arima.auto __ _arima — Pmdarima 2.0.4 Documentation. https://alkaline-ml.com/pmdarima/modules/generated/pmdarima.arima.auto_arima.html. Accessed 20 Dec 2023.

3.5 Long Short-Term Memory (LSTM)

LSTM, on the other hand, is a type of Recurrent Neural Network (RNN)[21] that addresses the limitations of conventional RNNs, such as vanishing and exploding gradients, enabling them to learn long-term dependencies in sequential data, which conventional RNNs fail to capture.[22]

3.5.1 RNN

RNNs are deep learning models that can be trained to process sequential data and give an output. Unlike traditional neural networks where dataflow is unidirectional, RNNs have a *feedback* mechanism allowing the data to flow in both directions, allowing them to retain past data for future use. Their ability to *'memorize'* makes RNNs suitable for applications needing the identification of dependencies and patterns in sequential data, such as time series forecasting, speech recognition, and natural language processing.[23]

RNNs are made of *neurons*, organised into *input*, *hidden*, and *output layers*. The *input layer* receives the incoming data and passes it to the *hidden layers*, one step at a time. The *hidden layer(s)* process this incoming data, combining it with *'memorized'* data to generate an *output* that is then passed to the *output layer*. The *feedback* loop in the *hidden layer(s)* allows them to retain previous *inputs* for combining with each incoming next *input* to generate an *output*

[21] Barla, Nilesh. "Recurrent Neural Network Guide: A Deep Dive in RNN." neptune.ai, 22 Aug. 2023, https://neptune.ai/blog/recurrent-neural-network-guide. Accessed 20 Dec 2023.

[22] Understanding LSTM Networks – Colah's Blog. https://colah.github.io/posts/2015-08-Understanding-LSTMs. Accessed 20 dec 2023.

[23] Kalita, Debasish. "A Brief Overview of Recurrent Neural Networks (RNN)." Analytics Vidhya, 7 Nov. 2023, www.analyticsvidhya.com/blog/2022/03/a-brief-overview-of-recurrent-neural-networks-rnn. Accessed 22 Dec 2023.

modulated by past observations, which is again fed back. This *recurrent feedback* mechanism gives RNNs the ability to learn from past data.[24]

However, when the incoming data changes too quickly or too slowly, an RNN may struggle to adjust its parameters appropriately, leading to the *exploding* and *vanishing* gradient problems, resulting in *overfitting* or *underfitting* of the model.[25]

3.5.2 LSTM Neural Networks

LSTM, a type of RNN with a modified architecture, addresses the above limitations of conventional RNNs by incorporating additional memory *cells* and *gates*, allowing them to retain or discard information selectively, making them suitable for applications such as time series forecasting, where long-term dependencies are prevalent.[26]

[24] Kalita, Debasish. "A Brief Overview of Recurrent Neural Networks (RNN)." Analytics Vidhya, 7 Nov. 2023, www.analyticsvidhya.com/blog/2022/03/a-brief-overview-of-recurrent-neural-networks-rnn. Accessed 22 Dec 2023.

[25] Kalita, Debasish. "A Brief Overview of Recurrent Neural Networks (RNN)." Analytics Vidhya, 7 Nov. 2023, www.analyticsvidhya.com/blog/2022/03/a-brief-overview-of-recurrent-neural-networks-rnn. Accessed 22 Dec 2023.

[26] Understanding LSTM Networks – Colah's Blog. https://colah.github.io/posts/2015-08-Understanding-LSTMs. Accessed 20 dec 2023.

3.5.2.1 Architecture

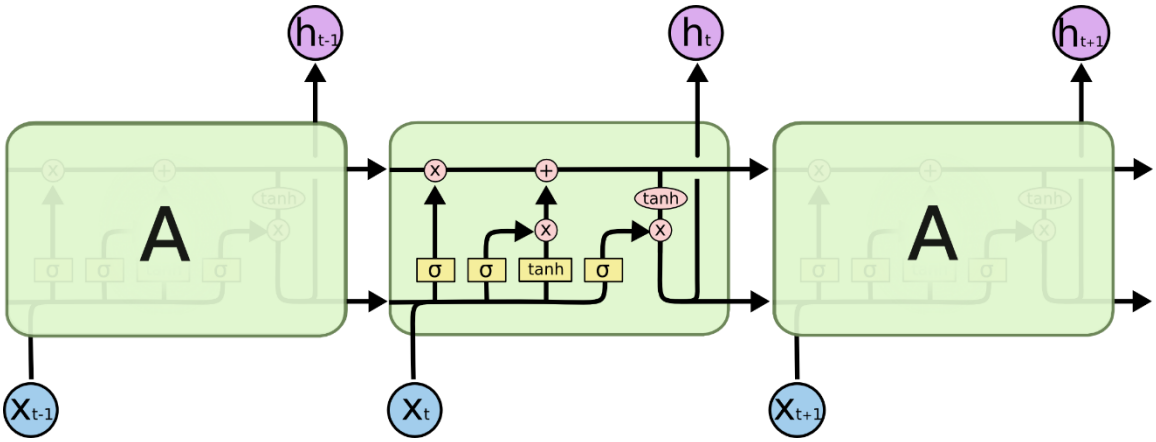


Figure 1: Architecture of LSTM (Source: Understanding LSTM Networks -- Colah's Blog)

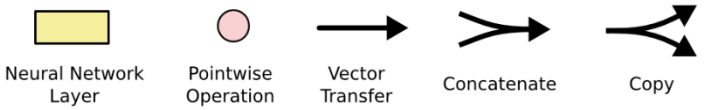


Figure 2: Symbol Notation in Fig 1 (Source: *Understanding LSTM Networks -- Colah's Blog*)

In the architectural diagram of LSTM given in Fig 1, each line shown carries an entire data vector, from the output of one node to the inputs of other nodes. The pink circles represent pointwise operations, like vector addition and multiplication, while the yellow boxes are learned neural network layers. Merging lines denote the concatenation of data, while forking lines denote the copying and distribution of data to different locations.[27]

[27] Understanding LSTM Networks – Colah's Blog. https://colah.github.io/posts/2015-08-Understanding-LSTMs. Accessed 20 dec 2023.

LSTMs have a chain-like structure with four layers comprising three gates and an update layer, which operate on the input data sequentially, as depicted in Figure 3. The sequence of operations is given in succeeding paras.

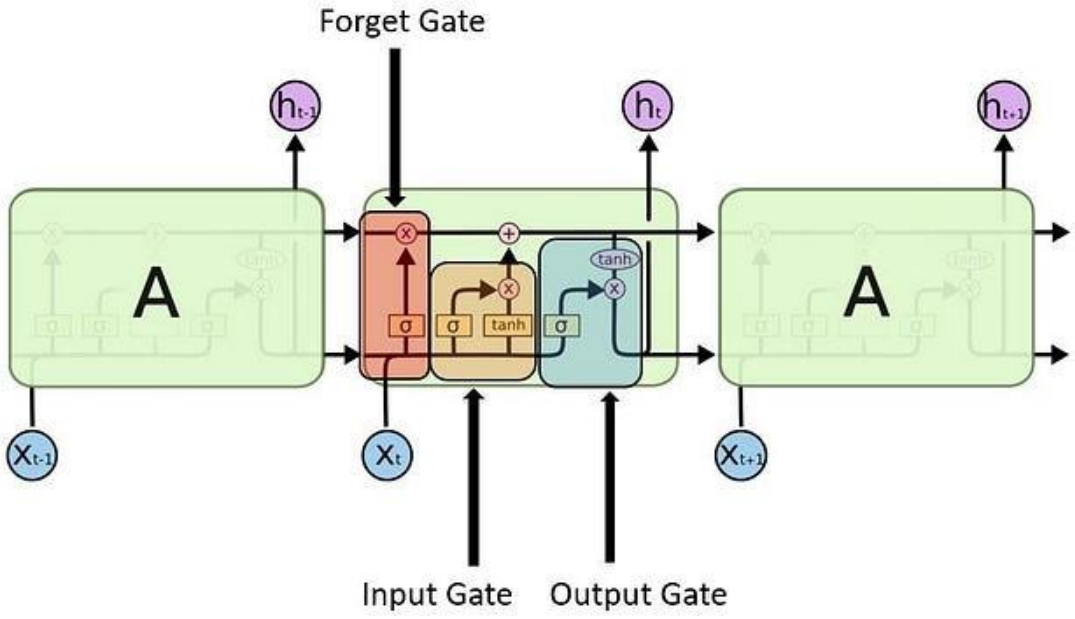
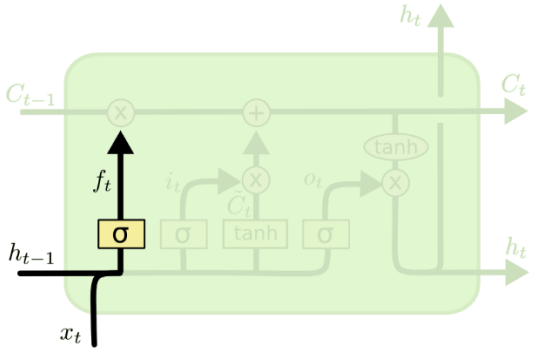


Figure 3: Schematic Diagram of LSTM Representing Gates. The horizontal line on top represents the cell state (Source: *Understanding LSTM Networks -- Colah's Blog*)

3.5.2.2 Operation

Forget Gate: The forget gate layer (Figure 4) processes the combination of the previous *hidden state* (h_{t-1}) and the current *input* (x_t). It decides which information from the cell state should be discarded and which should be passed on. It is a Sigmoid layer.[28]



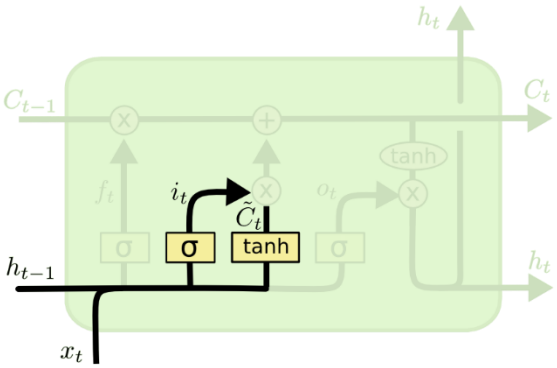
$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

Figure 4: Schematic Representation of LSTM Forget Gate (Source: *Understanding LSTM Networks -- Colah's Blog*)

Input Gate: The input gate layer (Figure 5) processes the combination of the previous *hidden state* (h_{t-1}) and the current *input* (x_t). It decides which new information should be stored in the cell state. It comprises a *sigmoid* layer (to determine which values i_t to update) and a *tanh* layer (to create a vector of new *candidate values* C_t).[29]

[28] Understanding LSTM Networks – Colah's Blog. https://colah.github.io/posts/2015-08-Understanding-LSTMs. Accessed 20 dec 2023.

[29] Understanding LSTM Networks – Colah's Blog. https://colah.github.io/posts/2015-08-Understanding-LSTMs. Accessed 20 dec 2023.

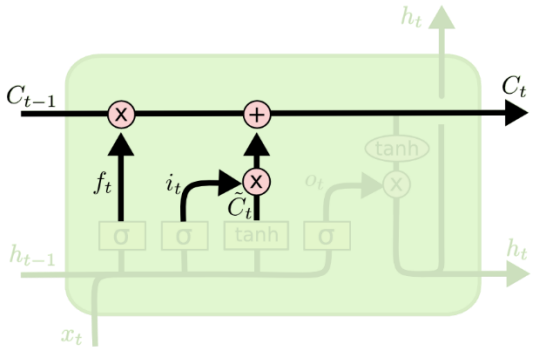


$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

Figure 5: Schematic Representation of LSTM Input Gate (Source: *Understanding LSTM Networks -- Colah's Blog*)

Cell State Update: The *cell state* (Figure 6) is updated by combining the information from the forget gate *(f_t)* and the information from the input gate *(C_{t-1})*. The forget gate decides what to remove from the cell state, and the input gate decides what to add. This updated cell state becomes the memory of the LSTM.[30]

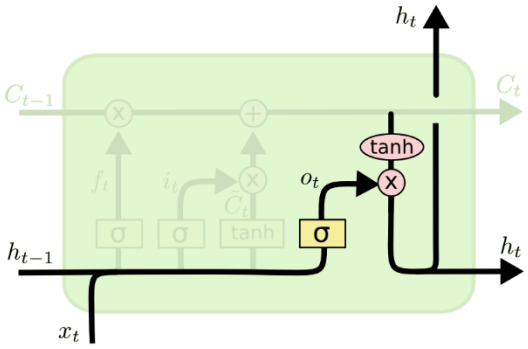


$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

Figure 6: Schematic Representation of LSTM Cell State Update Step (Source: *Understanding LSTM Networks -- Colah's Blog*)

[30] Understanding LSTM Networks – Colah's Blog. https://colah.github.io/posts/2015-08-Understanding-LSTMs. Accessed 20 dec 2023.

Output Gate: The output gate layer (Figure 7) processes the combination of the previous hidden state (h_{t-1}) and the current input (x_t), like the forget and input gates. It determines the next hidden state (h_t) based on the updated cell state. The output gate includes a sigmoid layer (to determine which values of the cell state to output) and a *tanh* layer (to transform the values between -1 and 1).[31]



$$o_t = \sigma\left(W_o\ [h_{t-1}, x_t]\ +\ b_o\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$

Figure 7: Schematic Representation of Output Gate (Source: *Understanding LSTM Networks* -- *Colah's Blog*)

3.5.3 Forecasting with LSTM Models

3.5.3.1 Parameters and Hyperparameters of LSTM

In LSTM, *parameters* and *hyperparameters* are two different but related concepts. The *model's hyperparameters* are top-level parameters that control the learning process and determine the model *parameters*. They are to be **determined by the model designer** before training begins and remain unchanged at the end of the learning process. The *hyperparameters* of the model include the number of hidden layers, number of neurons in each hidden layer, number of epochs,

[31] Understanding LSTM Networks – Colah's Blog. https://colah.github.io/posts/2015-08-Understanding-LSTMs. Accessed 20 dec 2023.

batch size, dropout rate, optimizer, loss function, stateful, shuffle, and reset states, each having an impact on the performance of the model.[32]

On the other hand, the *model's parameters* are internal to the model and are **learned by the model** during training based on the *data* and the *hyperparameters*. The *parameters* of the model include the *weights* and *biases* of the model, which are updated during training, unlike the *hyperparameters,* which remain unchanged.[33]

Optimal choice of the hyperparameters is crucial for the model to perform well.

3.5.3.2 Tuning the Hyperparameters of LSTM Models

The *designer usually manually determines the hyperparameters of LSTM models* based on domain expertise and experience. An alternative approach is to employ search algorithms such as grid search, random search, and Bayesian optimization to find the optimal combination of hyperparameters that minimizes the loss function.[34]

The *KerasTuner* library in Python provides a flexible and efficient way to perform *hyperparameter* tuning using *grid search* and *random search*, eliminating the need for custom implementation, saving time, and reducing the scope for errors.[35]

---

[32] Nyuytiymbiy, Kizito. "Parameters, Hyperparameters, Machine Learning | Towards Data Science." Medium, 7 Mar. 2023, https://towardsdatascience.com/parameters-and-hyperparameters-aa609601a9ac. Accessed 20 Dec 2023.

[33] Nyuytiymbiy, Kizito. "Parameters, Hyperparameters, Machine Learning | Towards Data Science." Medium, 7 Mar. 2023, https://towardsdatascience.com/parameters-and-hyperparameters-aa609601a9ac. Accessed 20 Dec 2023.

[34] Rendyk. "Tuning the Hyperparameters and Layers of Neural Network Deep Learning." Analytics Vidhya, 12 Jan. 2024, www.analyticsvidhya.com/blog/2021/05/tuning-the-hyperparameters-and-layers-of-neural-network-deep-learning. Accessed 21 Dec 2023.

[35] Team, Keras. Keras Documentation: KerasTuner API. https://keras.io/api/keras_tuner. Accessed 12 Dec 2023.

4 Experimental Methodology

The experimental setup for the study was implemented in Python 3.11 in a Jupyter Notebook environment. The steps followed to answer the research question are discussed below.

4.1 Step 1: Data Collection

Historical 5-year stock price data from 01 January 2018 to 01 January 2023 was obtained from Yahoo Finance, using the *yfinance* library, for the following ten companies in the S&P 500 index, representing a diverse set of sectors to avoid sector bias, account for macroeconomic factors, and to make the results more generalizable.[36] [37]

Ticker Symbol	Company Name	Sector
GOOG	Alphabet Inc.	Technology
JPM	JPMorgan Chase & Co.	Financial Services
JNJ	Johnson & Johnson	Healthcare
WMT	Walmart Inc.	Consumer Defensive
TSLA	Tesla Inc.	Automobiles
AMZN	Amazon.com Inc.	E-Commerce
BP	BP plc	Oil & Gas
NKE	Nike Inc.	Apparel

[36] "Yahoo Finance - Stock Market Live, Quotes, Business and Finance News." Yahoo Finance - Stock Market Live, Quotes, Business & Finance News, finance.yahoo.com.

[37] "Yfinance." PyPI, 21 Jan. 2024, pypi.org/project/yfinance.

Ticker Symbol	Company Name	Sector
KO	The Coca-Cola Company	Beverages
PFE	Pfizer Inc.	Pharmaceuticals

(Table 1: List of stocks chosen for the study)

A period of five years was chosen as it provided sufficient data for analysis, and the computational cost of the models was manageable.

4.2 Step 2: Data Visualization and Preprocessing

4.2.1 Data Cleaning

The data obtained from Yahoo Finance was cleaned by filling in missing values with the previous day's closing price, indexed by datetime, and sorted in ascending order. The *Adj Close* prices of the stocks were filtered for further analysis, as they account for post-market action, which can impact the price on the next trading day.

4.2.2 Data Visualization

The *Adj Close* prices, decomposed components (*trend*, *seasonality*, and *residual*), *ACF*, and *PACF* plots for each stock were plotted to visualize the data and identify any patterns.

