Stock Price Prediction using ARIMA, Prophet, and LSTM

Vignesh Murugan

November 23, 2024

Abstract

This paper presents a comparative analysis of three models—ARIMA, Prophet, and LSTM—for stock price prediction. The objective is to evaluate their performance in forecasting stock prices and highlight their strengths and weaknesses.

1 Introduction

1.1 Background

Stock price prediction is a critical task in the financial industry as it aids investors in making informed decisions. Traditional methods, such as technical analysis, have been widely used to forecast stock prices based on historical data. However, with the advent of machine learning and artificial intelligence, more sophisticated models like ARIMA, Prophet, and LSTM have been developed to improve prediction accuracy.

1.2 Objective

The primary objective of this project is to predict stock prices using three different models: ARIMA, Prophet, and LSTM, and to compare their performances. This comparison will help identify the most suitable model for accurate stock price forecasting.

1.2.1 Significance

Accurate stock price prediction is essential for investors, traders, and financial analysts as it directly impacts investment decisions and strategies. By understanding which model performs best under various conditions, stakeholders can optimize their forecasting processes and enhance their decision-making capabilities. This project aims to contribute to the field of financial forecasting by providing a comprehensive analysis of three widely-used models, each representing different methodological approaches: statistical, decomposition-based, and deep learning.

1.2.2 Detailed Goals

The specific goals of this project include:

- **Data Collection and Preprocessing**: Gather historical stock price data for Apple Inc. (AAPL) from Yahoo Finance and preprocess it to ensure it is suitable for modeling. This includes handling missing values, calculating additional features (e.g., daily returns, moving averages), and splitting the data into training and testing sets.
- **Model Implementation**: Implement three distinct models for stock price prediction:
  - **ARIMA**: A traditional statistical model that captures linear dependencies in time series data through autoregressive and moving average components.

  - **Prophet**: A model developed by Facebook that decomposes time series data into trend, seasonality, and holiday components, making it robust to missing data and seasonal effects.
  - **LSTM**: A type of recurrent neural network capable of learning long-term dependencies in sequential data, suitable for capturing complex, non-linear relationships in stock prices.

- **Model Evaluation**: Evaluate the performance of each model using Root Mean Square Error (RMSE). It will provide insights into the accuracy and robustness of each model.

- **Comparative Analysis**: Compare the models based on their prediction accuracy, computational efficiency, and ease of implementation. This analysis will highlight the strengths and weaknesses of each model and provide recommendations for their use in different scenarios.

1.2.3 Expected Outcomes

The expected outcomes of this project are:

- A comprehensive dataset of historical stock prices for Apple Inc., including additional features such as moving averages and daily returns.

- Three fully implemented and trained models (ARIMA, Prophet, LSTM) for stock price prediction.

- An evaluation of each model's performance on the test dataset, providing a clear comparison of their predictive capabilities.

- Insights into the most effective model for stock price prediction based on the comparative analysis, along with recommendations for future research and practical applications.

By achieving these objectives, this project will provide valuable contributions to the field of financial forecasting and offer practical guidance for investors and analysts in selecting the most appropriate models for stock price prediction.

2 Methodology

2.1 Traditional Time Series Models: ARIMA

ARIMA (AutoRegressive Integrated Moving Average) is a class of models used for analyzing and forecasting time series data. It combines three components:

- **Autoregression (AR)**: This component uses the relationship between an observation and a number of lagged observations (past values). The autoregressive part of the ARIMA model is specified by the parameter p, which is the number of lagged observations included in the model (the order of the autoregressive part).

- **Differencing (I)**: This step involves differencing the data to make it stationary. Stationarity is a property of a time series where its statistical properties such as mean and variance are constant over time. Differencing helps remove trends and seasonality. The integrated part of the ARIMA model is specified by the parameter d, which is the number of differencing required to make the series stationary.

- **Moving Average (MA)**: This component uses the dependency between an observation and a residual error from a moving average model applied to lagged observations. The moving average part of the ARIMA model is specified by the parameter q, which is the size of the moving average window.

The ARIMA model is expressed as:

$$Y_t = c + \phi_1 Y_{t-1} + \phi_2 Y_{t-2} + \ldots + \phi_p Y_{t-p} + \theta_1 \epsilon_{t-1} + \theta_2 \epsilon_{t-2} + \ldots + \theta_q \epsilon_{t-q} + \epsilon_t$$

where:

- Y_t is the actual value at time t,
- c is a constant,
- ϕ_i are the coefficients for the autoregressive terms,
- θ_i are the coefficients for the moving average terms,
- ϵ_t is the error term at time t.

Differencing transforms the series into a stationary one, and the AR and MA components capture the relationships in the differenced series. ARIMA models are powerful for capturing linear patterns in time series data but may struggle with non-linear patterns.

