Artificial Intelligence and Machine Learning

Project Report

Semester-IV (Batch-2022)

Stock Price Predictor

https://github.com/Sparsh865/Stock-Price-Predictor

Supervised By:

Mrs. Rishu Taneja

Submitted By:

Sparsh Mittal

2210990865

Group -13

Department of Computer Science and Engineering
Chitkara University Institute of Engineering & Technology,
Chitkara University, Punjab

Abstract

Stock price prediction has long been a challenging yet crucial task in financial analysis. This project tackles this challenge by leveraging Artificial Intelligence and Machine Learning (AIML) techniques, particularly Long Short-Term Memory (LSTM) models. Integrated with Python libraries, the project encompasses data collection from Yahoo Finance, preprocessing, model training, and visualization using Streamlit. Historical stock data is cleaned, split into training/testing sets, and normalized before input into the LSTM model. Predicted prices are compared against actual prices, offering insights into future market trends. This project presents a streamlined approach to stock price prediction, vital for informed decision-making in financial markets.

Table of Contents

Introduction

Background

Stock price prediction is a cornerstone of financial analysis, providing critical information for investors, traders, and financial institutions to navigate the complexities of the stock market. Historically, various methods have been employed to forecast stock prices, ranging from fundamental analysis, which assesses a company's financial health and market position, to technical analysis, which relies on historical price patterns and indicators to predict future movements. However, these traditional approaches often face challenges in capturing the intricate dynamics of financial markets, especially in volatile and unpredictable conditions.

In recent years, the emergence of Artificial Intelligence and Machine Learning (AIML) has revolutionized the field of stock price prediction. AIML techniques, particularly deep learning models such as Long Short-Term Memory (LSTM) networks, have shown remarkable promise in capturing complex patterns in time-series data, making them well-suited for predicting stock prices. Unlike traditional methods, which may struggle to handle nonlinear relationships and temporal dependencies in data, LSTM models excel at learning from sequential data and capturing long-term dependencies, making them highly effective for forecasting stock price trends.

Objectives

1. Develop an integrated AIML-based system capable of accurately predicting stock price trends.
2. Explore the effectiveness of LSTM models, known for their ability to capture temporal dependencies in sequential data, in forecasting stock prices.
3. Implement a robust data preprocessing pipeline to handle missing values, outliers, and feature engineering, ensuring the quality and reliability of input data.
4. Train and optimize LSTM models using historical stock data to generate reliable predictions of future stock prices.
5. Create an interactive visualization platform to present predicted price trends alongside historical data, facilitating easy interpretation and analysis by stakeholders.
6. Evaluate the performance of the prediction model using quantitative metrics such as R-2 squared, as well as qualitative analysis of predicted versus actual price movements.

Significance

1. **Informed Decision Making:** Accurate stock price predictions provide investors, traders, and financial institutions with valuable insights into potential market movements, enabling them to make more informed investment decisions and mitigate risk.

2. **Efficiency and Automation:** By automating the prediction process using AIML techniques, financial professionals can save time and resources previously spent on manual analysis, allowing them to focus on higher-level strategic tasks.

3. **Market Understanding and Insights:** Analysis of predicted price trends and comparison with actual market data yield valuable insights into market dynamics, trends, and patterns, enhancing our understanding of financial markets and facilitating more accurate future predictions.

4. **Technological Innovation:** Leveraging advanced AIML techniques for stock price prediction demonstrates innovation in financial analysis, pushing the boundaries of what is possible and paving the way for future advancements in predictive modeling and algorithmic trading strategies.

Problem Definition and Requirements

Problem Definition

The problem of stock price prediction entails accurately forecasting future stock prices based on historical data and relevant market indicators. This task is critical for investors, traders, and financial institutions to make informed decisions regarding buying, selling, or holding stocks. However, predicting stock prices is challenging due to the complex and dynamic nature of financial markets, where prices are influenced by a multitude of factors such as economic indicators, company performance, geopolitical events, and investor sentiment. Traditional methods of stock price prediction often rely on simplistic models or manual analysis, which may not effectively capture the underlying patterns and relationships in the data, leading to inaccurate forecasts.