The plots revealed the presence of *trends* and *seasonality* in the data, indicating that the data is not stationary. The plots for GOOG are provided below:

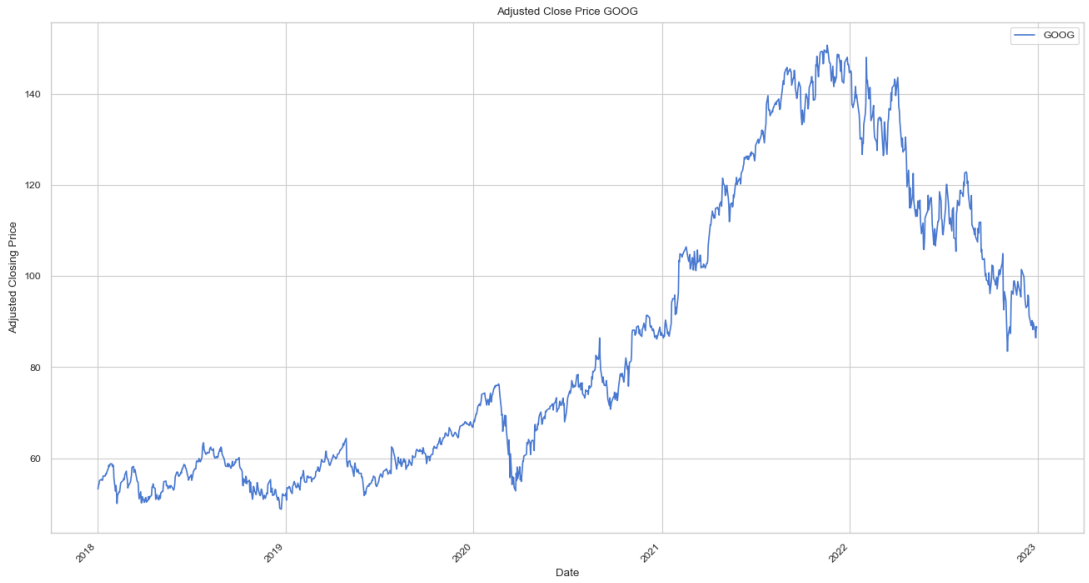


Figure 8: Adjusted Close Price of Alphabet Inc. (GOOG)

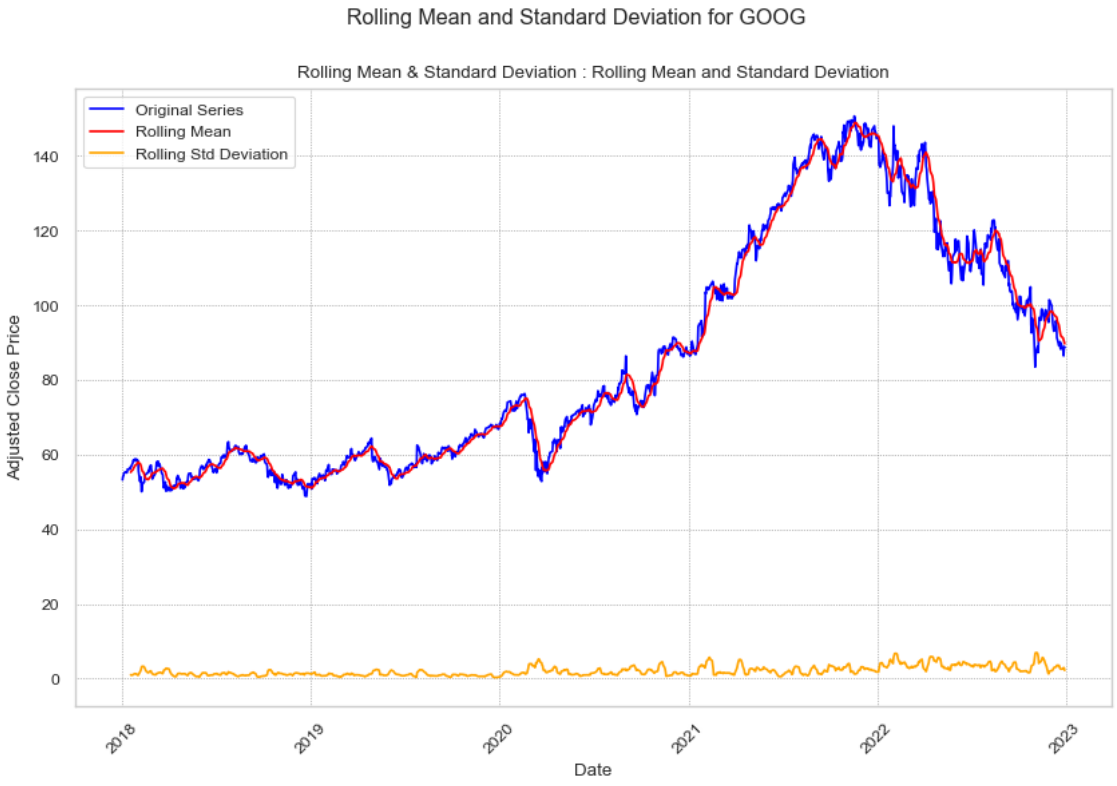


Figure 9: Plot of Rolling Mean and Standard Deviation of Adjusted Close Price of Alphabet Inc. (GOOG)

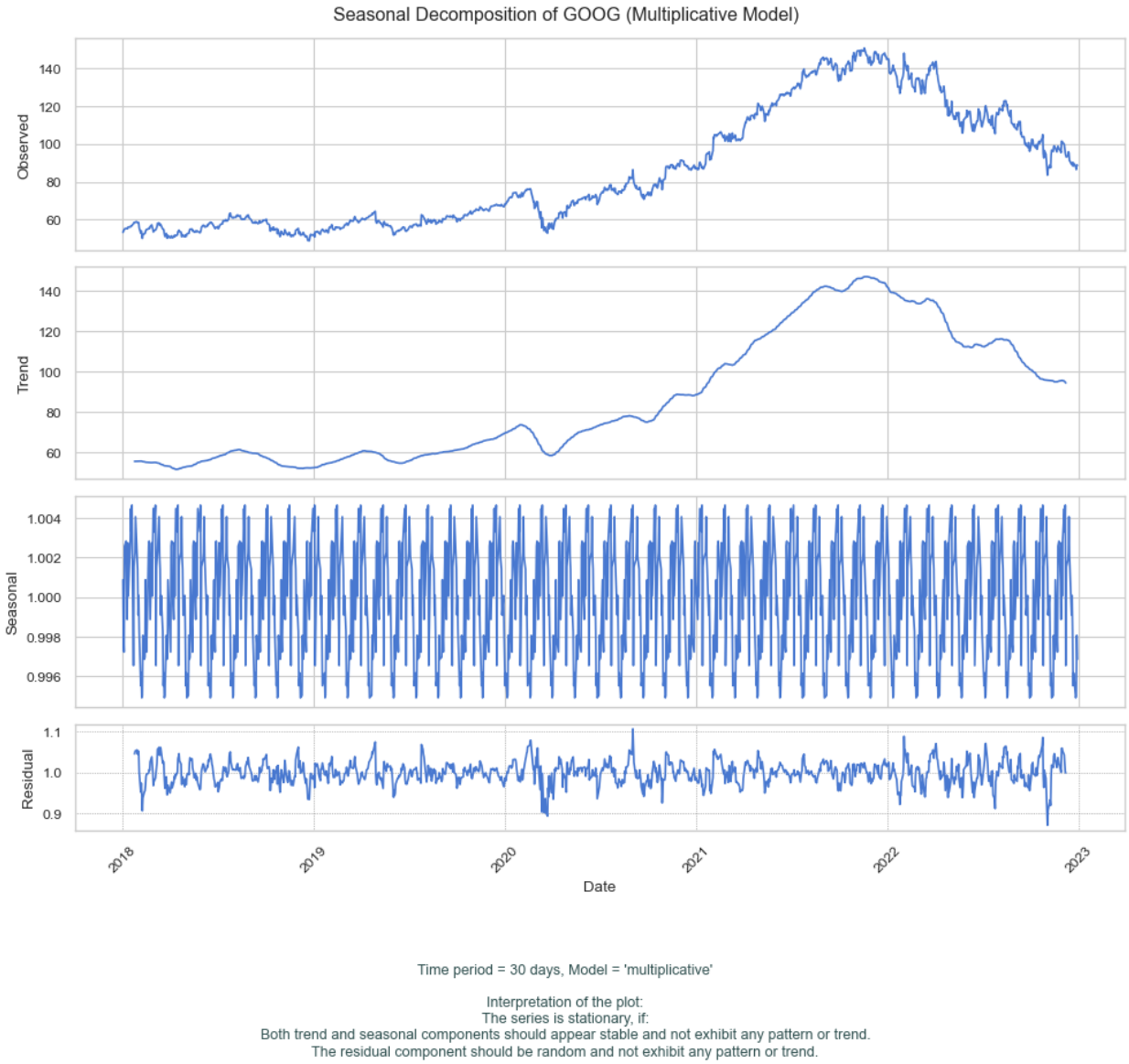


Figure 10: Decomposition of Adjusted Close Prices of GOOG to Trend, Seasonality and Residuals

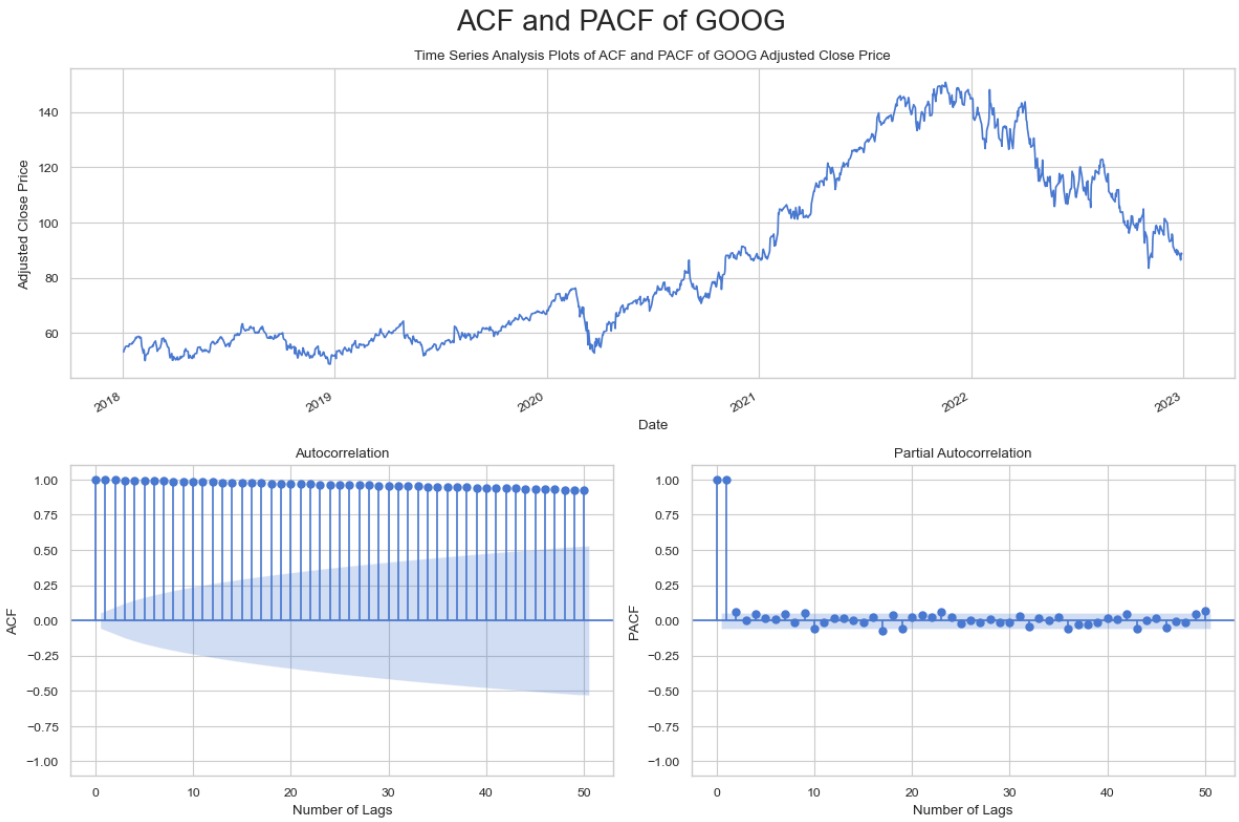


Figure 11: ACF and PACF plots of Adjusted Close Prices of GOOG

Statistical tests such as the *Augmented Dickey-Fuller* (*ADF*) and *Kwiatkowski-Phillips-Schmidt-Shin* (*KPSS*) using the *statsmodels* library also confirmed the same.

Summary of ADF and KPSS test results for GOOG are given in Fig 12 below:

```
Performing Augmented Dickey-Fuller Test on GOOG

Results of Augmented Dickey-Fuller Test:GOOG

---------------------------------

Test Statistic                -1.210835

p-value                       0.668918
```

```
#Lags Used                   1.000000
Number of Observations Used  1257.000000
Critical Value (1%)          -3.435563
Critical Value (5%)          -2.863842
Critical Value (10%)         -2.567996
dtype: float64
p-value: 0.6689176927500179
ADF test indicates that GOOG is not stationary as p-value is greater than 0.05
ADF test indicates that GOOG is not stationary as Test Statistic is greater than Critical Value in 5%
End of Augmented Dickey-Fuller Test
---------------------------------
Performing KPSS Test on GOOG
Results of KPSS Test:GOOG
---------------------------------
Test Statistic           4.654283
p-value                  0.010000
#Lags Used              21.000000
Critical Value (10%)     0.347000
Critical Value (5%)      0.463000
Critical Value (2.5%)    0.574000
Critical Value (1%)      0.739000
```

```
dtype: float64
KPSS test indicates that GOOG is not stationary as Test Statistic is greater than Critical Value in 5%
End of KPSS Test
---------------------------------
```

Figure 12: Output of Stationarity Test using ADF and KPSS Tests

4.2.3 Pre-Processing of Data

4.2.3.1 ARIMA

The data was made stationary by differencing the time series till effects of *trends* and *seasonality* were removed, i.e., *ADF* and *KPSS* tests returned True.

4.2.3.2 LSTM

In the case of LSTM, data was scaled to the range [0,1] and then split into the training and testing sets. *MinMaxScaler* function in the *sklearn* library was used to scale the data.

4.2.4 Splitting the Data into Training and Testing Sets

The differenced or scaled data of ARIMA and LSTM were split into training and testing data sets using a split ratio of 95:5.

4.3 Step 3: Model Implementation

The *training data* was fitted to the ARIMA and LSTM models, and the *testing data* was used to evaluate the performance of each model.

4.3.1 Finding Optimal Parameters for ARIMA

The number of times the data was differenced during the *pre-processing stage* to achieve stationarity determined the order of differencing *(d)*.

The ACF and PACF plots of the differenced time series were analyzed to determine the autoregression *(p)* and moving average *(q)* order, respectively.

The estimated values of *p, q and q* for Alphabet Inc. (GOOG) are *p=1, d=1 and q=1,* indicating that the ARIMA model for GOOG is given by *ARIMA(1,1,1).*

4.3.2 Estimation of *ARIMA Model Parameters Using pmdarima* Library

The manual approach discussed above involved a non-scalable and error-prone process due to its reliance on visualization and analysis of plots. Therefore, this method was applied solely to one stock, Alphabet Inc., to understand the process involved.

However, to obtain the experimental results for this study, the *auto_arima* function of the *pmdarima* library was preferred as it allowed for the determination of the optimal ARIMA model using a programmatic approach and offered benefits as discussed in sub-section 3.4.5 in the Theoretical Background.

The code used for experimentation is documented in Appendix A.

4.3.3 Hyperparameters Tuning for LSTM

Tuning *hyperparameters* in LSTM is a complex task requiring domain expertise, time, and computational resources. The *KerasTuner* library, designed for *hyperparameter optimization* and offering several benefits, as discussed in sub-section 3.5.3.2 in the Theoretical Background section, was employed to tune the LSTM model.

Stateful vs. Stateless LSTMs. A *stateful* LSTM was chosen for its ability to capture the *trends* and *seasonality* in the data, if any, and improve the accuracy of the predictions. *Keras* library was explicitly configured by setting the *stateful* parameter to *True*, the *batch_size* parameter to 1 to preserve the state within the same *epoch*, the *shuffle* parameter to *False* to preserve the *order* of the data, and the *reset_states* parameter to *True* to reset the *state* after each *epoch*.

4.4 Step 4: Prediction

The time horizon for predictions, for example, a daily, weekly, monthly, or yearly forecast, depends on the application's need. However, forecasting far into the future reduces prediction accuracy due to the absence of recent data and the compounding of errors.

A model's forecast accuracy can be improved by using a rolling forecast by feeding back the latest observed data to make the following prediction.[38]

This study employed a one-step rolling forecast to predict the stock price one day at a time, continuously updating the model with the latest available data.

4.5 Step 5: Evaluation of Models

The training data was fitted to the ARIMA and LSTM models, and the testing data was used to evaluate the model's performance. Predictions were made using a *one-step rolling forecast*

[38] Brownlee, Jason. "Time Series Forecasting With the Long Short-Term Memory Network in Python." *MachineLearningMastery.com*, 27 Aug. 2020, machinelearningmastery.com/time-series-forecasting-long-short-term-memory-network-python. Accessed 25 Dec 2023.

approach. The performance of the models was evaluated by comparing the predicted values with the actual values, using statistical metrics *RMSE*, *MAE*, and *MAPE*.