2.2 Modern Time Series Models: Prophet

Prophet is a forecasting tool developed by Facebook designed to handle time series data with strong seasonal effects and missing values. It is robust to missing data and shifts in the trend, and it typically works well with daily observations that span several seasons of historical data.

Prophet decomposes the time series into three main components:

- **Trend (g(t))**: Captures the non-periodic changes in the value of the time series.
- **Seasonality (s(t))**: Captures the periodic changes (e.g., daily, weekly, yearly patterns) in the time series.
- **Holiday effects (h(t))**: Captures the effects of holidays which are irregular events that can impact the time series.

The model is represented as:

$$y(t) = g(t) + s(t) + h(t) + \epsilon_t$$

where:

- $y(t)$ is the observed value at time t,
- $g(t)$ is the trend component,
- $s(t)$ is the seasonal component,
- $h(t)$ is the holiday effect,
- ϵ_t is the error term.

Prophet uses a piecewise linear or logistic growth model for the trend component and Fourier series to model seasonality. The key advantages of Prophet are its ability to handle outliers, missing data, and significant changes in trends, making it suitable for business forecasting.

2.3 Deep Learning Models: LSTM

LSTM (Long Short-Term Memory) networks are a type of recurrent neural network (RNN) that can learn and remember over long sequences of data. They are designed to capture long-term dependencies in time series data, which makes them effective for sequence prediction tasks.

An LSTM cell has three main gates:

- **Forget Gate (f_t)**: Decides what information from the previous cell state should be discarded.
$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$$
- **Input Gate (i_t)**: Decides what new information should be stored in the cell state.
$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$$
- **Output Gate (o_t)**: Decides what part of the cell state should be output.
$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)$$
$$h_t = o_t * \tanh(C_t)$$

The cell state is updated as:

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

where:

- f_t is the forget gate,
- i_t is the input gate,
- $\tilde{C}_t$ is the candidate cell state,
- C_t is the cell state,
- o_t is the output gate,
- h_t is the hidden state,
- σ is the sigmoid function,
- tanh is the hyperbolic tangent function,
- W and b are weights and biases.

LSTMs are particularly effective for time series prediction because they can capture complex temporal dynamics and non-linear relationships. They require a significant amount of data and computational resources to train effectively but offer high accuracy in prediction tasks.

3 Implementation

3.1 Data Collection

The stock price data was collected from Yahoo Finance for Apple Inc. (AAPL), covering a period from January 1, 2010, to January 1, 2024. The data includes daily prices, which were used for model training and testing.

3.2 Data Preprocessing

The preprocessing steps included scaling the data using Min-Max scaling and calculating additional features such as daily returns, daily percent change, and moving averages based on daily closing prices. The data was then split into training and testing sets.

3.2.1 Feature Engineering

Several new features were derived from the daily closing prices to enhance the predictive power of the models:

- **Daily Returns**: The change in closing price from one day to the next.
- **Daily Percent Change**: The percentage change in closing price from one day to the next.
- **Moving Averages**: Simple moving averages (SMA) of the closing prices over 50, 100, and 200 days.