The objective is to develop an advanced stock price prediction system that leverages state-of-the-art Artificial Intelligence and Machine Learning techniques, particularly deep learning models like Long Short-Term Memory (LSTM) networks. The system should be capable of processing large volumes of historical stock market data, extracting relevant features, training predictive models, and generating actionable insights for stakeholders. By accurately forecasting stock price movements, the system aims to empower investors and traders with valuable information to optimize their investment strategies and mitigate risk.

Software Requirements

- **Programming Language:** Python - Python is a versatile and widely-used programming language with extensive support for data analysis, machine learning, and web development.

- **Libraries and Frameworks:**
    1. **NumPy:** NumPy is a powerful library for numerical computing in Python, providing support for large multi-dimensional arrays and matrices, along with a collection of mathematical functions to operate on these arrays efficiently.

    2. **Pandas:** Pandas is a data manipulation library that offers data structures like DataFrame and Series, making it easy to manipulate and analyze structured data. It provides functionalities for data cleaning, reshaping, and aggregation.

    3. **Matplotlib:** Matplotlib is a plotting library that enables the creation of various types of static, interactive, and animated visualizations in Python. It offers a wide range of customization options for creating publication-quality plots.

    4. **Seaborn:** Seaborn is built on top of Matplotlib and provides a high-level interface for creating attractive and informative statistical graphics. It simplifies the process of creating complex visualizations like heatmaps, violin plots, and pair plots.

    5. **Scikit-learn:** Scikit-learn is a comprehensive machine learning library that provides simple and efficient tools for data mining and data analysis. It offers algorithms for classification, regression, clustering, dimensionality reduction, and model selection.

    6. **TensorFlow:** TensorFlow is an open-source deep learning framework developed by Google, offering a flexible ecosystem for building and deploying machine learning models. It provides support for building various types of neural networks, including LSTM networks.

7. **Keras:** Keras is a high-level neural networks API written in Python and designed to enable fast experimentation with deep learning models. It offers a user-friendly interface for building and training deep learning models, including LSTM networks.

8. **Yfinance:** Yfinance is a Python library that provides a convenient interface to Yahoo Finance API, allowing users to fetch historical market data, including stock prices, dividends, and corporate actions.

9. **Streamlit:** Streamlit is an open-source Python library for creating web applications and interactive data dashboards with minimal code. It allows developers to build interactive user interfaces for data exploration, visualization, and analysis directly from Python scripts.

10. **Openpyxl:** Openpyxl is a Python library for reading and writing Excel files (XLSX), allowing users to manipulate spreadsheet data programmatically.

- **Development Environment:** Jupyter Notebook or any Python IDE (e.g., PyCharm, Visual Studio Code) - Jupyter Notebook provides an interactive computing environment for writing and executing Python code, making it ideal for exploratory data analysis and prototyping machine learning models. Python IDEs offer integrated development environments with features like code editing, debugging, and version control integration.

- **Version Control Git** - Git is a distributed version control system that allows multiple developers to collaborate on a project efficiently. It enables tracking changes to the codebase, managing different versions of the project, and facilitating collaboration through features like branching and merging.

Dataset

The dataset used in this project is obtained from Yahoo Finance through the yfinance Python library. It comprises historical stock market data for the specified stock ticker, covering a defined period.
The dataset includes various fields that are essential for stock price prediction, including but not limited to:

Input Fields:

- **Date:** The date of the stock market data entry.
- **Open Price:** The opening price of the stock on the given date.
- **High Price:** The highest price of the stock reached during the trading day.
- **Low Price:** The lowest price of the stock reached during the trading day.
- **Close Price:** The closing price of the stock on the given date.
- **Volume:** The total number of shares traded during the trading day.
- **Adjusted Close Price:** The closing price adjusted for dividends, stock splits, and other corporate actions.

Output Field:

- **Close Price:** This is the target variable for stock price prediction, representing the closing price of the stock on the next trading day. The objective of the model is to predict this future price based on historical data and relevant market indicators.

The input fields serve as features or predictors for the model, providing historical information about the stock's performance, trading activity, and market conditions. The model utilizes these input fields to learn patterns and relationships that can help predict the future close price of the stock.

The output field, i.e., the close price of the next trading day, is the target variable that the model aims to predict. By training on historical data with known close prices and corresponding input features, the model learns to generalize and make accurate predictions for unseen data, enabling investors and traders to anticipate future stock price movements and make informed decisions.