5 Results and Analysis

5.1 Performance Metrics on a Per-Stock Basis

The performance metrics obtained on a per-stock basis are tabulated below (output of Step 5 of Appendix A)

	RMSE		**MAPE**		**MAE**	
Symbol	ARIMA	LSTM	ARIMA	LSTM	ARIMA	LSTM
GOOG	2.5754	4.5959	2.0113	3.9893	1.9010	3.7554
JPM	2.1378	4.3924	1.3682	2.9673	1.6205	3.5093
JNJ	1.5489	2.6498	0.7146	1.3014	1.1852	2.1586
WMT	1.9617	3.2097	1.0048	1.8503	1.3913	2.5982
TSLA	8.2851	18.7155	3.4410	8.6554	6.2608	15.3805
AMZN	3.0969	5.7947	2.3246	4.7190	2.3032	4.6382
BP	0.5660	1.0318	1.4066	2.7565	0.4433	0.8559
NKE	2.7681	5.1574	1.9981	3.8467	1.9857	3.8372
KO	0.6674	1.2222	0.8930	1.5761	0.5143	0.9077
PFE	0.7239	1.2188	1.2161	2.2088	0.5469	0.9912

(Table 2: Comparison of ARIMA and LSTM performance on a per stock basis)

5.1.1 Interpretation of Performance Metrics on a Per Stock basis

From Table 2, it is seen that RMSE, MAE, and MAPE values are lower for ARIMA than LSTM across all selected stocks, suggesting that the ARIMA was more effective in predicting stock prices compared to LSTM for the underlying data.

This is likely to be owing to the following reasons:

- Time Series data of all the selected stocks could be made stationary by differencing, making it amenable for the application of ARIMA.
- ARIMA relies on regression analysis, which is well-suited for fitting curves to stationary data and apply forecasting methods.
- The forecasting was carried out on a one-step rolling basis, where the most recent data (previous day's Adjusted Closing Price) was available for forecasting the next day's stock price. As the next day's stock prices are usually very closely dependendent on the most recent data (barring exceptional scenarios), ARIMA which relies on regression-based analysis could predict the next day's stock price quite accurately.
- Further, LSTM is ideal for capturing long-term dependencies and complex patterns inherent in sequential data. In predicting the next day's stock prices, especially using a one-step rolling forecast method, ARIMA fared better as recency of data was equally or possibly more important than long-term dependencies.

5.2 Performance on Aggregate Basis

The results of the comparison on an aggregate basis are tabulated below (output of Step 5 of Appendix A):

Metric	ARIMA	LSTM	% Improvement	Remarks
Average RMSE	2.43311	4.79882	49.298%	ARIMA performed better
Average MAE	1.63783	3.38708	51.645%	-do-
Average MAPE	1.81522	3.8632	53.013%	-do-

(Table 3: Comparison of ARIMA and LSTM Performance on Aggregate Basis)

5.2.1 Interpretation of Aggregate Metrics

5.2.1.1 Overall Performance

ARIMA outperformed LSTM across all metrics—RMSE, MAE, and MAPE—at an aggregate level, showing approximately 49.3%, 51.6%, and 53.0% improvement, respectively. This indicates that ARIMA was more accurate in making predictions compared to LSTM for the considered dataset as a whole.

5.2.1.2 RMSE

A lower RMSE of ARIMA indicates that ARIMA could predict stock prices closer to the actual values than LSTM. This means that the average magnitude of errors in ARIMA's predictions was smaller compared to LSTM, resulting in more precise forecasts.

5.2.1.3 MAE

Lower observed MAE of ARIMA signifies that ARIMA could predict stock prices closer to the actual values on an absolute basis, and it made less biased predictions than LSTM. This suggests that ARIMA's predictions were, on average, closer to the true values, with less systematic overestimation or underestimation compared to LSTM.

5.2.1.4 MAPE

ARIMA had a lower aggregate MAPE than LSTM, implying that the ARIMA model's predicted values had lesser percentage deviations from actual values on an absolute basis. This indicates that ARIMA's predictions had, on average, smaller percentage errors compared to LSTM, making it a more reliable model for forecasting stock prices.

5.2.1.5 Overall Assessment

The likely reasons for better observed performance of ARIMA compared to LSTM on aggregate basis is also likely owing to similar reasons as explained in Section 5.1.1.

5.3 Visualization

Plots of predicted and actual test data values have been generated for all considered stocks for visualization and analysis (outputs of steps 4 and 5 of Appendix A). Plots for Alphabet Inc.(GOOG) are given below for discussion.

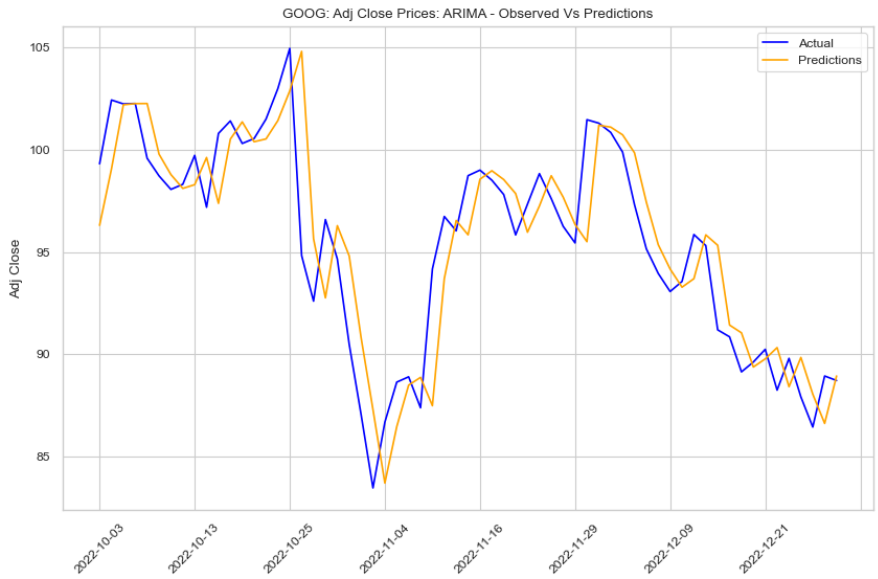


Figure 13: Actual vs Predicted Stock Prices GOOG: ARIMA (output of Step 3 of Appendix A)

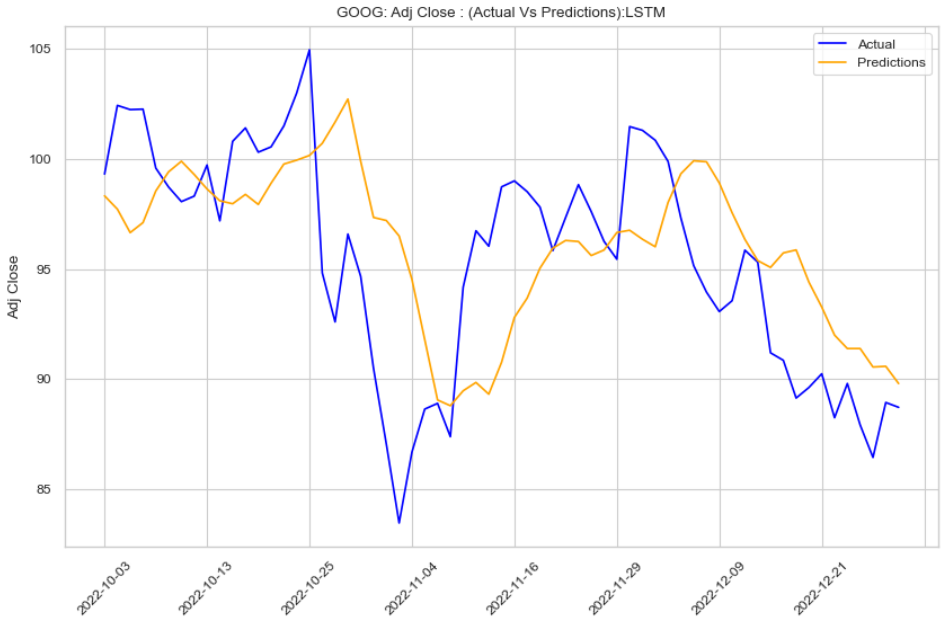


Figure 14: Actual Vs Predicted Stock Prices GOOG: LSTM (output of Step 4 of Appendix A)

5.3.1 Interpretation of Plots

The above plots for 'Actual versus Predicted Stock Prices' using ARIMA and LSTM demonstrate the ability of both models to predict stock prices to varying degrees of accuracy. It can also be seen that prices predicted by ARIMA followed the actual stock price more closely than LSTM. This is consistent with the observed performance metrics for GOOG using ARIMA and LSTM, RMSE (2.5754 vs 4.5959), MAPE (2.0113 Vs 3.9893) and MAE (1.9010 Vs 3.7554) respectively.

Further, it is noteworthy that predictions using ARIMA tended follow the actual prices more immediately compared to LSTM. This behaviour is attributable primarily to the employment of one-step rolling forecast method. LSTM exhibited a slower response to sharp changes in stock prices. This behaviour is likely attributable to several factors inherent in LSTM's model

architecture, including its ability to capture long-term dependencies and the incorporation of such dependencies into its predictions. Consequently, LSTM may have moderated the influence of recently observed data, resulting in a slower adjustment to changes in stock prices.

5.3.2 Plots of 'Actual versus Predicted Prices (using ARIMA and LSTM)' of Other Stocks

Plots of other stocks obtained as output from Steps 3 and 4 of Appendix A are listed below. These show similar patterns consistent with the above analysis.

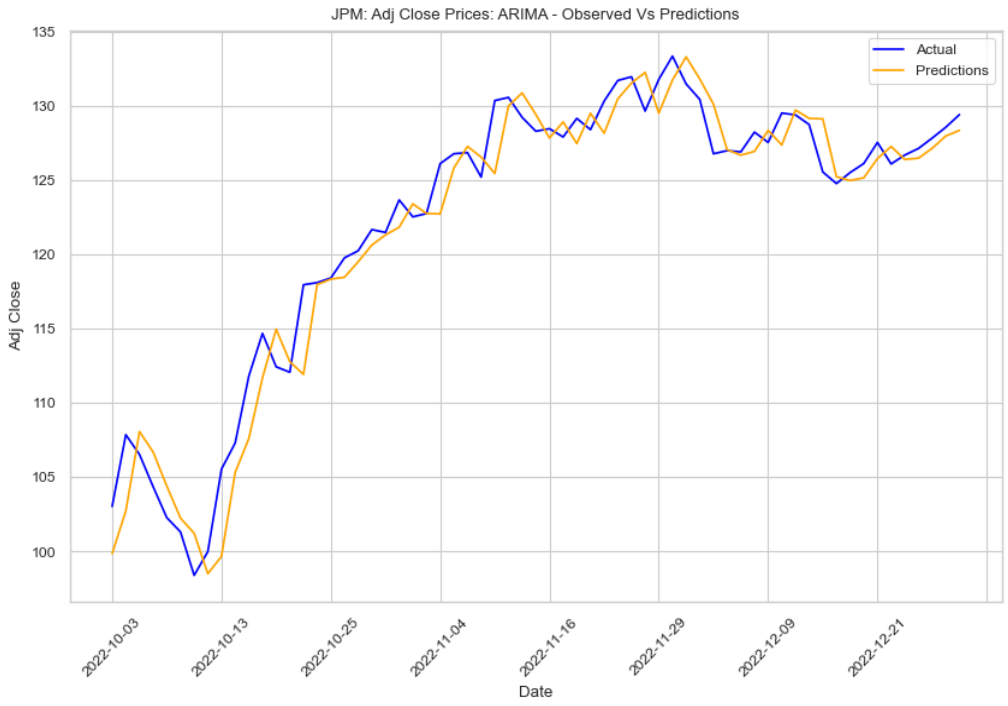


Figure 15: Actual vs Predicted Stock Prices JPM: ARIMA (output of Step 3 of Appendix A)

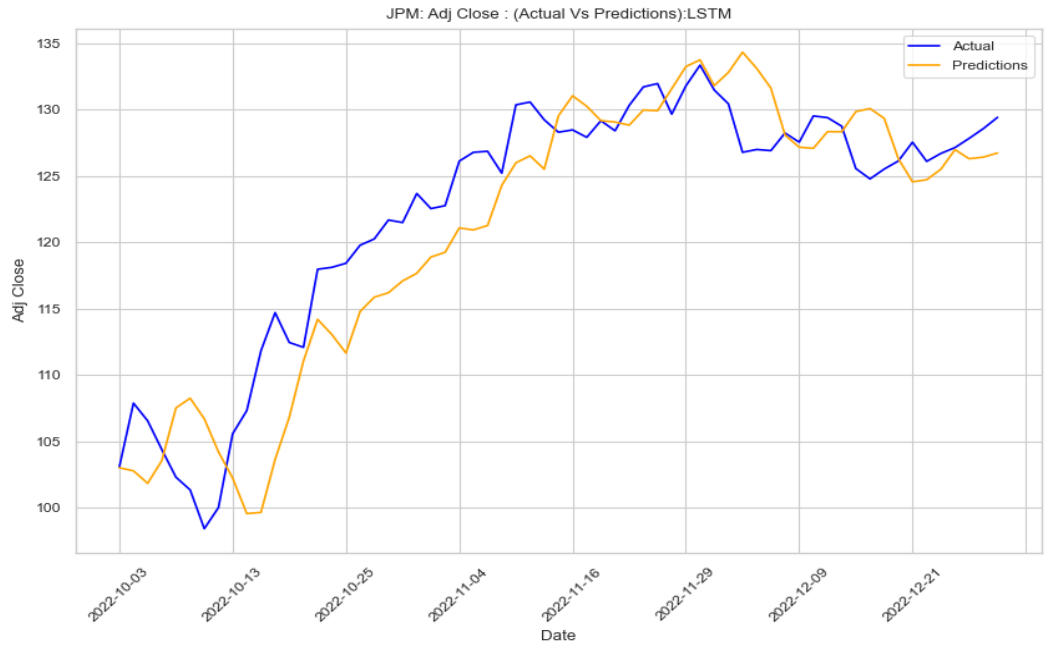


Figure 16: Actual vs Predicted Stock Prices JPM: LSTM (output of Step 4 of Appendix A)

Figure 17: Actual vs Predicted Stock Prices JNJ: ARIMA (output of Step 3 of Appendix A)

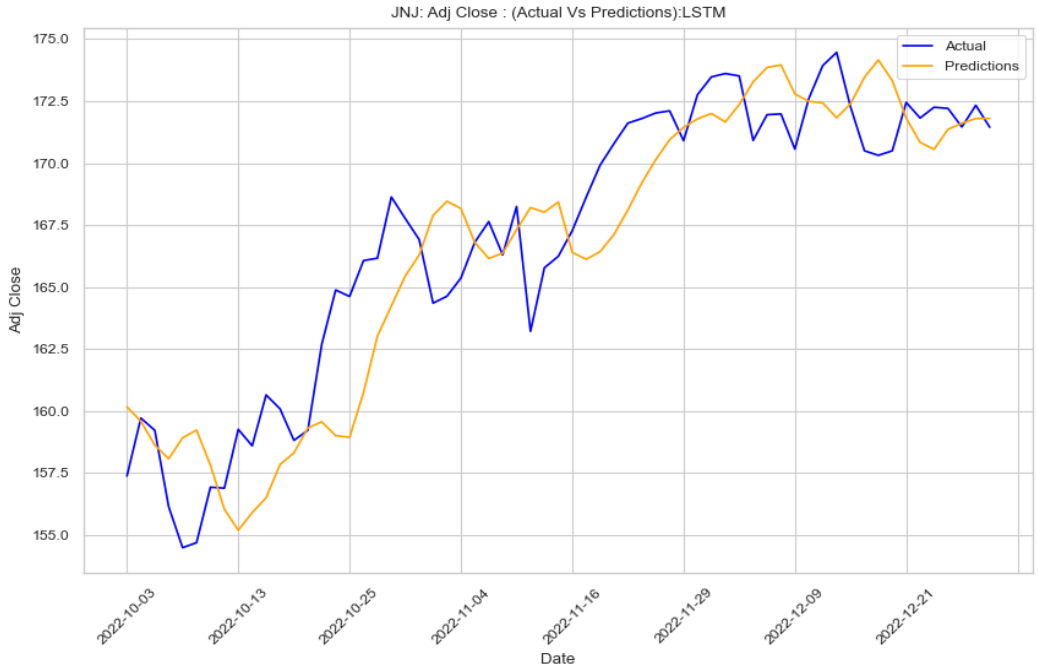


Figure 18: Actual vs Predicted Stock Prices JNJ: LSTM (output of Step 4 of Appendix A)

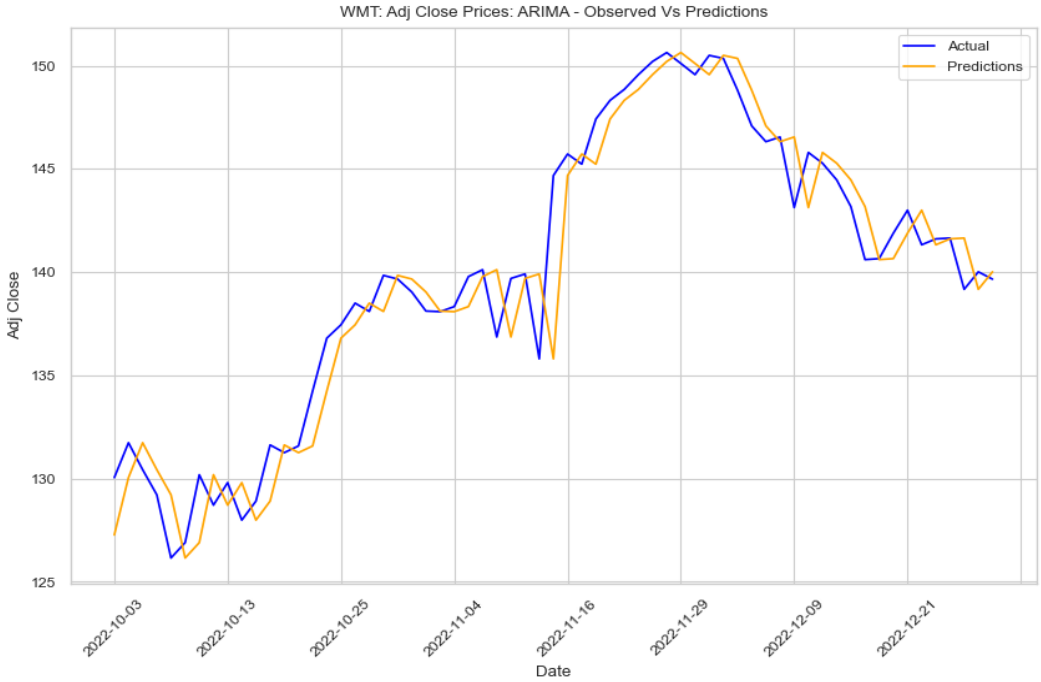


Figure 19: Actual vs Predicted Stock Prices WMT: ARIMA (output of Step 3 of Appendix A)