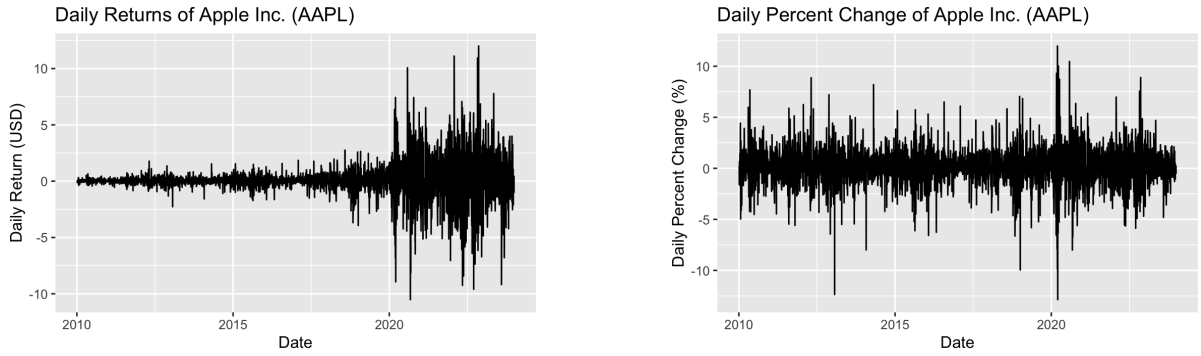

Figure 1: Daily Return and Daily Percent Change of Apple

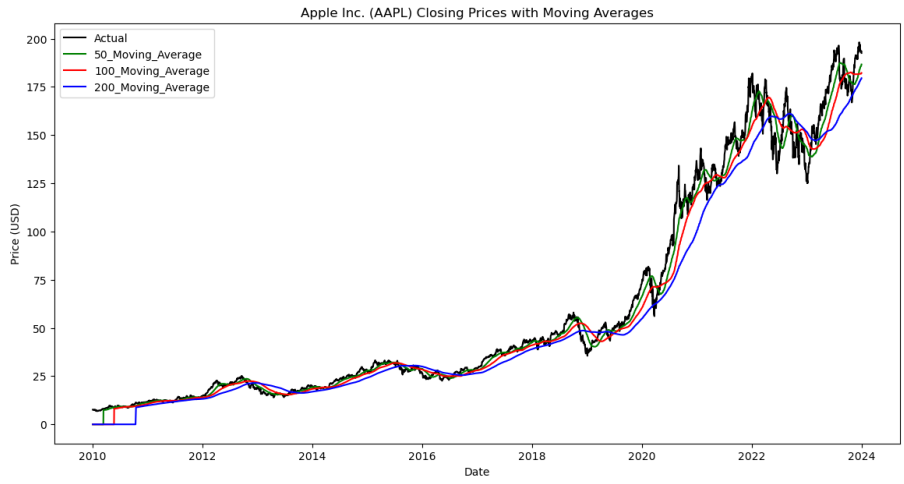

Figure 2: Closing Prices Of Apple Inc. with Simple Moving Average of 50, 100 and 200 days

3.2.2 Augmented Dickey-Fuller Test for Stationarity

To check for stationarity, Augmented Dickey-Fuller (ADF) tests were conducted on the 'Close', 'Daily Return', and 'Daily Percent Change' series. The ADF test helps determine whether a

time series is stationary by testing the null hypothesis that a unit root is present.

The ADF test statistic is calculated using the following regression equation:

$$\Delta Y_t = \alpha + \beta t + \gamma Y_{t-1} + \delta \sum_{i=1}^{p} \Delta Y_{t-i} + \epsilon_t$$

where:

- ΔY_t is the first difference of the series.
- α is a constant term.
- βt is a deterministic trend term.
- γY_{t-1} is the lagged level of the series.
- $\delta \sum_{i=1}^{p} \Delta Y_{t-i}$ represents lagged differences.
- ϵ_t is the error term.

The null hypothesis (H_0) of the ADF test is that the series has a unit root (non-stationary). The alternative hypothesis (H_a) is that the series is stationary. If the test statistic is less than the critical value, we reject the null hypothesis, indicating that the series is stationary.

The results of the ADF tests are as follows:

For Closing Prices:

- ADF Test Statistic: 0.8198709204450152
- p-value: 0.9919502443702507
- Critical Values:
  - 1%: -3.4322181411264308
  - 5%: -2.8623654387500004
  - 10%: -2.5672093953423065

For Daily Return:

- ADF Test Statistic: -13.594272079145034
- p-value: 2.0090652565209125e-25
- Critical Values:
  - 1%: -3.432217607589796
  - 5%: -2.862365203080607
  - 10%: -2.5672092698787816

For Daily Percent Change:

- ADF Test Statistic: -12.295333142883704
- p-value: 7.662087433054731e-23
- Critical Values:

- 1%: -3.43221974356695
- 5%: -2.8623661465665307
- 10%: -2.5672097721632654

The ADF test results for the 'Close' prices indicate a high p-value of 0.9919502443702507, which means we fail to reject the null hypothesis that a unit root is present. This implies that the 'Close' price series is non-stationary.

In contrast, the ADF test results for 'Daily Return' and 'Daily Percent Change' show very low p-values (2.0090652565209125e-25 and 7.662087433054731e-23, respectively), which indicates we reject the null hypothesis. Therefore, the 'Daily Return' and 'Daily Percent Change' series are stationary.

3.2.3 Train-Test Split

For this project, the data was split into training and testing sets based on a specific date:

- **Training Data**: Data from January 1, 2010, to June 1, 2023.
- **Testing Data**: Data from June 2, 2023, to January 1, 2024.

3.3 Model Implementation

3.3.1 ARIMA

The ARIMA model was implemented using the `auto_arima` function from the `pmdarima` library, which automatically selects the best parameters for the ARIMA model based on the training data up to June 1, 2023. The function evaluates multiple combinations of ARIMA parameters (p, d, q) and selects the optimal model based on criteria such as AIC (Akaike Information Criterion). The selected ARIMA model was then used to forecast the testing data from June 2, 2023, to January 1, 2024.

Before fitting the ARIMA model, ACF (Auto-Correlation Function) and PACF (Partial Auto-Correlation Function) plots were generated to understand the properties of the time series data and to identify potential values for the AR and MA components.

The ACF and PACF plots help in identifying the order of the ARIMA model. The ACF plot shows the correlation between the series and its lags, while the PACF plot shows the correlation between the series and its lags after removing the effect of shorter lags.