Proposed Design/ Methodology

Design and Working

1. **User Interface (UI):**

- The user interface is designed using Streamlit, a Python library for creating web applications. Users interact with the application through a web browser.
- The UI includes a title, a text input field for entering the stock ticker symbol, and various sections for displaying data, visualizations, and predictions.

2. **Data Retrieval and Preprocessing:**

- Stock data is retrieved from Yahoo Finance using the yfinance library based on the user's input (stock ticker symbol) and a specified date range.
- The retrieved data is stored in a pandas DataFrame and saved to an Excel file named "Dow 30.xlsx".
- Data preprocessing involves handling missing values and removing duplicate columns from the DataFrame to ensure data cleanliness and consistency.

3. **Data Analysis and Visualization:**

- Descriptive statistics of the stock data are computed and displayed using Streamlit.
- Various visualizations are generated using Matplotlib, including:
- Closing price vs. time chart
- Closing price vs. time chart with 100-day and 200-day moving averages

4. **Model Training and Testing:**

- The project employs a Long Short-Term Memory (LSTM) model for predicting stock prices.
- The LSTM model is trained separately in a Jupyter Notebook (LSTM model.ipynb).
- Training data is scaled using MinMaxScaler to ensure numerical stability and convergence during model training.
- After training, the LSTM model is saved as an HDF5 file named "LSTM_model.h5".

5. **Prediction and Evaluation:**

- Testing data is prepared by combining past 100 days' data from the training set with the remaining data.
- The trained LSTM model is loaded from the saved HDF5 file.
- Predictions are made using the loaded model on the testing data.
- Predictions are scaled back to their original values for evaluation and comparison with the actual stock prices.
- Predictions vs. original prices are plotted and displayed using Matplotlib and Streamlit for visual evaluation.

Folder Structure

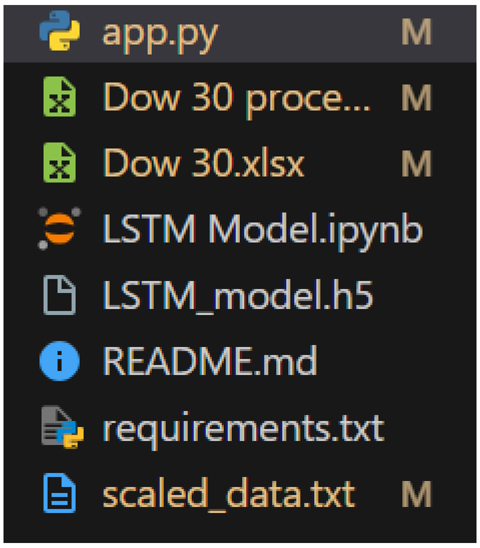

app.py M
Dow 30 proce... M
Dow 30.xlsx M
LSTM Model.ipynb
LSTM_model.h5
README.md
requirements.txt
scaled_data.txt M

Figure 1 Folder Structure

- **app.py:** The main Python script containing the code for the stock trend prediction application.
- **Dow 30.xlsx:** Excel file containing historical stock data downloaded from Yahoo Finance for the user input from January 1, 2019, to December 31, 2023.
- **Dow 30 processed.xlsx:** Excel file containing the processed and cleaned-up data from Dow 30.
- **LSTM Model.ipynb:** Jupyter Notebook file where the LSTM model for stock price prediction is created and trained.
- **LSTM_model.h5:** The trained LSTM model saved in Hierarchical Data Format version 5 (HDF5) file format.
- **README.md:** Readme file providing information about the project, its components, and instructions for running the code.
- **requirements.txt:** Text file containing a list of libraries required to run the project. This file is used with package managers like pip for installing dependencies.
- **scaled_data.txt:** Text file containing scaled data used in the project.

Machine Learning Technique

Stock price prediction typically falls under supervised learning because it involves using historical data (input) with corresponding stock prices (output) to train a model. In this context:

- **Labeled Data:** The historical data consists of various features such as past stock prices, trading volume, financial indicators, and other relevant information. Each data point (e.g., each day) is associated with the corresponding stock price (output label).

- **Training Process:** During the training phase, the algorithm learns the patterns and relationships between the input features and the corresponding stock prices. It iteratively adjusts its parameters to minimize the discrepancy between its predicted stock prices and the actual observed prices from the historical data.