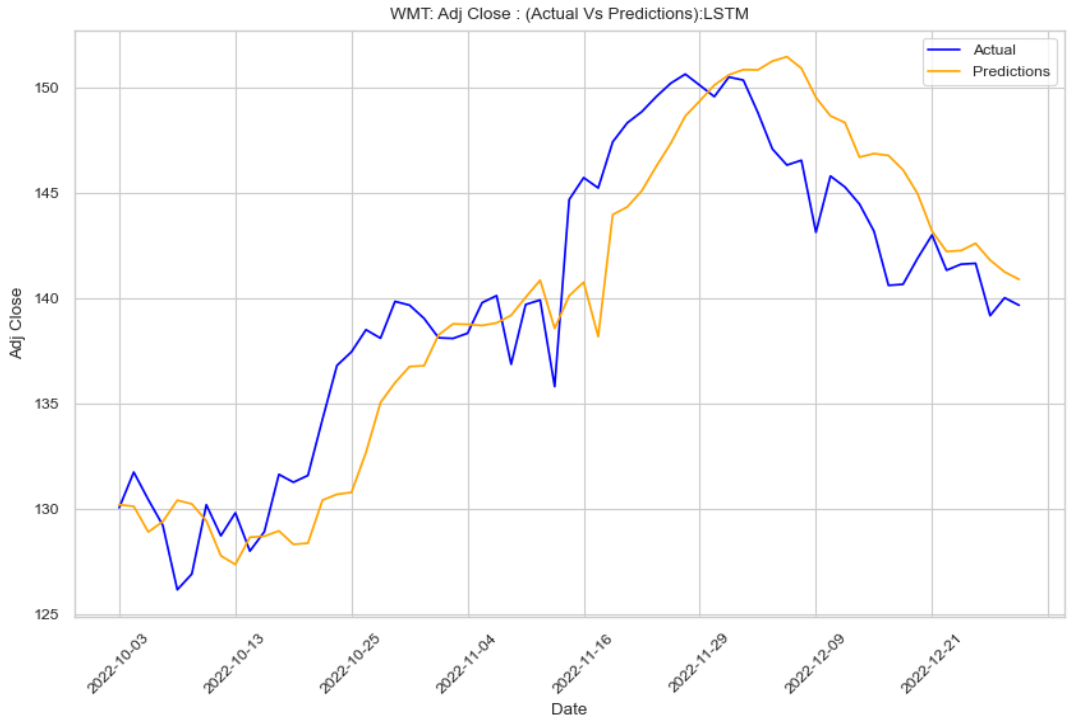


Figure 20: Actual vs Predicted Stock Prices WMT: LSTM (output of Step 4 of Appendix A)

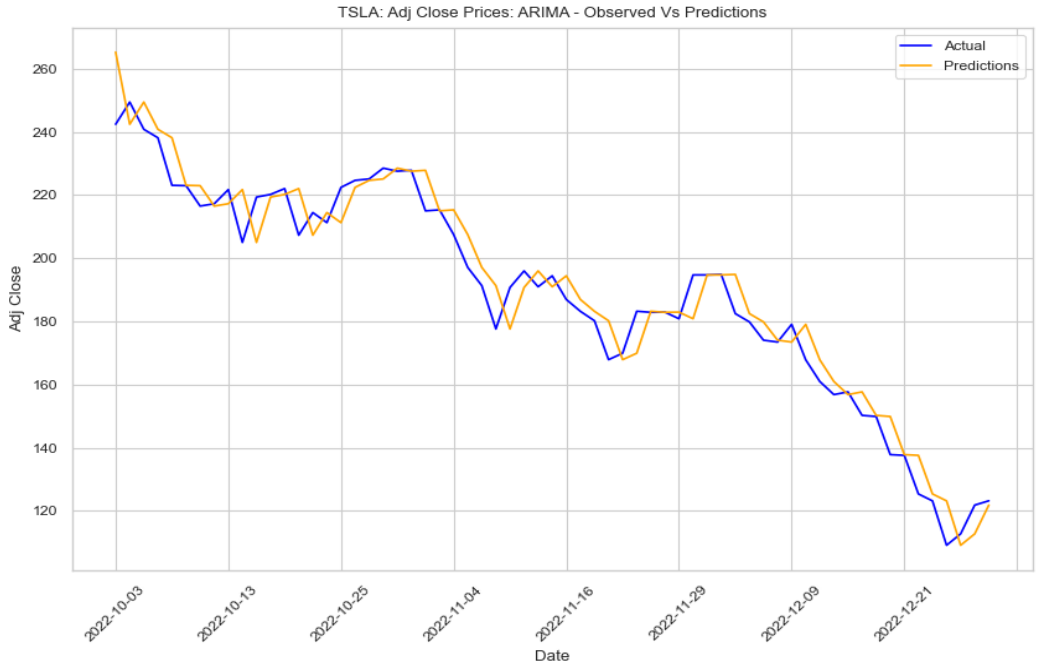


Figure 21: Actual vs Predicted Stock Prices TSLA: ARIMA (output of Step 3 of Appendix A)

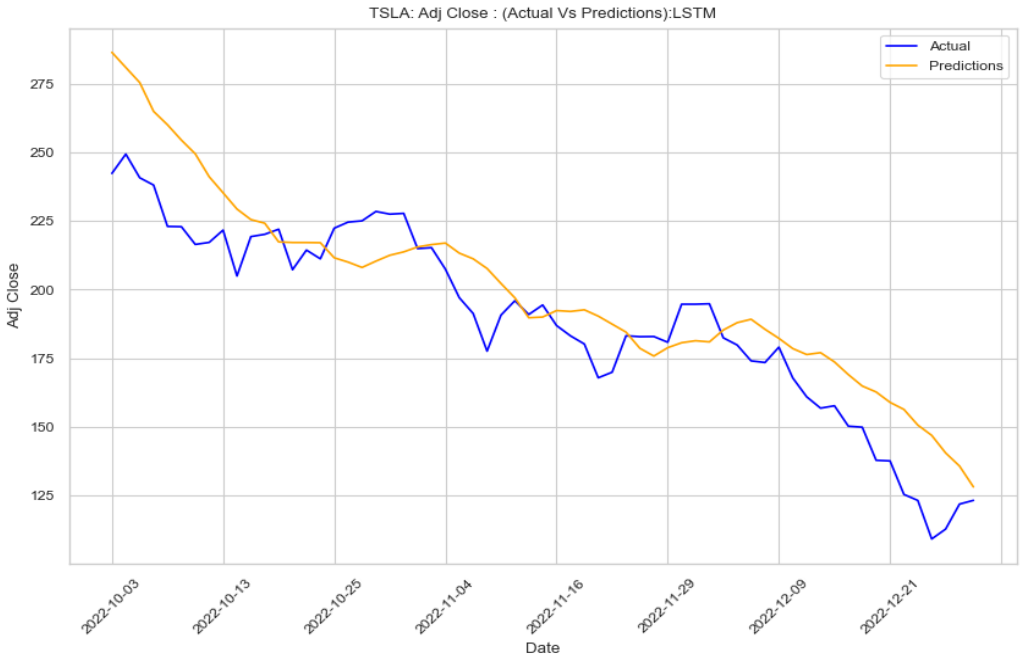


Figure 22: Actual vs Predicted Stock Prices TSLA: LSTM (output of Step 4 of Appendix A)

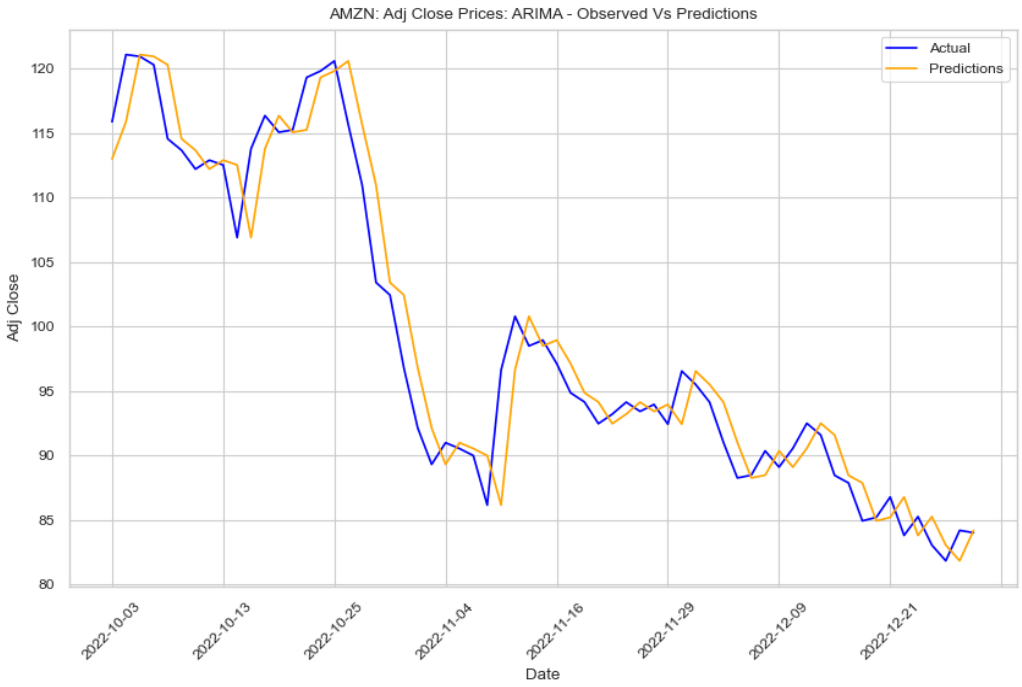


Figure 23: Actual vs Predicted Stock Prices AMZN: ARIMA (output of Step 3 of Appendix A)

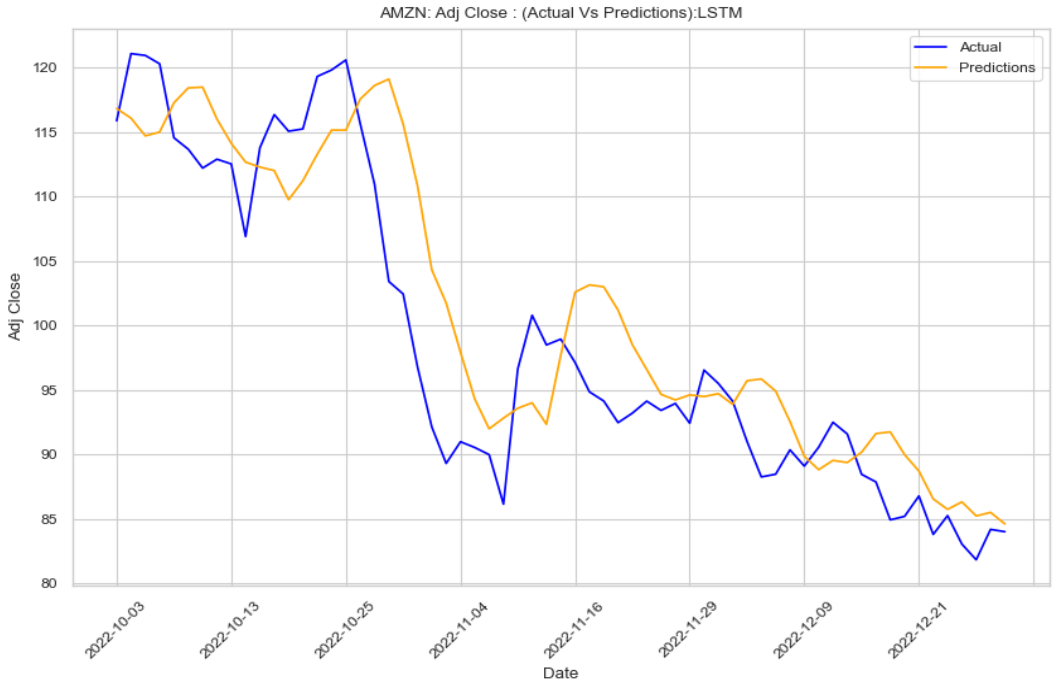


Figure 24: Actual vs Predicted Stock Prices AMZN: LSTM (output of Step 4 of Appendix A)

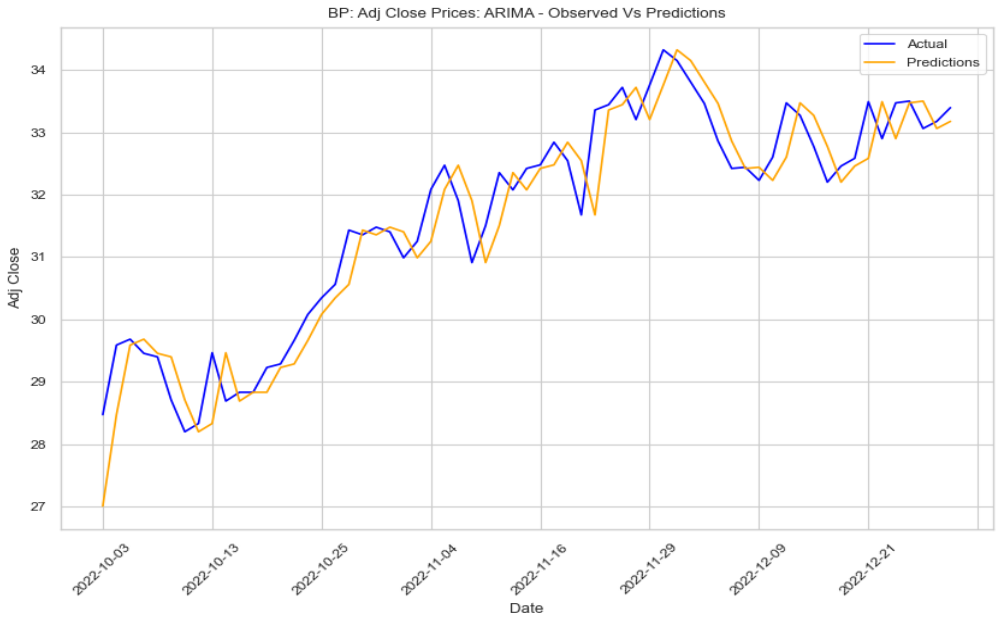


Figure 25: Actual vs Predicted Stock Prices BP: ARIMA (output of Step 3 of Appendix A)

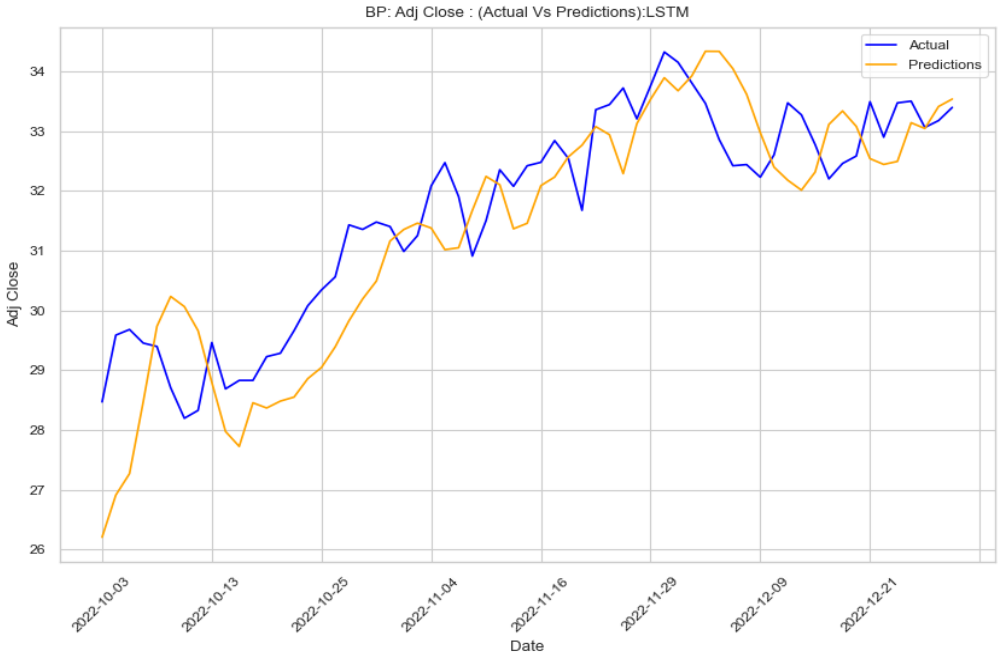


Figure 26: Actual vs Predicted Stock Prices BP: LSTM (output of Step 4 of Appendix A)