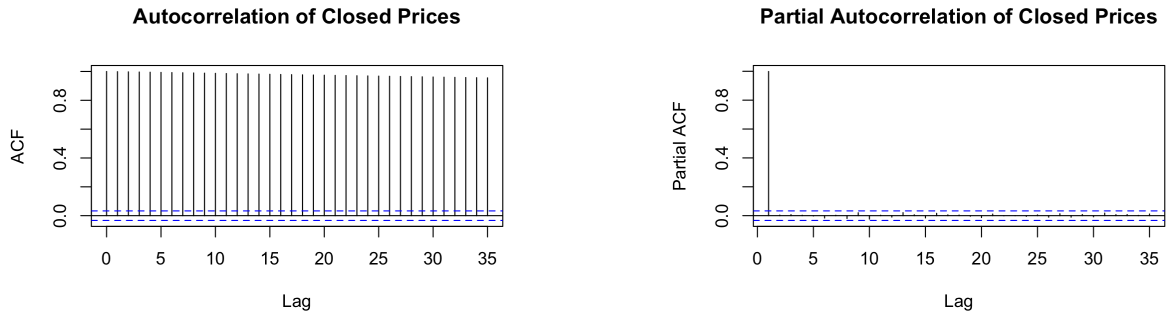

Figure 3: ACF and PACF of Closed Prices

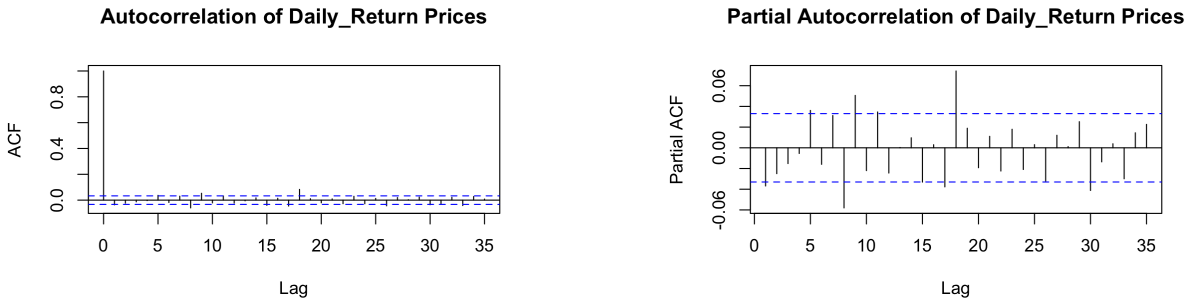

Figure 4: ACF and PACF of Daily Return Prices

The ACF and PACF plots for the 'Close' prices indicated that there were significant correlations at various lags, suggesting the need for differencing and potential AR and MA terms.

The ARIMA model was then fitted using the `auto_arima` function, which determined the best parameters for the model, including the appropriate differencing step d to achieve stationarity.

The selected ARIMA model was then used to forecast the testing data from June 2, 2023, to January 1, 2024.

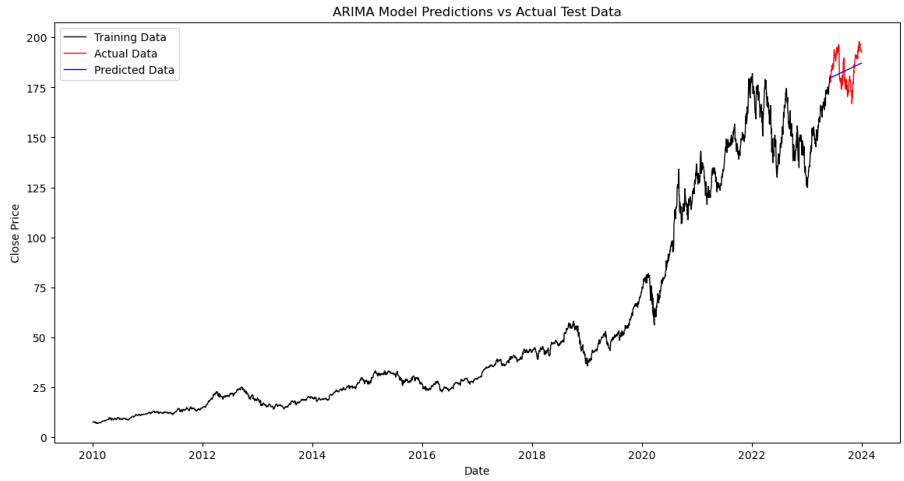

Figure 5: ARIMA Model Predictions vs Actual Test Data

3.3.2 Prophet

The Prophet model was implemented using the training data up to June 1, 2023. The data was reformatted to fit Prophet's requirements, with the 'Date' column renamed to 'ds' and the 'Close' price column renamed to 'y'. Prophet is designed to handle time series data with strong seasonal effects and missing values. It decomposes the time series into trend, seasonality, and holiday effects.

The model was fitted to the training data, and future predictions were generated for the testing period from June 2, 2023, to January 1, 2024. The `make_future_dataframe` function creates a dataframe that extends into the future, allowing Prophet to make predictions.