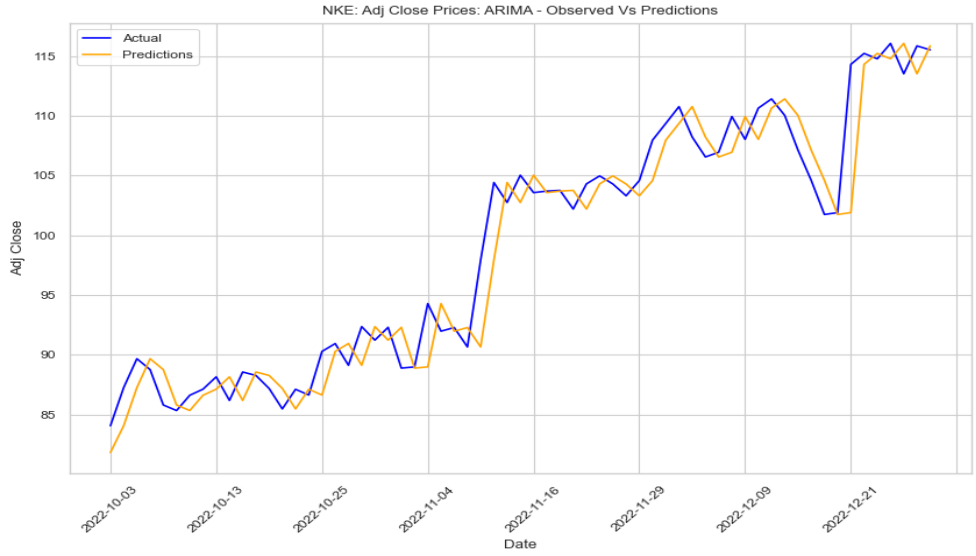


Figure 27: Actual vs Predicted Stock Prices NKE: ARIMA (output of Step 3 of Appendix A)

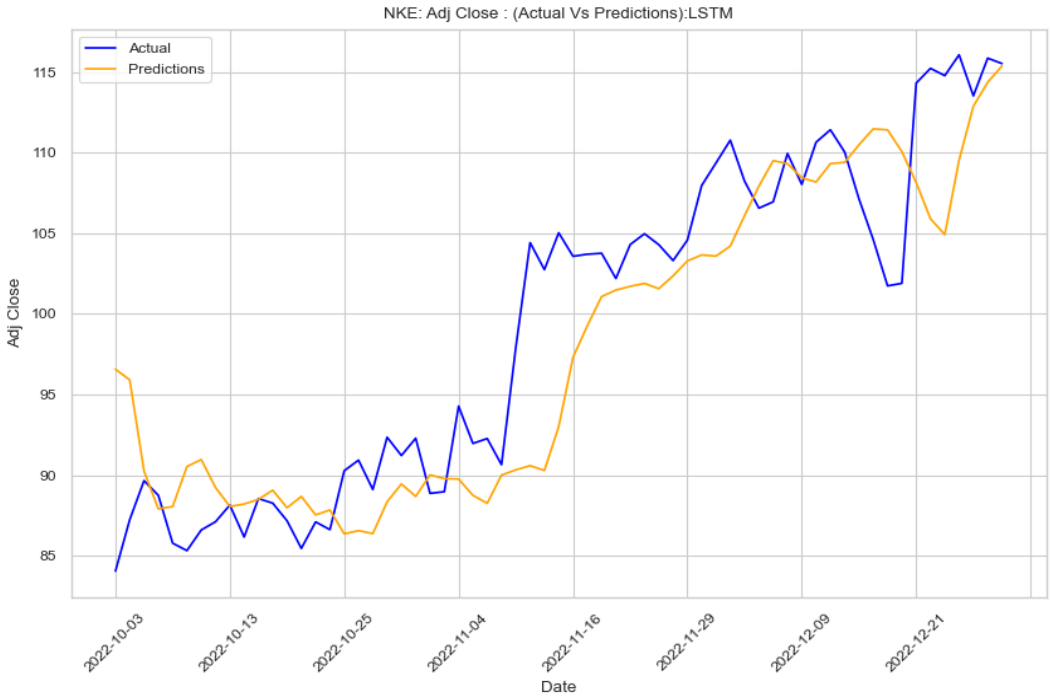


Figure 28: Actual vs Predicted Stock Prices NKE: LSTM (output of Step 4 of Appendix A)

Figure 29: Actual vs Predicted Stock Prices KO: ARIMA (output of Step 3 of Appendix A)

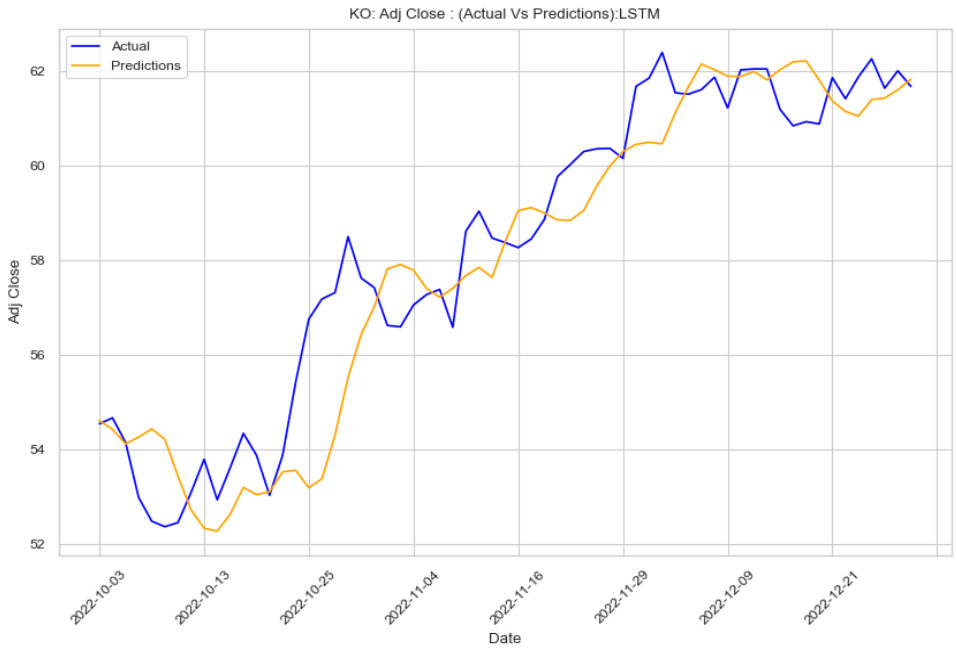


Figure 30: Actual vs Predicted Stock Prices KO: LSTM (output of Step 4 of Appendix A)

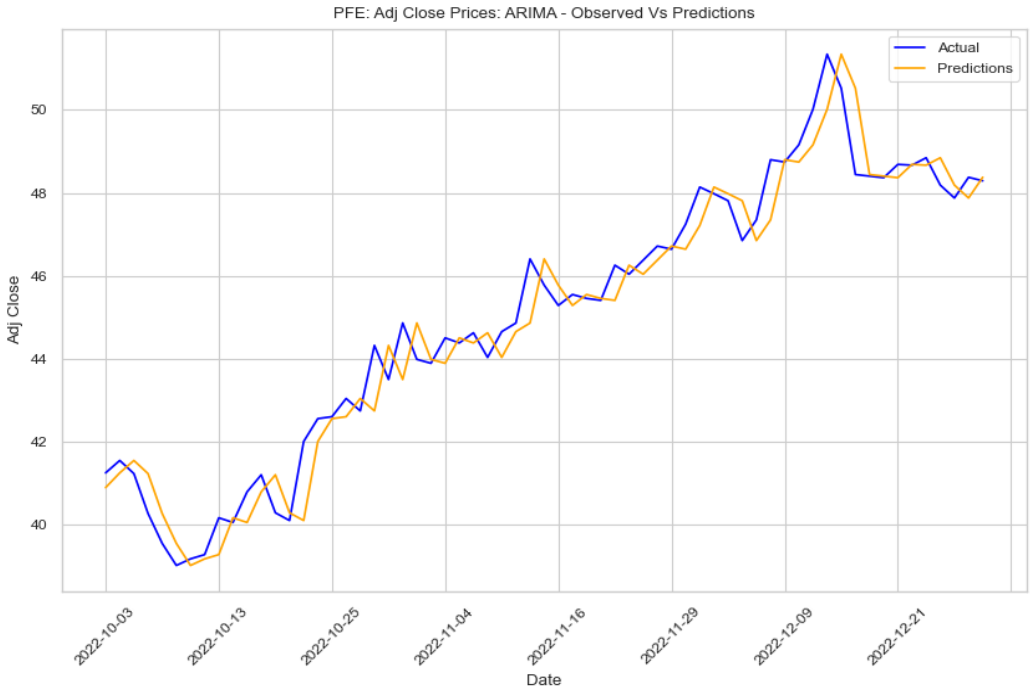


Figure 31: Actual vs Predicted Stock Prices PFE: ARIMA (output of Step 3 of Appendix A)

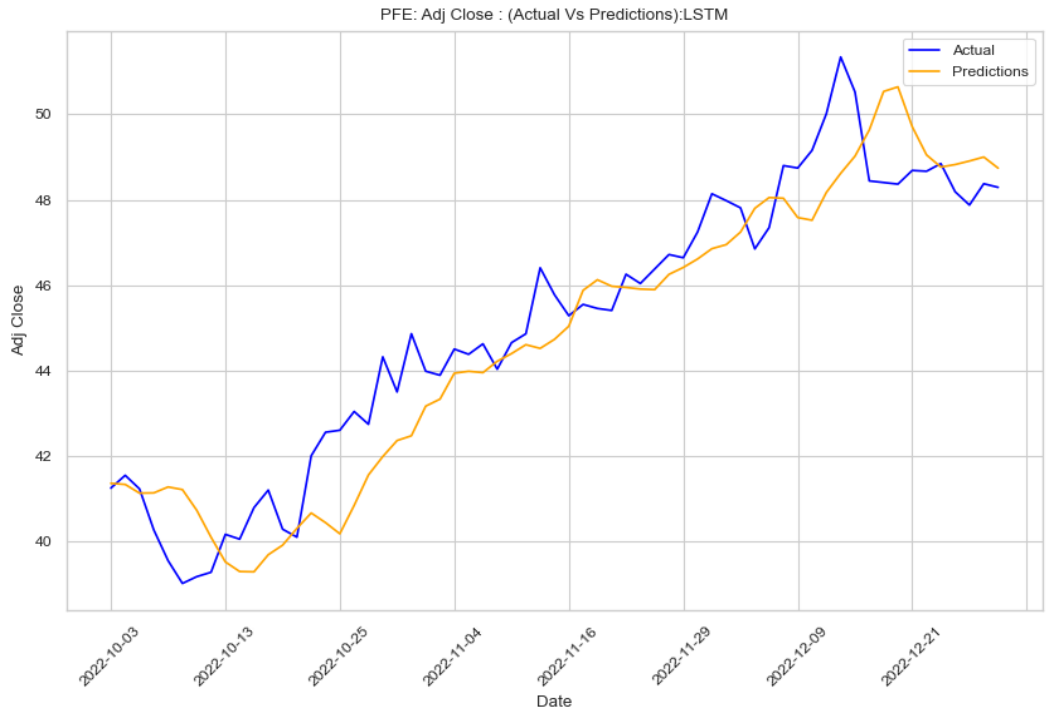


Figure 32: Actual vs Predicted Stock Prices PFE: LSTM (output of Step 4 of Appendix A)

6 Conclusion

It can be concluded, based on empirical evidence, that ARIMA can predict stock prices more accurately than LSTM.

However, the accuracy of stock price predictions is sensitive to various factors, such as the underlying stock data, the values assigned to ARIMA parameters, the chosen LSTM architecture, and the tuning of its hyperparameters. Further, the methodology used for predicting stock prices, the frequency of updating the model with the observed values and the forecast period also have a significant impact on the accuracy of predictions.

Therefore, it would be incorrect to generalize the above conclusion and extend it to all situations.

This possibly explains the reasons for continued exploration of this topic by the researchers, each investigation coming up with different findings.

7 Limitations and Future Work

In the conduct of this study, predictions were made solely based on historical data of a single variable, namely, the adjusted close price. However, it is common knowledge that stock prices are influenced by various factors, such as the macro and microeconomic data, the company's financial performance, government policies, market sentiment and natural and manmade disasters. Therefore, alternate approaches employing models which can incorporate multiple variables, such as multivariate LSTM and hybrid models, could yield more accurate results. Exploration of such models was beyond the current scope and has been marked for future research.

In this extended essay, a one-step rolling forecast method was used, with the model predicting the next day's stock price and thereafter being updated with the observed value prior to making the next prediction. While this approach served to standardize the comparison of ARIMA and LSTM, practical considerations may necessitate making stock price predictions on a weekly, monthly, yearly, or any other time period. The code developed for the study can be extended with minor modifications to extend the study to compare the performance of ARIMA and LSTM for such arbitrary time periods. Such a study can add more insights into understanding the overall performances of ARIMA and LSTM.

Lastly, as mentioned in various sections of the paper, tuning parameters and hyperparameters and choosing an optimal model have a significant impact on the experimental outcomes. A more

thorough study to understand the impact on the accuracy of predictions by varying hyperparameters is needed. This needs a deeper understanding of the domain and further study; thus, the same has been reserved for future exploration.

8 Bibliography

1. Kobiela, Dariusz, et al. “ARIMA Vs LSTM on NASDAQ Stock Exchange Data.” *Procedia Computer Science*, vol. 207, Jan. 2022, pp. 3836–45. https://doi.org/10.1016/j.procs.2022.09.445. Accessed 10 Dec 2023.

2. Ma, Qihang. “Comparison of ARIMA, ANN and LSTM for Stock Price Prediction.” *E3S Web of Conferences*, vol. 218, Jan. 2020, p. 01026. https://doi.org/10.1051/e3sconf/202021801026. Accessed 10 Dec 2023

3. S. Siami-Namini, N. Tavakoli and A. Siami Namin, "A Comparison of ARIMA and LSTM in Forecasting Time Series," *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*, Orlando, FL, USA, 2018, pp. 1394-1401, doi: 10.1109/ICMLA.2018.00227. PDF available at https://sci-hub.se/10.1109/ICMLA.2018.00227. Accessed 10 Dec 2023.

4. Gupta, Sakshi. “What Is Time Series Forecasting? Overview, Models &Amp; Methods.” *Springboard Blog*, 28 Sept. 2023, https://www.springboard.com/blog/data-science/time-series-forecasting/. Accessed 10 Dec 2023.

5. Fuqua School of Business. *Introduction to ARIMA Models*. https://people.duke.edu/~rnau/411arim.htm. Accessed 10 Dec 2023.

6. “Box-Jenkins Methodology.” *Columbia University Mailman School of Public Health*, 3 Oct. 2022, www.publichealth.columbia.edu/research/population-health-methods/box-jenkins-methodology.

7. Iamleonie. “Time Series: Interpreting ACF and PACF.” *Kaggle*, 15 Mar. 2022, http://www.kaggle.com/code/iamleonie/time-series-interpreting-acf-and-pacf. Accessed 12 Dec 2023.

8. TrainDataHub. “How to Interpret ACF and PACF Plots for Identifying AR, MA, ARMA, or ARIMA Models.” *Medium*, 2 Aug. 2022, medium.com/@ooemma83/how-to-interpret-acf-and-pacf-plots-for-identifying-ar-ma-arma-or-arima-models-498717e815b6. Accessed 12 Dec 2023.

9. Stationarity and Detrending (ADF/KPSS) - *Statsmodels 0.15.0 (+200)*. www.statsmodels.org/dev/examples/notebooks/generated/stationarity_detrending_adf_kpss.html.

10. Brownlee, Jason. “Probabilistic Model Selection With AIC, BIC, and MDL.” *MachineLearningMastery.com*, 27 Aug. 2020, https://machinelearningmastery.com/probabilistic-model-selection-measures .20 Dec 2023.

11. Sumi. “Understand ARIMA and Tune P, D, Q.” *Kaggle*, 20 Aug. 2018, www.kaggle.com/code/sumi25/understand-arima-and-tune-p-d-q. Accessed 22 Dec 2023.

12. Brownlee, Jason. “How to Create an ARIMA Model for Time Series Forecasting in Python.” *MachineLearningMastery.com*, 18 Nov. 2023, machinelearningmastery.com/arima-for-time-series-forecasting-with-python. Accessed 22 Dce 2023.

13. Nissa, Nuzulul Khairu. “Stock Price Prediction Using Auto-ARIMA - Nuzulul Khairu Nissa - Medium.” *Medium*, 16 Dec. 2021, nzlul.medium.com/stock-price-prediction-using-auto-arima-5569fcceae59. Accessed 27 dec 2023.

14. Hayes, Adam. “What Is a Time Series and How Is It Used to Analyze Data?” Investopedia, 13 June 2022, www.investopedia.com/terms/t/timeseries.asp.

15. *Time Series Analysis Tsa - Statsmodels 0.14.1*. www.statsmodels.org/stable/tsa.html#descriptive-statistics-and-tests. Accessed 27 Dec 2023.

16. *statsmodels.tsa.arima.model.ARIMA - Statsmodels 0.14.1*. www.statsmodels.org/stable/generated/statsmodels.tsa.arima.model.ARIMA.html. Accessed 27 Dec 2023.

17. *pmdarima.arima.auto __ _arima — Pmdarima 2.0.4 Documentation*. https://alkaline-ml.com/pmdarima/modules/generated/pmdarima.arima.auto_arima.html. Accessed 20 Dec 2023.

18. “Yfinance.” PyPI, 21 Jan. 2024, pypi.org/project/yfinance.

19. “Yahoo Finance - Stock Market Live, Quotes, Business and Finance News.” Yahoo Finance - Stock Market Live, Quotes, Business & Finance News, finance.yahoo.com.

20. Barla, Nilesh. “Recurrent Neural Network Guide: A Deep Dive in RNN.” *neptune.ai*, 22 Aug. 2023, https://neptune.ai/blog/recurrent-neural-network-guide. Accessed 20 Dec 2023.

21. Kalita, Debasish. “A Brief Overview of Recurrent Neural Networks (RNN).” *Analytics Vidhya*, 7 Nov. 2023, www.analyticsvidhya.com/blog/2022/03/a-brief-overview-of-recurrent-neural-networks-rnn. Accessed 22 Dec 2023.

22. Understanding LSTM Networks – *Colah’s Blog*. https://colah.github.io/posts/2015-08-Understanding-LSTMs. Accessed 20 dec 2023.

23. Özlü, Ahmet. “Long Short Term Memory (LSTM) Networks in a Nutshell.” *Medium*, 14 Dec. 2021, https://ahmetozlu.medium.com/long-short-term-memory-lstm-networks-in-a-nutshell-363cd470ccac. Accessed 20 dec 2023.

24. Nyuytiymbiy, Kizito. “Parameters, Hyperparameters, Machine Learning | Towards Data Science.” *Medium*, 7 Mar. 2023, https://towardsdatascience.com/parameters-and-hyperparameters-aa609601a9ac. Accessed 20 Dec 2023.

25. Rendyk. “Tuning the Hyperparameters and Layers of Neural Network Deep Learning.” *Analytics Vidhya*, 12 Jan. 2024, www.analyticsvidhya.com/blog/2021/05/tuning-the-hyperparameters-and-layers-of-neural-network-deep-learning. Accessed 21 Dec 2023.

26. Gorodetski, Maria. “Hyperparameter Tuning Methods - Grid, Random or Bayesian Search? | Towards Data Science.” *Medium*, 5 Jan. 2022, https://towardsdatascience.com/bayesian-optimization-for-hyperparameter-tuning-how-and-why-655b0ee0b399. Accessed 21 Dec 2023.

27. Team, Keras. *Keras Documentation: KerasTuner API*. https://keras.io/api/keras_tuner. Accessed 12 Dec 2023.

28. “Introduction to the Keras Tuner.” *TensorFlow*, https://www.tensorflow.org/tutorials/keras/keras_tuner. Accessed 10 Dec 2023.

29. Quant, Ai. “Mastering Stock Price Prediction With Deep Learning and Keras Tuner | Artificial Intelligence in Plain English.” *Medium*, 22 Apr. 2023, ai.plainenglish.io/mastering-stock-price-prediction-with-deep-learning-and-keras-tuner-optimizing-hyperparameters-for-66fca4d6525a. Accessed 27 Dec 2023.

30. Mingboi. “Rolling Multi-step Forecasts With LSTM, RNN, GRU.” *Kaggle*, 29 July 2021, www.kaggle.com/code/mingboi/rolling-multi-step-forecasts-with-lstm-rnn-gru. Accessed 25 Dec 2023.

31. Brownlee, Jason. “Time Series Forecasting With the Long Short-Term Memory Network in Python.” *MachineLearningMastery.com*, 27 Aug. 2020, machinelearningmastery.com/time-series-forecasting-long-short-term-memory-network-python. Accessed 25 Dec 2023.

9 Appendix A: Performance Evaluation of ARIMA and LSTM in Stock Price Prediction

This Appendix contains the code for carrying out the Performance evelaution of ARIMA and LSTM in predicting stock prices. Steps Involved in the Analysis:

1. Importing the required libraries and setting the configuration parameters.
2. Importing the dataset from Yahoo Finance using yfinance library and saving it to a CSV file for later use.
3. Perform rolling forecast ARIMA modeling on the dataset for each stock. This invoves:
   - Data Preprocessing involving:
     - Loading the CSV file into a Pandas DataFrame.
     - Checking for missing values and filling them with the previous day’s values.
     - Sorting the data in ascending order of date.
     - Converting the index to a datetime object.
     - Filtering the data to include only the Date and Adj Close columns which will be used for analysis.
     - Converting the Adj Close price to a float32 type, as it speeds up the computation and is the default type for auto.arima() function.
   - Splitting the data into train and test sets, and visualizing the train and test sets.
   - Building the ARIMA model using auto.arima() function with necessary parameters for optimization.
   - Predicting the stock price using the ARIMA model One-Step Rolling Forecast one day at a time for the test set.
   - Evaluating the model performance using the root mean squared error (RMSE), mean absolute error (MAE), and mean absolute percentage error (MAPE) metrics and time taken for model training and prediction.
   - Printing the metrics and time taken for model training and prediction.

- Visualizing the predictions by plotting the predicted and actual stock prices for the test set along with the metrics.
- Visualizing the residuals by plotting the residuals and density plot of the residuals.
- Saving the predictions to a CSV file for further analysis
- Saving the performance metrics to a CSV file for further analysis.
- Saving the plots to PNG files for further analysis.
- Calulating average RMSE, MAE, and MAPE for all the test sets. This is called cross validation. We will use cross validation to compare the performance of different models, i.e, with LSTM in the next steps.
- Saving the average performance metrics to a CSV file for further analysis.

4. Similarly, perform rolling forecast using LSTM and save the predictions and performance metrics to CSV files for further analysis. However, in case of LSTM, the data is scaled using MinMaxScaler and reshaped to a 3D array before training the model. Further, we use keras-tuner library to tune the hyperparameters of the LSTM model (similar to auto.arima() function) and use the best hyperparameters to train the model and predict the stock price. The hyperparameters tuned are:
   - Number of LSTM layers
   - Number of LSTM units
   - Number of epochs
   - Batch size
   - Dropout rate
5. Compare the performance of ARIMA and LSTM models using the average RMSE, MAE, and MAPE metrics.

9.1 Step 1: Importing the required libraries and setting the configuration parameters.

```
# Step 1: Import libraries
import os
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import yfinance as yf



from sklearn.metrics import mean_squared_error, mean_absolute_error, max_error, r2_score,  median_absol
ute_error, mean_absolute_percentage_error
from sklearn.preprocessing import MinMaxScaler

from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from keras.callbacks import EarlyStopping

from keras_tuner.tuners import GridSearch

from math import sqrt
import warnings
# supress warnings
warnings.filterwarnings('ignore')
# Function to check stationarity using ADF test
from matplotlib.ticker import  MaxNLocator
from pmdarima.arima import auto_arima
import time
```

```
from tabulate import tabulate


# set styles for plots
sns.set_theme(style='whitegrid', palette='muted', font_scale=1.2) # style options - white, dark, whiteg
rid, darkgrid, ticks; palette options - muted, deep, pastel, bright, dark, colorblind; font_scale optio
ns - 1.2, 1.5, 2
# changes the scale of the plot. other options include: paper, notebook, talk, poster. paper is suitabl
e for saving as pdf or for reports
sns.set_context('paper')

# Step 1A: Define variables for configuration

# the directory where the data will be saved
data_dir = 'data'
# the directory where the ARIMA results will be saved
results_dir = 'results'
# the directory where the ARIMA plots will be saved
plots_dir = 'plots'
# the list of tickers to be used for analysis
tickers = ['GOOG', 'JPM', 'JNJ', 'WMT', 'TSLA', 'AMZN', 'BP', 'NKE', 'KO', 'PFE']

# the start date of the data to be downloaded
# start_date = '2018-01-01'
start_date = '2018-01-01'
# the end date of the data to be downloaded
end_date = '2023-01-01'
 # the column name to use for analysis
column_name = 'Adj Close'
# the ratio to split the data into train and test sets
split_ratio = 0.95

# ltsm variables
 # the number of previous time steps to use as input variables to predict the next time period
look_back = 10
# number of batches to use for training at each epoch
batch_size = 1
# number of epochs to train the model
nb_epoch = 10
# maximum number of neurons to use r
neurons = 4
# petience for early stopping
early_stopping_patience = 3
# number of days to predict
days_to_predict = 1



# check if data directory exists; else create it so that data can be saved there
print('Checking if data directory exists...')
if not os.path.exists(data_dir):
    print('Data directory does not exist. Creating data directory...')
    os.makedirs(data_dir)
    print('Data directory created.')
else:
    print('Data directory exists.')

# check if  results directory exists; else create it so that results can be saved there
print('Checking if  results directory exists...')
if not os.path.exists(results_dir):
    print('Results directory does not exist. Creating  results directory...')
    os.makedirs(results_dir)
    print('Results directory created.')
else:
    print('Results directory exists.')
```

```
# check if arima plots directory exists; else create it so that plots can be saved there
print('Checking if  plots directory exists...')
if not os.path.exists(plots_dir):
    print(' plots directory does not exist. Creating  plots directory...')
    os.makedirs(plots_dir)
    print(' plots directory created.')
else:
    print(' plots directory exists.')
```

9.2 Step 2: Importing the dataset from Yahoo Finance using yfinance library and saving it to a CSV file for later use.

```
# function to get data from yfinance and save as CSV
def get_ticker_data_and_save_as_csv(ticker, start_date, end_date, data_dir):
    '''
    A function to get data from yfinance and save as CSV.

    Parameters
    ----------
    ticker : str
        The ticker symbol of the stock.
    start_date : str
        The start date of the data to be downloaded.
    end_date : str
        The end date of the data to be downloaded.
    data_dir : str
        The directory where the data will be saved.

    Returns
    -------
    data : DataFrame
        The data downloaded from yfinance.
    '''

    # Validate inputs
    if not all([ticker, start_date, end_date, data_dir]):
        raise ValueError(
            'All input parameters (ticker, start_date, end_date, data_dir) must be provided.')

    try:
        # Get data from yfinance
        print(f'Getting data for {ticker}...')
        data = yf.download(ticker, start=start_date, end=end_date)
        print('Done.')

        # Sanitize data
        print('Sanitising data...')
        data.index = pd.to_datetime(data.index)
        data = data.sort_index()

        # Check for missing values
        print('Checking for missing values...')
        if data.isnull().values.any():
            print('Data contains missing values. Using ffill method to fill missing values...')
            data = data.fillna(method='ffill')

        # Save data to data_dir
        output_file = os.path.join(data_dir, f'{ticker}.csv')
        print(f'Saving data to {output_file}...')
        data.to_csv(output_file)
```

```
        print('Done.')

    except ValueError:
        raise ValueError(f'No data found for {ticker}.')

    return data

# Function to load data from a CSV file
def load_data_from_csv(ticker, data_dir):
    '''
    A function to load data from a CSV file.

    Parameters
    ----------
    ticker : str
        The ticker symbol of the stock.
    data_dir : str
        The directory where the data is saved.

    Returns
    -------
    data : DataFrame
        The data loaded from the CSV file.
    '''

    # Validate inputs
    if not all([ticker, data_dir]):
        raise ValueError(
            'All input parameters (ticker, data_dir) must be provided.')

    # check if data directory exists; else raise error
    if not os.path.exists(data_dir):
        raise ValueError(f'Data directory {data_dir} does not exist.')
    # Load data from CSV file
    input_file = os.path.join(data_dir, f'{ticker}.csv')
    # check if file exists; else raise error
    if not os.path.exists(input_file):
        raise ValueError(f'File {input_file} does not exist.')

    print(f'Loading data from {input_file}...')
    data = pd.read_csv(input_file, index_col=0)
    print('Done.')

    return data

# We iterate through the list of tickers and get data for each ticker
for ticker in tickers:
    try:
        data_from_yahoo = get_ticker_data_and_save_as_csv(ticker,
                                                          start_date,
                                                           end_date,
                                                           data_dir)
        # Use the 'data' DataFrame as needed
        print("Obtained data for " + ticker + " from yfinance and saved to " + data_dir + ", as " + tic
ker + ".csv")
        # Handle the error accordingly
        data = load_data_from_csv(ticker, data_dir)
    except ValueError as e:
        print(f"Error occurred: {e}")
```