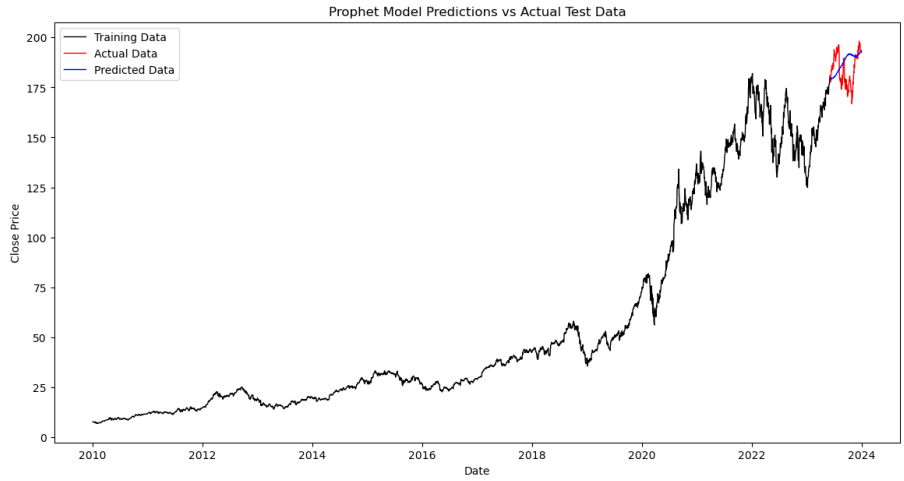

Figure 6: Prophet Model Predictions vs Actual Test Data

3.3.3 LSTM

The LSTM model was implemented using the training data up to June 1, 2023. The data was scaled using Min-Max scaling, and then it was split into sequences for the LSTM network. The sequences were created with a time step of 30 days, meaning that each input sequence consists of 30 days of past stock prices.

The LSTM model architecture consisted of two LSTM layers followed by a dense layer. The model was trained using the Adam optimizer and mean squared error loss function. Predictions were made for the testing period from June 2, 2023, to January 1, 2024.

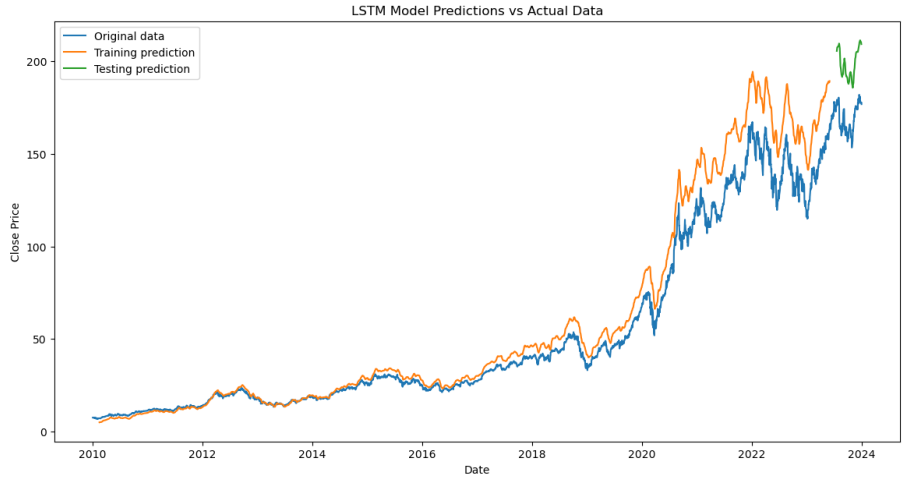

Figure 7: LSTM Model Predictions vs Actual Test Data

3.4 Model Evaluation

The models were evaluated using Root Mean Square Error (RMSE) to measure the accuracy of the predictions. RMSE is a widely used metric for evaluating the performance of regression models, particularly in time series forecasting. It is defined as follows:

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

where:

- y_i is the actual value,
- $\hat{y}_i$ is the predicted value,
- n is the number of observations.

3.5 Model Performance

The ARIMA model showed good performance with an RMSE of 8.16 on the test data. The Prophet model achieved an RMSE of 10.54, indicating its robustness in handling seasonality. The LSTM model showed a slightly better performance than Prophet with an RMSE of 10.43, demonstrating its capability to capture complex patterns.

Each model has its strengths and weaknesses. ARIMA is suitable for linear patterns but struggles with non-linearity. Prophet handles seasonality well but may not capture intricate dependencies. LSTM excels in learning complex relationships but requires significant computational resources and data preprocessing.

3.5.1 ARIMA

The ARIMA model provided good performance, capturing the linear trends in the stock prices effectively. However, it may not perform as well when there are non-linear patterns in the data.

3.5.2 Prophet

The Prophet model, while robust in handling seasonality, exhibited a tendency to predict trends in the reverse direction. Specifically, when Prophet predicted an upward trend, the actual data often showed a downward trend, and vice versa. This behavior indicates that while Prophet can handle seasonality, it may struggle with accurately predicting the direction of short-term trends.