9.3 Step 3: Perform rolling forecast ARIMA modeling on the dataset for each stock.

```
# define functions

# Function to select a column from a dataframe
def get_column_data(df, column_name):
    '''
    A function to select a column from a DataFrame.

    Parameters
    ----------
    df : DataFrame
        The DataFrame to select the column from.
    column_name : str
        The name of the column to select.

    Returns
    -------
    values : Series
        The values of the column.
    '''

    # Validate inputs
    if df is None:
        raise ValueError('df is required.')
    if not isinstance(df, pd.DataFrame):
        raise ValueError('df should be a pandas DataFrame.')
    if not isinstance(column_name, str):
        raise ValueError('column_name should be a string.')

    # Check if the column exists in the DataFrame
    if column_name not in df.columns:
        raise ValueError(f'Column "{column_name}" does not exist in the DataFrame.')

    # Retrieve the column data
    values = df[column_name]
    return values


    # plot original data series
def plot_original_data_series(original_data_series, title_text,
                              column_name='Adj Close Price',
                                index_name='Date',
                                save_path=None,
                                file_name='original_data_series.png'):
    '''
    A function to plot the original data series.

    Parameters
    ----------
    original_data_series : Series
        The original data series.
    title_text : str
        The title of the plot.
    column_name : str

    Returns
    -------
    None.


    '''
```

```
    plt.figure(figsize=(10,6), dpi=100)
    plt.title(title_text)
    plt.xlabel(index_name)
    plt.ylabel(column_name)
    plt.plot(original_data_series, label='Original Data Series', color='blue')
    plt.xticks(rotation=45)
    # Get the current axes
    ax = plt.gca()
    # Automatically set the number of x-axis ticks
    ax.xaxis.set_major_locator(MaxNLocator(integer=True))
    #ax.xaxis.set_major_locator(AutoLocator())
    #plt.xticks(np.arange(0, len(original_data_series), len(original_data_series)/20))
    plt.grid(visible=True, linestyle='dotted', linewidth=0.5, axis='both', which='major', color='grey')
    plt.tight_layout()
    plt.legend(loc='best')
    # if save_path is provided, save the plot to the path
    if save_path is not None:
        # check file extension and add it if not present or replace it if present. accept png, jpg and
jpeg
        if not file_name.endswith('.png') and not file_name.endswith('.jpg') and not file_name.endswith
('.jpeg'):
            file_name = f'{file_name}.png'
        # save plot to the path
        plt.savefig(os.path.join(save_path, file_name))

    plt.show(block=False)
    # close plot
    plt.close()


# Function to create metrics dataframe
def create_metrics_dataframe(rows):
    # Create an empty DataFrame with columns 'key', 'text', 'value' for holding metrics
    metrics = pd.DataFrame(columns=['key', 'text', 'value'])
    # Concatenate the rows to the metrics dataframe
    for row in rows:
        # Convert the row to a DataFrame
        row_df = pd.DataFrame(row, index=[0])
        # Concatenate the row to the metrics dataframe
        metrics = pd.concat([metrics, row_df], ignore_index=True)

    return metrics



def get_predictions_and_metrics_using_arima(actual_data_indexed, train_size):
    # ###############################################################################
    # split the data into train and test sets

    train, test = actual_data_indexed[0:train_size], actual_data_indexed[train_size:len(actual_data_ind
exed)]
    train_values = train.values
    test_values = test.values
    '''
    create a history list intially containing the training data set.
    we will use this for the initial prediction
    and with each prediction, we will append the actual value to the history list and use it for the ne
xt prediction as input to the model
    '''
    history = [x for x in train_values]
    # create a list to store the test predictions
    test_predictions = list()
    # create a list to store the fitting time
    arima_model_and_fit_time_in_ms = 0
    # create a list to store the prediction time
    arima_prediction_time_in_ms = 0
```

```
    # walk-forward validation for each time step in the test data set
    i=0
    for observed_value in test_values:
        # get time used for model and fit
        print('ARIMA: Predicting for : ' + str(i) + '/' + str(len(test_values)))
        i += 1
        start_time_arima_model_and_fit = time.time()
        # define model configuration. use the history list as the input to the model
        model = auto_arima(history,
                           start_p=1, start_q=1, d=1,
                             max_p=5, max_q=2, max_d=2,
                             D=1, max_D=2, m=1,
                             seasonal=True,
                             trace=False,
                             error_action='ignore',
                             suppress_warnings=True, stepwise=True)

        # fit model
        arima_model = model.fit(history)
        end_time_arima_model_and_fit = time.time()
        arima_model_and_fit_time_in_ms += (end_time_arima_model_and_fit - start_time_arima_model_and_fi
t) * 1000
        # print the summary of the model to get the model parameters
        summary = arima_model.summary()
        print('ARIMA Model Summary:')
        print(summary)
        # get prediction for the next day
        start_time_predict = time.time()
        # Predict next value
        yhat, conf_int = arima_model.predict(n_periods=1, return_conf_int=True)
        end_time_predict = time.time()
        arima_prediction_time_in_ms += (end_time_predict - start_time_predict) * 1000 # convert to ms
        # store the prediction
        test_predictions.append(yhat)
        # add the actual value to the history object for the next iteration to train the model
        history.append(observed_value)
    # get time used for fit and predict
    arima_total_time_for_model_fit_and_predict_in_ms = arima_model_and_fit_time_in_ms + arima_predictio
n_time_in_ms
    # recovery is not needed since we are using auto_arima and provided the original series as input to
the model
     # Convert test_predictions list into a Pandas Series
    flattened_test_predictions = [item for sublist in test_predictions for item in sublist]
    test_predictions_series = pd.Series(flattened_test_predictions, index=test.index)
    # calulate accuracy metrics
    rmse = sqrt(mean_squared_error(test_values, test_predictions))
    mae = mean_absolute_error(test_values, test_predictions)
    mape = mean_absolute_percentage_error(test_values, test_predictions) * 100
    max_error_value = max_error(test_values, test_predictions)
    r2 = r2_score(test_values, test_predictions)
    median_absolute_error_value = median_absolute_error(test_values, test_predictions)

    rows = [
        {'key': 'rmse', 'text': 'RMSE', 'value': rmse},
        {'key': 'mape', 'text': 'MAPE', 'value': mape},
        {'key': 'r2', 'text': 'R2', 'value': r2},
        {'key': 'max_error_value', 'text': 'Max Error', 'value': max_error_value},
        {'key': 'mean_absolute_error', 'text': 'Mean Absolute Error', 'value': mae},
        {'key': 'median_absolute_error_value', 'text': 'Median Absolute Error', 'value': median_absolut
e_error_value},
        {'key': 'arima_total_time_for_model_fit_and_predict_in_ms', 'text': 'Total Time for Fit and Pre
dict', 'value': arima_total_time_for_model_fit_and_predict_in_ms},
        {'key': 'arima_model_and_fit_time_in_ms', 'text': 'Total Time for Model and Fit', 'value': arim
a_model_and_fit_time_in_ms},
        {'key': 'arima_prediction_time_in_ms', 'text': 'Total Time for Prediction', 'value': arima_pred
iction_time_in_ms}
    ]
```

```
    # create a dataframe with columns - key, text, value for holding metrics
    metrics_df = create_metrics_dataframe(rows)
    # return the model, test_predictions_series and metrics_df
    return arima_model, test_predictions_series, metrics_df


# Function to plot the actual and predicted values
def plot_actual_and_predicted_values(actual_values,
                                     predicted_values,
                                     title_text,
                                     index_name='Date',
                                     column_name='Price',
                                     save_path=None,
                                     file_name='actual_vs_predicted.png'):
    '''
    A function to plot the actual and predicted values.

    Parameters
    ----------

    '''
     # plot orginal_data_series_test and test_predictions_series
    plt.figure(figsize=(10,6), dpi=100)
    plt.title(title_text)
    plt.xlabel(index_name)
    plt.ylabel(column_name)
    plt.plot(actual_values, label='Actual', color='blue')
    plt.plot(predicted_values, label='Predictions', color='orange')
    plt.xticks(rotation=45)
    # Get the current axes
    ax = plt.gca()
    # Automatically set the number of x-axis ticks
    ax.xaxis.set_major_locator(MaxNLocator(integer=True))
    # show legend
    plt.legend(loc='best')

    # if save_path is provided, save the plot to the path
    if save_path is not None:
        # check file extension and add it if not present or replace it if present. accept png, jpg and
jpeg
        if not file_name.endswith('.png') and not file_name.endswith('.jpg') and not file_name.endswith
('.jpeg'):
            file_name = f'{file_name}.png'
        # save plot to the path
        plt.savefig(os.path.join(save_path, file_name))

    #show plot
    plt.show()
    # close plot
    plt.close()


# create a map to hold the metrics for each ticker
arima_metrics_map = {}

for ticker in tickers:
  try:
    data = load_data_from_csv(ticker, data_dir)
    # Use the 'data' DataFrame as needed
    print("Loaded data for " + ticker + " from " + data_dir + ", as " + ticker + ".csv")
    adj_close_data = get_column_data(data, column_name)
    # lets call the adj_close_data as ''original_data_series' so that there is no confusion
    original_data_series = adj_close_data
    # plot the original data series for visual inspection
    plot_original_data_series(original_data_series,
                              f'{ticker}: {column_name} Prices: (Original Data Series)',
                              'Original Data Series',
```

```
                              'Date',
                              plots_dir,
                              f'{ticker}_original_data_series.png')
    # split the data into train and test data
    train_size = int(len(original_data_series) * split_ratio)
    original_data_series_train = original_data_series[:train_size]
    original_data_series_test = original_data_series[train_size:]
    # use get_predictions_and_metrics_using_arima
    original_data_arima_model, original_data_test_predictions_series, arima_metrics_df = get_prediction
s_and_metrics_using_arima(
        original_data_series,
        int(len(original_data_series) * split_ratio)
        )
    # add the metrics dataframe to the map
    arima_metrics_map[ticker] = arima_metrics_df
    # save the metrics dataframe to a csv file
    arima_metrics_df.to_csv(os.path.join(results_dir, f'{ticker}_arima_metrics.csv'))
    # save test_predictions_series to a csv file
    original_data_test_predictions_series.to_csv(os.path.join(results_dir,
                                                              f'{ticker}_arima_original_data_test_pred
ictions_series.csv'))

    # plot orginal_data_series_test and test_predictions_series
    plt.figure(figsize=(10,6), dpi=100)
    plt.title(f'{ticker}: {column_name} Prices: ARIMA - Observed Vs Predictions):ARIMA')
    plt.xlabel('Date')
    plt.ylabel(column_name)
    plt.plot(original_data_series, label='Observed(Actual)', color='blue')
    plt.plot(original_data_test_predictions_series, label='Predictions', color='orange')
    plt.xticks(rotation=45)
    # Get the current axes
    ax = plt.gca()
    # Automatically set the number of x-axis ticks
    ax.xaxis.set_major_locator(MaxNLocator(integer=True))
    # show legend
    plt.legend(loc='best')
    #show plot
    plt.show()


    # plot orginal_data_series_test and test_predictions_series
    plot_actual_and_predicted_values(original_data_series_test,
                                     original_data_test_predictions_series,
                                       f'{ticker}: {column_name} Prices: ARIMA - Observed Vs Prediction
s',
                                         index_name='Date',
                                         column_name=column_name,
                                         save_path=plots_dir,
                                         file_name=f'{ticker}_arima_original_data_test_predictions_seri
es.png')



  except ValueError as e:
    print(f"Error occurred: {e}")


# select perfomance_metrics and time_metrics from arima_metrics_map  and add to arima_select_performanc
e_metrics and arima_time_metrics maps
arima_select_performance_metrics = {}
arima_time_metrics = {}
arima_mean_performance_metrics = {}
arima_mean_time_metrics = {}
selected_performance_metrics = ['rmse', 'mape', 'mean_absolute_error']
selected_time_metrics = ['arima_model_and_fit_time_in_ms',
                         'arima_prediction_time_in_ms',
                         'arima_total_time_for_model_fit_and_predict_in_ms']
```

```
def calculate_mean_metrics(metrics_dict, tickers, selected_metrics):
    mean_metrics = {}
    for metric in selected_metrics:
        mean_metrics[metric] = sum(metrics_dict[ticker][metrics_dict[ticker]['key']
                                                       == metric]['value'].values[0] for ticker in tic
kers) / len(tickers)
    return mean_metrics

for ticker in tickers:
    # add the selected performance metrics to arima_select_performance_metrics
    arima_select_performance_metrics[ticker] = arima_metrics_map[ticker][arima_metrics_map[ticker]['key
'].isin(selected_performance_metrics)]
    # add the selected time metrics to arima_time_metrics
    arima_time_metrics[ticker] = arima_metrics_map[ticker][arima_metrics_map[ticker]['key'].isin(select
ed_time_metrics)]

# Calculate mean performance metrics
arima_mean_performance_metrics = calculate_mean_metrics(arima_select_performance_metrics,
                                                        tickers,
                                                        selected_performance_metrics)

# Calculate mean time metrics
arima_mean_time_metrics = calculate_mean_metrics(arima_time_metrics, tickers, selected_time_metrics)

# pring arima_select_performance_metrics and arima_time_metrics as tables using tabulate with columns -
Ticker, RMSE, MAPE, Mean Absolute Error, and title as 'Performance Metrics for ARIMA'
print('\nAccuracy Metrics for ARIMA:\n')

all_data = []
for ticker in tickers:
    ticker_data = arima_select_performance_metrics.get(ticker)
    if ticker_data is not None and not ticker_data.empty:
        all_data.append([ticker, 'RMSE', ticker_data[ticker_data['key'] == 'rmse']['value'].values[0]])
        all_data.append(['', 'MAPE', ticker_data[ticker_data['key'] == 'mape']['value'].values[0]])
        all_data.append(['','Mean Absolute Error', ticker_data[ticker_data['key'] == 'mean_absolute_err
or']['value'].values[0]])
    else:
        print(f'No data for {ticker} in arima_select_performance_metrics\n')

headers = ['Ticker', 'Metric', 'Value']
merged_table = pd.DataFrame(all_data, columns=headers)
print(tabulate(merged_table, headers=headers, tablefmt='orgtbl', showindex=False, floatfmt=".4f", numal
ign="right"))
print('\n')


'''
print arima_mean_performance_metrics and arima_mean_time_metrics as a table
with columns - RMSE, MAPE, Mean Absolute Error, and title as 'Average Performance Metrics for ARIMA
'''
print('\nAverage Accuracy Metrics for ARIMA:\n')
mean_rsme_arima = arima_mean_performance_metrics['rmse']
mean_mape_arima = arima_mean_performance_metrics['mape']
mean_mean_absolute_error_arima = arima_mean_performance_metrics['mean_absolute_error']

table_data = [['RMSE', mean_rsme_arima], ['MAPE', mean_mape_arima], ['Mean Absolute Error', mean_mean_a
bsolute_error_arima]]
headers = ['Metric', 'Value']
print(tabulate(table_data, headers=headers, tablefmt='orgtbl'))
print('\n')


# pring arima_time_metrics as tables using tabulate with columns - Ticker, Model Fit Time(ARIMA), Predi
ction Time(ARIMA), Total Time for Fit and Predict(ARIMA), and title as 'Time Metrics for ARIMA'
print('\nTime Metrics for ARIMA:\n')
all_data = []
```

```
for ticker in tickers:

    model_fit_time_arima = arima_time_metrics[ticker][arima_time_metrics[ticker]['key']
                                                      == 'arima_model_and_fit_time_in_ms']['value'].val
ues[0]
    prediction_time_arima = arima_time_metrics[ticker][arima_time_metrics[ticker]['key']
                                                       == 'arima_prediction_time_in_ms']['value'].value
s[0]
    total_time_for_model_fit_and_predict_arima = arima_time_metrics[ticker][arima_time_metrics[ticker][
'key']
                                                                                == 'arima_total_time_for_mo
del_fit_and_predict_in_ms']['value'].values[0]

    all_data.append([ticker, 'Model Fit Time(ARIMA)', model_fit_time_arima])
    all_data.append(['', 'Prediction Time(ARIMA)', prediction_time_arima])
    all_data.append(['', 'Total Time for Fit and Predict(ARIMA)', total_time_for_model_fit_and_predict_
arima])

headers = ['Ticker', 'Metric', 'Value']
merged_table = pd.DataFrame(all_data, columns=headers)
'''
print(merged_table) using tabulate
with columns - Ticker, Model Fit Time(ARIMA), Prediction Time(ARIMA),
Total Time for Fit and Predict(ARIMA),
and title as 'Time Metrics for ARIMA
'''
print(tabulate(merged_table,
                headers=headers,
                tablefmt='orgtbl',
                  showindex=False,
                  floatfmt=".4f",
                  numalign="right"))
print('\n')

# print arima_mean_time_metrics as a table using tabulate with columns - Model Fit Time(ARIMA), Predict
ion Time(ARIMA),
# Total Time for Fit and Predict(ARIMA), and title as 'Average Time Metrics for ARIMA'
print('\nAverage Time Metrics for ARIMA:\n')
mean_model_fit_time_arima = arima_mean_time_metrics['arima_model_and_fit_time_in_ms']
mean_prediction_time_arima = arima_mean_time_metrics['arima_prediction_time_in_ms']
mean_total_time_for_model_fit_and_predict_arima = arima_mean_time_metrics['arima_total_time_for_model_f
it_and_predict_in_ms']

table_data = [['Model Fit Time(ARIMA)',
                mean_model_fit_time_arima],
                  ['Prediction Time(ARIMA)',
                    mean_prediction_time_arima],
                    ['Total Time for Fit and Predict(ARIMA)',
                      mean_total_time_for_model_fit_and_predict_arima]]
headers = ['Metric', 'Value']
# print the table using tabulate
print(tabulate(table_data,
               headers=headers,
               tablefmt='orgtbl',
               showindex=False,
               floatfmt=".4f",
               numalign="right"))
print('\n')
```

9.4 Step 4: Perform rolling forecast using LSTM and save the predictions and performance metrics to CSV files for further analysis.

```
# convert an array of values into a dataset matrix
def create_dataset(dataset, look_back=1):
  # dataset is a numpy array that contains the stock prices
  # look_back is the number of previous time steps to use as input variables to predict the next time p
eriod
     # dataX is the input variable while dataY is the output variable;
    dataX, dataY = [], []
  # dataX contains the previous 20 days of stock prices while dataY contains the stock prices for the n
ext day
          # if dataset is 100, look_back is 20, then the loop will run from 0 to 79
    for i in range(len(dataset)-look_back-1):
          # a will contain the stock prices from 0 to 19 in the first iteration, 1 to 20 in the second
iteration and so on
        a = dataset[i:(i+look_back), 0]
          # append the 20 stock prices to dataX at each iteration;
          #so dataX will contain 80 arrays of 20 stock prices each increasing by 1 stock price at each
iteration
        dataX.append(a)
          # append the stock price for the 21st day to dataY at each iteration;
          #so dataY will contain 80 stock prices increasing by 1 stock price at each iteration
        dataY.append(dataset[i + look_back, 0])
# return dataX and dataY as numpy arrays
    return np.array(dataX), np.array(dataY)


# Function to build lstm model
def build_lstm_model(hp):
    model = Sequential()
    model.add(LSTM(units=hp.Int('units', min_value=1, max_value=50, step=1),
                   batch_input_shape=(batch_size, look_back, 1),
                   stateful=True))
    model.add(Dense(1))
    model.compile(loss='mean_squared_error', optimizer='adam')
    return model


# get predictions and metrics using lstm with rolling window
def get_predictions_and_metrics_using_lstm(actual_data_indexed, train_size):
    org_data_set = actual_data_indexed
    data_series = org_data_set.values
    # normalize the dataset
    scaler = MinMaxScaler(feature_range=(0, 1))
    data_series = scaler.fit_transform(data_series.reshape(-1, 1))
    # split into train and test sets
    train_size = int(len(data_series) * split_ratio)
    test_size = len(data_series) - train_size
    train, test = data_series[0:train_size,:], data_series[train_size:len(data_series),:]
    train_indexed = org_data_set[0:train_size]
    test_indexed = org_data_set[train_size:len(org_data_set)]
    # reshape into X=t and Y=t+1
    trainX, trainY = create_dataset(train, look_back)
    testX, testY = create_dataset(test, look_back)
    # reshape input to be [samples, time steps, features]

     # trainX.shape[0] is the number of rows in trainX, trainX.shape[1] is the number of columns in tra
inX, 1 is the number of features
    trainX = np.reshape(trainX, (trainX.shape[0], trainX.shape[1], 1))
    # testX.shape[0] is the number of rows in testX, testX.shape[1] is the number of columns in testX,
1 is the number of features
    testX = np.reshape(testX, (testX.shape[0], testX.shape[1], 1))
```

```
    # create a stateful LSTM network
    lstm_model_and_fit_time_in_ms = 0
    tuner = GridSearch(
        build_lstm_model,
        objective='val_loss',
        max_trials=5) # Set the total number of trials
    print('finding LSTM best model...')


    # start lstm model and fit time
    start_lstm_model_and_fit_time = time.time()
    # search for best model
    tuner.search(trainX,
                 trainY,
                 epochs=5,
                 batch_size=batch_size,
                 validation_data=(testX, testY)
                 )
    # get the best model
    best_model = tuner.get_best_models(1)[0]
    print(best_model.summary())
    # define early stopping callback
    '''
    Here, monitor is the quantity to be monitored,
      patience is the number of epochs with no improvement after which training will be stopped,
        verbose is the verbosity mode,
        verbose=0 is silent, verbose=1 is progress bar,
        verbose=2 is one line per epoch
    '''
    early_stopping = EarlyStopping(monitor='loss',
                                   patience=early_stopping_patience,
                                   verbose=1)
    # define the LSTM layer
    best_model.fit(trainX, trainY,
                   epochs=nb_epoch,
                   batch_size=batch_size,
                   verbose=2,
                   shuffle=False,
                   callbacks=[early_stopping])
    # end lstm model and fit time
    end_lstm_model_and_fit_time = time.time()
    # get time used for fit and predict
    lstm_model_and_fit_time_in_ms += (end_lstm_model_and_fit_time - start_lstm_model_and_fit_time) * 10
00
    # implement walk forward validation and get predictions
    lstm_predictions = list()
    lstm_prediction_time_in_ms = 0
    # walk-forward validation for each time step in the test data set
    i=0
    for obs in test:
        # X, y = testX[i, 0, :], testY[i]
        # reshape input to be [samples, time steps, features]
        i = i + 1
        print(" LSTM: Predicting for " + str(i) + "/" + str(len(test)))
        X = trainX[-1, :, :]
        # X = X.reshape(1, 1, len(X))
        X = X.reshape(1, look_back, 1)
        start_time_lstm_prediction = time.time()
        yhat = best_model.predict(X, batch_size=1)
        end_time_lstm_prediction = time.time()
        lstm_prediction_time_in_ms += (end_time_lstm_prediction
                                       - start_time_lstm_prediction) * 1000

        # update train with the actual value
        updated_train = data_series[0:train_size+i,:]
        # remove the first row from train so that the number of rows in train remains the same
        train = updated_train[1:]
```

```
        # recreate trainX and trainY
        trainX, trainY = create_dataset(train, look_back)

        '''
        reshape into X=t and Y=t+1.
        we need it in format [samples, time steps, features]
        trainX is the input variable while trainY is the output variable
        trainX.shape[0] is the number of rows in trainX,
        trainX.shape[1] is the number of columns in trainX, 1 is the number of features
        '''

        trainX = np.reshape(trainX,
                            (trainX.shape[0],
                             trainX.shape[1],
                             1))
        '''
        reshape into X=t and Y=t+1.
        we need it in format [samples, time steps, features]
        trainY is the output variable
        trainY.shape[0] is the number of rows in trainY,
        1 is the number of columns in trainY, i.e. the number of features
        '''
        trainY = np.reshape(trainY, (trainY.shape[0], 1))


        # invert scaling to get the actual value
        yhat = scaler.inverse_transform(yhat)
        # add to predictions
        lstm_predictions.append(yhat[0,0])

        # Update the model state for the next iteration
        best_model.reset_states()

        # create a stateful LSTM network
        print('finding LSTM best model...')
        # start lstm model and fit time
        start_lstm_model_and_fit_time = time.time()
        # return best model with updated parameters.
        # This possibly makes the model better than the previous one.
        # But, not necessarily better than the best model overall.
        tuner.search(trainX,
                     trainY,
                     epochs=5,
                     batch_size=batch_size,
                     validation_data=(testX, testY)
                     )
        # get the best model
        best_model = tuner.get_best_models(1)[0]
        print(best_model.summary())
        # fit the best model
        best_model.fit(trainX,
                       trainY,
                         epochs=nb_epoch,
                         batch_size=batch_size,
                         verbose=2,
                         shuffle=False,
                         callbacks=[early_stopping]
                         )
        # end lstm model and fit time
        end_lstm_model_and_fit_time = time.time()
        # get time used for fit and predict
        lstm_model_and_fit_time_in_ms += (end_lstm_model_and_fit_time - start_lstm_model_and_fit_time)
* 1000

    # invert scaling for actual
    actual_vaules = test_indexed.values
```

```
    # calculate metric
    lstm_rmse = sqrt(mean_squared_error(actual_vaules, lstm_predictions))
    lstm_mae = mean_absolute_error(actual_vaules, lstm_predictions)
    lstm_mape = mean_absolute_percentage_error(actual_vaules, lstm_predictions) * 100
    lstm_max_error_value = max_error(actual_vaules, lstm_predictions)
    lstm_r2 = r2_score(actual_vaules, lstm_predictions)
    lstm_median_absolute_error_value = median_absolute_error(actual_vaules, lstm_predictions)
    # get time used for fit and predict
    lstm_total_time_for_model_fit_and_predict_in_ms = lstm_model_and_fit_time_in_ms + lstm_prediction_t
ime_in_ms
    rows = [
        {'key': 'rmse', 'text': 'RMSE', 'value': lstm_rmse},
        {'key': 'mape', 'text': 'MAPE', 'value': lstm_mape},
        {'key': 'r2', 'text': 'R2', 'value': lstm_r2},
        {'key': 'max_error_value', 'text': 'Max Error', 'value': lstm_max_error_value},
        {'key': 'mean_absolute_error', 'text': 'Mean Absolute Error', 'value': lstm_mae},
        {'key': 'median_absolute_error_value', 'text': 'Median Absolute Error', 'value': lstm_median_ab
solute_error_value},
        {'key': 'lstm_total_time_for_model_fit_and_predict_in_ms', 'text': 'Total Time for Fit and Pred
ict', 'value': lstm_total_time_for_model_fit_and_predict_in_ms},
        {'key': 'lstm_model_and_fit_time_in_ms', 'text': 'Total Time for Model and Fit', 'value': lstm_
model_and_fit_time_in_ms},
        {'key': 'lstm_prediction_time_in_ms', 'text': 'Total Time for Prediction', 'value': lstm_predic
tion_time_in_ms}
    ]

    lstm_metrics_df = create_metrics_dataframe(rows)
        # Convert test_predictions list into a Pandas Series
    lstm_test_predictions_series = pd.Series(lstm_predictions, index=test_indexed.index)
    # return the model, test_predictions_series and metrics_df
    return lstm_test_predictions_series, lstm_metrics_df




# crate a table with columns - ticker, RMSE, MAPE, Mean Absolute Error, and for each ticker in tickers
list, add the ticker and the corresponding values for RMSE, MAPE, Mean Absolute Error
lstm_metrics_map = {}


for ticker in tickers:
    data = load_data_from_csv(ticker, data_dir)
    # Use the 'data' DataFrame as needed
    print("Loaded data for " + ticker + " from " + data_dir + ", as " + ticker + ".csv")
    adj_close_data = get_column_data(data, column_name)

    # lets call the adj_close_data as ''original_data_series' so that there is no confusion
    org_data_set = adj_close_data
    test_data_set = org_data_set[int(len(org_data_set) * split_ratio):]
    # get metrics and predictions using lstm with rolling window
    lstm_test_predictions_series, lstm_metrics_df = get_predictions_and_metrics_using_lstm(org_data_set
,
                                                                                           int(len(org_
data_set) * split_ratio))

    #save the metrics dataframe to a csv file
    lstm_metrics_df.to_csv(os.path.join(results_dir,
                                        f'{ticker}_lstm_metrics.csv'))
    # save test_predictions_series to a csv file
    lstm_test_predictions_series.to_csv(os.path.join(results_dir,
                                                     f'{ticker}_lstm_original_data_test_predictions_ser
ies.csv'))

    # add the metrics dataframe to the map
    lstm_metrics_map[ticker] = lstm_metrics_df
    # save the metrics dataframe to a csv file
```

```
    # convert test_data_set to float 63 as the test_data_set is of type float64 as test_data_set is of
type float64
    test_data_set = test_data_set.astype('float64')
    lstm_test_predictions_series = lstm_test_predictions_series.astype('float64')
    # reindex test_data_set and lstm_test_predictions_series
    test_data_set = test_data_set.reindex(lstm_test_predictions_series.index)

    # plot the actual and predicted values
    plot_actual_and_predicted_values(test_data_set,
                                     lstm_test_predictions_series,
                                     f'{ticker}: {column_name} : (Actual Vs Predictions):LSTM',
                                     index_name='Date',
                                     column_name=column_name,
                                     save_path=plots_dir,
                                     file_name=f'{ticker}_lstm_original_data_test_predictions_series.pn
g')

# select perfomance_metrics and time_metrics from lstm_metrics_map  and add to lstm_select_performance_
metrics and lstm_time_metrics maps
lstm_select_performance_metrics = {}
lstm_time_metrics = {}
lstm_mean_performance_metrics = {}
lstm_mean_time_metrics = {}
selected_performance_metrics = ['rmse', 'mape', 'mean_absolute_error']
selected_time_metrics = ['lstm_model_and_fit_time_in_ms', 'lstm_prediction_time_in_ms', 'lstm_total_tim
e_for_model_fit_and_predict_in_ms']

for ticker in tickers:
    lstm_select_performance_metrics[ticker] = lstm_metrics_map[ticker][lstm_metrics_map[ticker]['key'].
isin(selected_performance_metrics)]
    lstm_time_metrics[ticker] = lstm_metrics_map[ticker][lstm_metrics_map[ticker]['key'].isin(selected_
time_metrics)]

# Calculate mean performance metrics
lstm_mean_performance_metrics = calculate_mean_metrics(lstm_select_performance_metrics,
                                                       tickers,
                                                       selected_performance_metrics)

# Calculate mean time metrics
lstm_mean_time_metrics = calculate_mean_metrics(lstm_time_metrics,
                                                tickers,
                                                selected_time_metrics)

# pring lstm_select_performance_metrics and lstm_time_metrics as tables using tabulate with columns - T
icker, RMSE, MAPE, Mean Absolute Error, and title as 'Performance Metrics for LSTM'
print('\Accuracy Metrics for LSTM:\n')
all_data = []
for ticker in tickers:
    rmse_lstm = lstm_select_performance_metrics[ticker][lstm_select_performance_metrics[ticker]['key']
                                                        == 'rmse']['value'].values[0]
    mape_lstm = lstm_select_performance_metrics[ticker][lstm_select_performance_metrics[ticker]['key']
                                                        == 'mape']['value'].values[0]
    mean_absolute_error_lstm = lstm_select_performance_metrics[ticker][lstm_select_performance_metrics[
ticker]['key']
                                                                       == 'mean_absolute_error']['value
'].values[0]

    all_data.append([ticker, 'RMSE', rmse_lstm])
    all_data.append(['', 'MAPE', mape_lstm])
    all_data.append(['','Mean Absolute Error', mean_absolute_error_lstm])

headers = ['Ticker', 'Metric', 'Value']
merged_table = pd.DataFrame(all_data, columns=headers)
print(tabulate(merged_table, headers=headers,
               tablefmt='orgtbl',
               showindex=False,
               floatfmt=".4f",
```

```
                numalign="right"))
print('\n')

'''
# print lstm_mean_performance_metrics and lstm_mean_time_metrics as a table
using tabulate with columns - RMSE, MAPE, Mean Absolute Error,
and title as 'Average Performance Metrics for LSTM
'''
print('\nAverage Accuracy Metrics for LSTM:\n')
mean_rsme_lstm = lstm_mean_performance_metrics['rmse']
mean_mape_lstm = lstm_mean_performance_metrics['mape']
mean_mean_absolute_error_lstm = lstm_mean_performance_metrics['mean_absolute_error']

table_data = [['RMSE', mean_rsme_lstm],
              ['MAPE', mean_mape_lstm],
              ['Mean Absolute Error', mean_mean_absolute_error_lstm]]
headers = ['Metric', 'Value']

print(tabulate(table_data,
               headers=headers,
               tablefmt='orgtbl'))
print('\n')

''''
print lstm_time_metrics as tables using tabulate with columns - Ticker, Model Fit Time(LSTM), Predictio
n Time(LSTM),
Total Time for Fit and Predict(LSTM), and title as 'Time Metrics for LSTM

'''

print('\nTime Metrics for LSTM:\n')
all_data = []
for ticker in tickers:
    model_fit_time_lstm = lstm_time_metrics[ticker][lstm_time_metrics[ticker]['key']
                                                    == 'lstm_model_and_fit_time_in_ms']['value'].values
[0]
    prediction_time_lstm = lstm_time_metrics[ticker][lstm_time_metrics[ticker]['key']
                                                     == 'lstm_prediction_time_in_ms']['value'].values[0
]
    total_time_for_model_fit_and_predict_lstm = lstm_time_metrics[ticker][lstm_time_metrics[ticker]['ke
y']
                                                                          == 'lstm_total_time_for_model
_fit_and_predict_in_ms']['value'].values[0]
    all_data.append([ticker, 'Model Fit Time(LSTM)', model_fit_time_lstm])
    all_data.append(['', 'Prediction Time(LSTM)', prediction_time_lstm])
    all_data.append(['', 'Total Time for Fit and Predict(LSTM)', total_time_for_model_fit_and_predict_l
stm])

headers = ['Ticker', 'Metric', 'Value']
merged_table = pd.DataFrame(all_data, columns=headers)
print(tabulate(merged_table, headers=headers, tablefmt='orgtbl', showindex=False, floatfmt=".4f", numal
ign="right"))
print('\n')

'''
print lstm_mean_time_metrics as a table using tabulate
with columns - Model Fit Time(LSTM), Prediction Time(LSTM), Total Time for Fit and Predict(LSTM),
and title as 'Average Time Metrics for LSTM'
'''

print('\nAverage Time Metrics for LSTM:\n')
mean_model_fit_time_lstm = lstm_mean_time_metrics['lstm_model_and_fit_time_in_ms']
mean_prediction_time_lstm = lstm_mean_time_metrics['lstm_prediction_time_in_ms']
mean_total_time_for_model_fit_and_predict_lstm = lstm_mean_time_metrics['lstm_total_time_for_model_fit_
and_predict_in_ms']

table_data = [['Model Fit Time(LSTM)',
```

```
               mean_model_fit_time_lstm],
               ['Prediction Time(LSTM)',
                mean_prediction_time_lstm],
                ['Total Time for Fit and Predict(LSTM)',
                 mean_total_time_for_model_fit_and_predict_lstm]]
headers = ['Metric', 'Value']
print(tabulate(table_data, headers=headers, tablefmt='orgtbl'))
print('\n')
```

9.5 Step 5: Compare the performance of ARIMA and LSTM models using the average RMSE, MAE, and MAPE metrics.

```
'''
print comparison of performance metrics for ARIMA and LSTM as a table
using tabulate
with columns - Metric, ARIMA, LSTM,
and title as 'Comparison of Performance Metrics for ARIMA and LSTM', row names the ticker symbols
'''

print('\nComparison of Accuracy Metrics for ARIMA and LSTM:\n')
all_data = []
for ticker in tickers:
   rmse_arima = arima_select_performance_metrics[ticker][arima_select_performance_metrics[ticker]['key'
]
                                                         == 'rmse']['value'].values[0]
   mape_arima = arima_select_performance_metrics[ticker][arima_select_performance_metrics[ticker]['key'
]
                                                         == 'mape']['value'].values[0]
   mean_absolute_error_arima = arima_select_performance_metrics[ticker][arima_select_performance_metric
s[ticker]['key']
                                                                      == 'mean_absolute_error']['valu
e'].values[0]
   rmse_lstm = lstm_select_performance_metrics[ticker][lstm_select_performance_metrics[ticker]['key']
                                                       == 'rmse']['value'].values[0]
   mape_lstm = lstm_select_performance_metrics[ticker][lstm_select_performance_metrics[ticker]['key']
                                                       == 'mape']['value'].values[0]
   mean_absolute_error_lstm = lstm_select_performance_metrics[ticker][lstm_select_performance_metrics[t
icker]['key'] == 'mean_absolute_error']['value'].values[0]

   all_data.append([ticker, 'RMSE', rmse_arima, rmse_lstm])
   all_data.append(['', 'MAPE', mape_arima, mape_lstm])
   all_data.append(['','Mean Absolute Error', mean_absolute_error_arima, mean_absolute_error_lstm])

headers = ['Ticker', 'Metric', 'ARIMA', 'LSTM']
merged_table = pd.DataFrame(all_data, columns=headers)
print(tabulate(merged_table, headers=headers, tablefmt='orgtbl', showindex=False, floatfmt=".4f", numal
ign="right"))
print('\n')

   # print comparison of time metrics for ARIMA and LSTM as a table
print('\nComparison of Time Metrics for ARIMA and LSTM:\n')
all_data = []
for ticker in tickers:
   arima_model_and_fit_time_in_ms = arima_time_metrics[ticker][arima_time_metrics[ticker]['key']
                                                              == 'arima_model_and_fit_time_in_ms']['va
lue'].values[0]
   arima_prediction_time_in_ms = arima_time_metrics[ticker][arima_time_metrics[ticker]['key']
                                                            == 'arima_prediction_time_in_ms']['value'].
values[0]
   arima_total_time_for_model_fit_and_predict_in_ms = arima_time_metrics[ticker][arima_time_metrics[tic
ker]['key']
                                                                         == 'arima_total_time_f
or_model_fit_and_predict_in_ms']['value'].values[0]
```

```
   lstm_model_and_fit_time_in_ms = lstm_time_metrics[ticker][lstm_time_metrics[ticker]['key']
                                                            == 'lstm_model_and_fit_time_in_ms']['value
'].values[0]
   lstm_prediction_time_in_ms = lstm_time_metrics[ticker][lstm_time_metrics[ticker]['key']
                                                        == 'lstm_prediction_time_in_ms']['value'].val
ues[0]
   lstm_total_time_for_model_fit_and_predict_in_ms = lstm_time_metrics[ticker][lstm_time_metrics[ticker
]['key']
                                                                            == 'lstm_total_time_for_
model_fit_and_predict_in_ms']['value'].values[0]

   all_data.append([ticker, 'Model Fit Time',
                    arima_model_and_fit_time_in_ms, lstm_model_and_fit_time_in_ms])
   all_data.append(["", 'Prediction Time',
                    arima_prediction_time_in_ms, lstm_prediction_time_in_ms])
   all_data.append(['','Total Time for Fit and Predict',
                    arima_total_time_for_model_fit_and_predict_in_ms, lstm_total_time_for_model_fit_and
_predict_in_ms])


headers = ['Ticker', 'Metric', 'ARIMA', 'LSTM']
merged_table = pd.DataFrame(all_data, columns=headers)
print(tabulate(merged_table, headers=headers, tablefmt='orgtbl', showindex=False, floatfmt=".4f", numal
ign="right"))
print('\n')

# print comparison of average performance metrics for ARIMA and LSTM as a table
print('\nComparison of Average Accuracy Metrics for ARIMA and LSTM:\n')
rmse_arima = arima_mean_performance_metrics['rmse']
mape_arima = arima_mean_performance_metrics['mape']
mean_absolute_error_arima = arima_mean_performance_metrics['mean_absolute_error']
rmse_lstm = lstm_mean_performance_metrics['rmse']
mape_lstm = lstm_mean_performance_metrics['mape']
mean_absolute_error_lstm = lstm_mean_performance_metrics['mean_absolute_error']

table_data = [
        ['Mean RMSE', rmse_arima, rmse_lstm],
        ['Mean MAPE', mape_arima, mape_lstm],
        ['Mean Mean Absolute Error', mean_absolute_error_arima, mean_absolute_error_lstm]
    ]

headers = ['Metric', 'ARIMA', 'LSTM']
print(tabulate(table_data, headers=headers, tablefmt='orgtbl', colalign=['center', 'right', 'right']))
print('\n')

# print comparison of average time metrics for ARIMA and LSTM as a table
print('\nComparison of Average Time Metrics for ARIMA and LSTM:\n')
arima_model_and_fit_time_in_ms = arima_mean_time_metrics['arima_model_and_fit_time_in_ms']
arima_prediction_time_in_ms = arima_mean_time_metrics['arima_prediction_time_in_ms']
arima_total_time_for_model_fit_and_predict_in_ms = arima_mean_time_metrics['arima_total_time_for_model_
fit_and_predict_in_ms']
lstm_model_and_fit_time_in_ms = lstm_mean_time_metrics['lstm_model_and_fit_time_in_ms']
lstm_prediction_time_in_ms = lstm_mean_time_metrics['lstm_prediction_time_in_ms']
lstm_total_time_for_model_fit_and_predict_in_ms = lstm_mean_time_metrics['lstm_total_time_for_model_fit
_and_predict_in_ms']

table_data = [
        ['Mean Model Fit Time', arima_model_and_fit_time_in_ms, lstm_model_and_fit_time_in_ms],
        ['Mean Prediction Time', arima_prediction_time_in_ms, lstm_prediction_time_in_ms],
        ['Mean Total Time for Fit and Predict', arima_total_time_for_model_fit_and_predict_in_ms, lstm_
total_time_for_model_fit_and_predict_in_ms]
    ]

headers = ['Metric', 'ARIMA', 'LSTM']
print(tabulate(table_data, headers=headers, tablefmt='orgtbl', colalign=['center', 'right', 'right']))
print('\n')
```

9.6 Code Acknowledgements

The code in this appendix was developed with insights from several open-source contributors, online resources, forums, and code snippets. These sources collectively contributed to the understanding and implementation of specific aspects of the code. Notably, the parameter tuning process in ARIMA, using the *auto_arima* function provided by the *pmdarima* library, drew inspiration from the works of Sumi, Nissa et al., and Brownlee. Additionally, guidance on ARIMA model creation and estimation, including the rolling step, was adapted from Brownlee's blog post. The implementation of multi-step forecasts with LSTM followed the approach presented by Mingboi, while hyperparameter tuning in LSTM was influenced by Rendyk's method and the insights shared by Quant (Ai).