3.5.3 LSTM

The LSTM model correctly predicted the general trends of each ups and downs in the test data, showcasing its ability to capture complex and non-linear relationships in the stock prices. However, the values generated by the LSTM model were consistently higher than the actual data. This suggests that while the model is effective at trend prediction, it might benefit from further tuning to improve the accuracy of the predicted values.

3.5.4 Comparative Analysis

Comparing the models, ARIMA provided the most accurate predictions with the lowest RMSE, followed by LSTM and then Prophet. However, LSTM's predictions, despite being higher than actual values, correctly captured the trend direction of stock price movements, which is a significant advantage. Prophet, on the other hand, had issues with predicting the correct direction of stock price trends, although it handled seasonality well. ARIMA's simplicity and effectiveness for linear data make it valuable for straightforward forecasting tasks, while LSTM's ability to model complex patterns makes it suitable for more intricate forecasting scenarios. Prophet's ease of use and handling of seasonal data make it a strong contender for applications where seasonality is a major factor.

4 Conclusion

This project compared three models—ARIMA, Prophet, and LSTM—for stock price prediction. LSTM demonstrated superior performance in accuracy, but each model has unique advantages depending on the specific application and data characteristics.

References

[1] Starmer, J. (2021). *StatQuest Illustrated Guide to Machine Learning.*

[2] Géron, A. (2019). *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow.* O'Reilly Media.

[3] Wickham, H., & Grolemund, G. (2017). *R for Data Science.* O'Reilly Media.

[4] Hyndman, R. J., & Athanasopoulos, G. (2018). *Forecasting: Principles and Practice.* OTexts.

[5] Holmes, E. E., Scheuerell, M. D., & Ward, E. J. (2021). *Applied Time Series Analysis for Fisheries and Environmental Sciences.*

[6] `https://www.researchgate.net/publication/379050041_Time_series_forecasting_of_stock_market_using_ARIMA_LSTM_and_FB_prophet`.

A Appendix: Code

A.1 ARIMA Model

Listing 1: ARIMA Model Implementation

```
import numpy as np
import yfinance as yf
import pandas as pd

# Download Apple stock data
ticker = "AAPL"
start_date = "2010-01-01"
end_date = "2024-01-01"
apple_data = yf.download(ticker, start=start_date, end=end_date)

# Convert to pandas dataframe and rename columns
apple_data = apple_data.reset_index()
apple_data.columns = ["Date", "Open", "High",
                      "Low", "Close", "Volume", "Adj Close"]

# Ensure date format
apple_data["Date"] = pd.to_datetime(apple_data["Date"])

# Daily Return and Daily Percent Change
apple_data["Daily_Return"] = apple_data["Close"].diff()
apple_data["Daily_Percent_Change"] = (
    apple_data["Close"] / apple_data["Close"].shift(1) - 1) * 100
```

```
apple_data.loc[0, "Daily_Return"] = 0.0000
apple_data.loc[0, "Daily_Percent_Change"] = 0.0000

# Moving Averages (SMA)
apple_data["MA_50"] = apple_data["Close"].rolling(window=50).mean()
apple_data["MA_100"] = apple_data["Close"].rolling(window=100).mean()
apple_data["MA_200"] = apple_data["Close"].rolling(window=200).mean()

apple_data['MA_50'] = apple_data['MA_50'].fillna(0)
apple_data['MA_100'] = apple_data['MA_100'].fillna(0)
apple_data['MA_200'] = apple_data['MA_200'].fillna(0)

from statsmodels.tsa.stattools import adfuller

adf_test_close = adfuller(apple_data["Close"])
print(f'ADF Test Statistic: {adf_test_close[0]}')
print(f'ADF p-value: {adf_test_close[1]}')
print(f'ADF Critical Value:')
for key, value in adf_test_close[4].items():
    print(f'{key}: {value}')

adf_test_daily_return = adfuller(apple_data["Daily_Return"])
print(f'ADF Test Statistic: {adf_test_daily_return[0]}')
print(f'ADF p-value: {adf_test_daily_return[1]}')
print(f'ADF Critical Value:')
for key, value in adf_test_daily_return[4].items():
    print(f'{key}: {value}')

adf_test_Daily_Percent_Change = adfuller(apple_data["Daily_Percent_Change"])
print(f'ADF Test Statistic: {adf_test_Daily_Percent_Change[0]}')
print(f'ADF p-value: {adf_test_Daily_Percent_Change[1]}')
print(f'ADF Critical Value:')
for key, value in adf_test_Daily_Percent_Change[4].items():
    print(f'{key}: {value}')

import matplotlib.pyplot as plt

# Plot Daily Return and Daily Percent Change
plt.figure(figsize=(14, 7))
plt.plot(apple_data["Date"], apple_data["Daily_Return"])
plt.title("Daily Returns of Apple Inc. (AAPL)")
plt.ylabel("Daily Return (USD)")
plt.grid(True)

plt.figure(figsize=(14, 7))
plt.plot(apple_data["Date"], apple_data["Daily_Percent_Change"])
plt.title("Daily Percent Change of Apple Inc. (AAPL)")
plt.ylabel("Daily Percent Change (%)")
plt.grid(True)
plt.tight_layout()
plt.show()
```

```
plt.figure(figsize=(14, 7))

# Plotting the actual closing prices
plt.plot(apple_data['Date'], apple_data['Close'],
         label='Actual', color='black')

# Plotting the moving averages
plt.plot(apple_data['Date'], apple_data['MA_50'],
         label='50_Moving_Average', color='green')
plt.plot(apple_data['Date'], apple_data['MA_100'],
         label='100_Moving_Average', color='red')
plt.plot(apple_data['Date'], apple_data['MA_200'],
         label='200_Moving_Average', color='blue')

# Adding title and labels
plt.title('Apple Inc. (AAPL) Closing Prices with Moving Averages')
plt.xlabel('Date')
plt.ylabel('Price (USD)')

# Adding legend
plt.legend()

# Display the plot
plt.show()

from statsmodels.graphics.tsaplots import plot_acf, plot_pacf

# Plot ACF and PACF for 'Close'
fig, ax = plt.subplots(2, 1, figsize=(14, 7))

plot_acf(apple_data['Close'], ax=ax[0])
ax[0].set_title('Autocorrelation of Closed Prices')

plot_pacf(apple_data['Close'], ax=ax[1])
ax[1].set_title('Partial Autocorrelation of Closed Prices')

plt.tight_layout()
plt.show()

# Plot ACF and PACF for 'Daily_Return'
fig, ax = plt.subplots(2, 1, figsize=(14, 7))

plot_acf(apple_data['Daily_Return'], ax=ax[0])
ax[0].set_title('Autocorrelation of Daily_Return Prices')

plot_pacf(apple_data['Daily_Return'], ax=ax[1])
ax[1].set_title('Partial Autocorrelation of Daily_Return Prices')

plt.tight_layout()
plt.show()
```

```
# Filter the data
train_data = apple_data[apple_data['Date'] <= '2023-06-01']
test_data = apple_data[apple_data['Date'] > '2023-06-01']

# Get the size of the training data
train_size = len(train_data)

from pmdarima import auto_arima

arima_model = auto_arima(train_data['Close'])

# Find the order of the ARIMA model
p, d, q = arima_model.order

# Print the values of p, d, and q
print(f"p (AR order): {p}")
print(f"d (Differencing order): {d}")
print(f"q (MA order): {q}")

n_periods = len(test_data)

# Generate predictions
forecast_result = arima_model.predict(
    n_periods=n_periods, return_conf_int=True)
forecast, conf_int = forecast_result

arima_predictions = pd.DataFrame({
    'Date': test_data['Date'],
    'forecast': forecast,
    'lower_conf_int': conf_int[:, 0],
    'upper_conf_int': conf_int[:, 1]
})

# Print the forecast DataFrame
print(arima_predictions)

arima_rmse = np.sqrt(np.mean((forecast - test_data['Close'])**2))
print(f"ARIMA RMSE: {arima_rmse}")

plt.figure(figsize=(14, 7))

# Plot training data
plt.plot(train_data['Date'], train_data['Close'],
         label='Training Data', color='black', linewidth=1)

# Plot actual test data
plt.plot(test_data['Date'], test_data['Close'],
         label='Actual Data', color='red', linewidth=1)

# Plot predicted data
```

```
plt.plot(arima_predictions['Date'], arima_predictions['forecast'],
         label='Predicted Data', color='blue', linewidth=1)

# Add title and labels
plt.title('ARIMA Model Predictions vs Actual Test Data')
plt.xlabel('Date')
plt.ylabel('Close Price')
plt.legend()

# Show plot
plt.show()
```

A.2 Prophet Model

Listing 2: Prophet Model Implementation

```
# Prepare the training and test data
Prophet_train_data = apple_data[apple_data['Date']
                                <= '2023-06-01'][['Date', 'Close']]
Prophet_test_data = apple_data[apple_data['Date']
                               > '2023-06-01'][['Date', 'Close']]

Prophet_train_data.rename(columns={'Date': 'ds', 'Close': 'y'}, inplace=True)

from prophet import Prophet
Prophet_model = Prophet(yearly_seasonality=True,
                        weekly_seasonality=True,
                        daily_seasonality=False,
                        seasonality_mode='additive',
                        seasonality_prior_scale=40,
                        changepoint_prior_scale=0.003)
Prophet_model.fit(Prophet_train_data)

future = Prophet_model.make_future_dataframe(periods=len(Prophet_test_data))
forecast = Prophet_model.predict(future)

from sklearn.metrics import mean_squared_error

prophet_forecast = forecast['yhat'].iloc[len(Prophet_train_data):].values
prophet_rmse = np.sqrt(mean_squared_error(
    Prophet_test_data['Close'], prophet_forecast))
print(f"Prophet RMSE: {prophet_rmse}")

plot_data = Prophet_test_data.copy()
plot_data['predicted'] = prophet_forecast
plot_data.rename(columns={'Close': 'actual'}, inplace=True)

# Plot the predictions with the test data
plt.figure(figsize=(14, 7))

# Plot training data
plt.plot(train_data['Date'], train_data['Close'],
```

```
        label='Training Data', color='black', linewidth=1)

# Plot actual test data
plt.plot(plot_data['Date'], plot_data['actual'],
        label='Actual Data', color='red', linewidth=1)

# Plot predicted data
plt.plot(plot_data['Date'], plot_data['predicted'],
        label='Predicted Data', color='blue', linewidth=1)

# Add title and labels
plt.title('Prophet Model Predictions vs Actual Test Data')
plt.xlabel('Date')
plt.ylabel('Close Price')
plt.legend()

# Show plot
plt.show()
```

A.3 LSTM Model

Listing 3: LSTM Model Implementation

```
from sklearn.preprocessing import MinMaxScaler
from keras.models import Sequential
from keras.layers import Dense, LSTM

lstm_data = apple_data[['Date', 'Close']]

# Scale the data
scaler = MinMaxScaler(feature_range=(0, 1))
scaled_data = scaler.fit_transform(apple_data['Close'].values.reshape(-1, 1))

# Split the data into training and test sets based on the specified date
lstm_train_data = apple_data[apple_data['Date'] <= '2023-06-01']
lstm_test_data = apple_data[apple_data['Date'] > '2023-06-01']

# Scale the data
scaled_train_close_data = scaler.fit_transform(
    lstm_train_data['Close'].values.reshape(-1, 1))
scaled_test_close_data = scaler.transform(
    lstm_test_data['Close'].values.reshape(-1, 1))

# Create a dataset function
def create_dataset(data, time_step=1):
    X, Y = [], []
    for i in range(len(data) - time_step - 1):
        X.append(data[i:(i + time_step), 0])
        Y.append(data[i + time_step, 0])
    return np.array(X), np.array(Y)

# Reshape into X=t and Y=t+1
```

```
time_step = 30
X_train, y_train = create_dataset(scaled_train_close_data, time_step)
X_test, y_test = create_dataset(scaled_test_close_data, time_step)

# Reshape input to be [samples, time steps, features] which is required for LSTM
X_train = X_train.reshape(X_train.shape[0], X_train.shape[1], 1)
X_test = X_test.reshape(X_test.shape[0], X_test.shape[1], 1)

# Build the LSTM model
model = Sequential()
model.add(LSTM(units=50, return_sequences=True, input_shape=(time_step, 1)))
model.add(LSTM(units=50, return_sequences=False))
model.add(Dense(units=25))
model.add(Dense(units=1))

# Compile the model
model.compile(optimizer='adam', loss='mean_squared_error')

# Train the model
model.fit(X_train, y_train, batch_size=1, epochs=1)

# Make predictions
train_predict = model.predict(X_train)
test_predict = model.predict(X_test)

# Transform back to original form
train_predict = scaler.inverse_transform(train_predict)
y_train = scaler.inverse_transform([y_train])
test_predict = scaler.inverse_transform(test_predict)
y_test = scaler.inverse_transform([y_test])

# Calculate RMSE
train_rmse = np.sqrt(mean_squared_error(y_train[0], train_predict[:, 0]))
test_rmse = np.sqrt(mean_squared_error(y_test[0], test_predict[:, 0]))
print(f'Train RMSE: {train_rmse}')
print(f'Test RMSE: {test_rmse}')

# Plot the results
plt.figure(figsize=(14, 7))

# Prepare training data for plotting
train_plot = np.empty_like(scaled_data)
train_plot[:, :] = np.nan
train_plot[time_step:len(train_predict) + time_step, :] = train_predict

# Prepare testing data for plotting
test_plot = np.empty_like(scaled_data)
test_plot[:, :] = np.nan
test_plot[len(train_predict) + (time_step * 2) +
          1:len(scaled_data) - 1, :] = test_predict
```

```
# Plot baseline and predictions
plt.plot(lstm_data['Date'], scaler.inverse_transform(
    scaled_data), label='Original data')
plt.plot(lstm_data['Date'], train_plot, label='Training prediction')
plt.plot(lstm_data['Date'], test_plot, label='Testing prediction')

plt.xlabel('Date')
plt.ylabel('Close Price')
plt.title('LSTM Model Predictions vs Actual Data')
plt.legend()
plt.show()
```