- **Regression Task:** Stock price prediction is typically formulated as a regression problem where the goal is to predict a continuous value (the stock price) based on the input features. The algorithm aims to approximate the underlying mapping function between the input variables (historical data) and the output variable (stock price).

- **Supervision:** The historical stock prices act as supervision for the learning process. The algorithm learns from past data, guided by the known outcomes, and aims to generalize this knowledge to make predictions on unseen data (future stock prices).

LSTM Model

Long Short-Term Memory (LSTM) is a type of recurrent neural network (RNN) architecture designed to address the vanishing gradient problem, which occurs when training traditional RNNs on long sequences of data. LSTMs are well-suited for modeling sequential data and have been widely used in various fields, including natural language processing, speech recognition, and time series prediction, such as stock price forecasting.

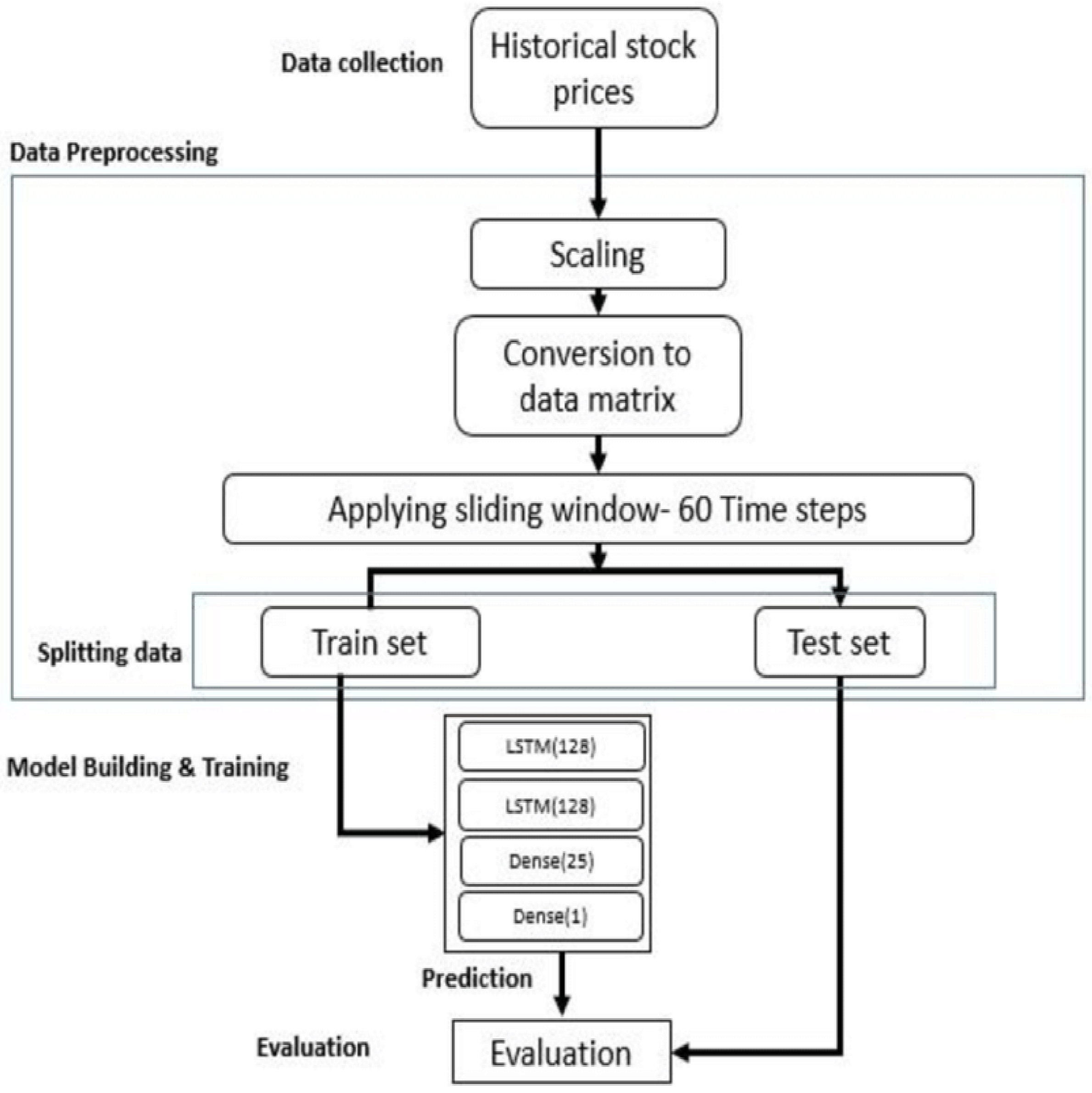
Usage for Stock Price Prediction:

- **Sequence Modeling:** Stock price data can be treated as a sequence, where each data point represents the price at a specific time. LSTMs are well-suited for modeling such sequential data due to their ability to capture temporal dependencies.
- **Feature Extraction:** LSTMs can automatically learn relevant features from the input data, such as patterns, trends, and seasonality, without the need for manual feature engineering.
- **Temporal Dynamics:** Stock prices exhibit complex temporal dynamics influenced by various factors such as market sentiment, economic indicators, and company performance. LSTMs are capable of capturing these dynamics and making predictions based on historical patterns.

Implementation in the Project:

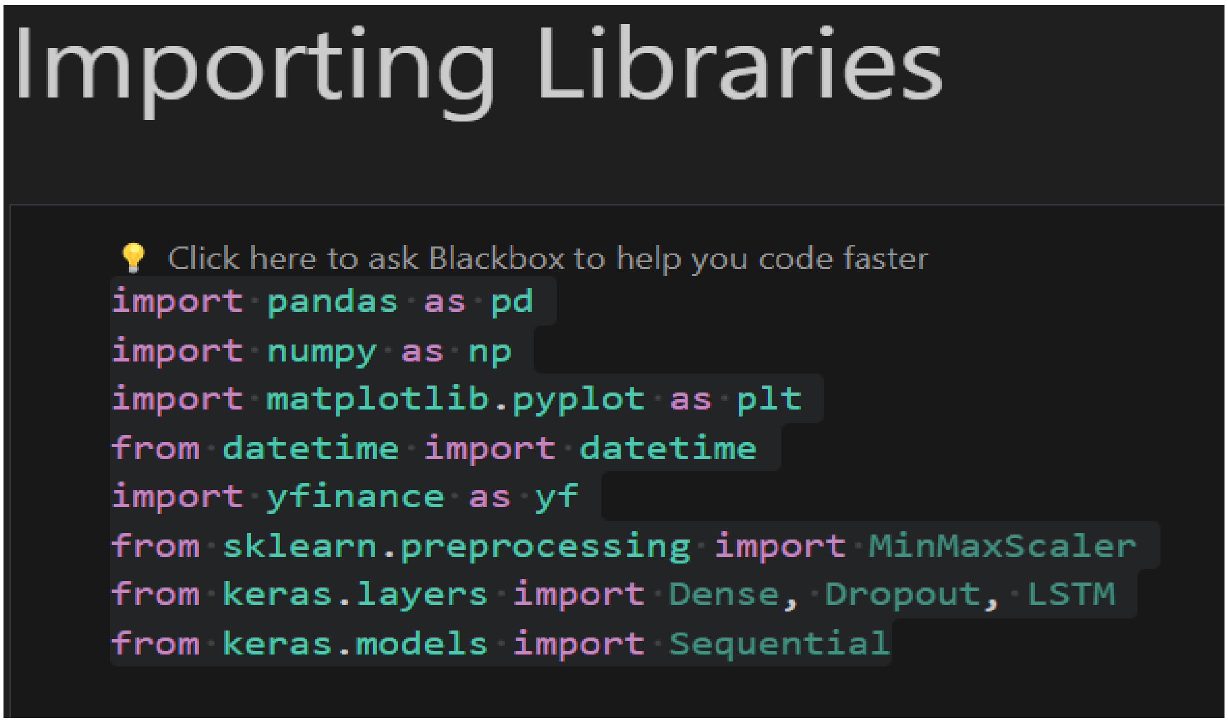
- **Model Training:** In the project, an LSTM model is trained using historical stock price data. The model learns from the sequential patterns in the data to make predictions about future price movements.
- **Temporal Dependencies:** The LSTM model captures temporal dependencies in the stock price data, allowing it to learn from past prices and make predictions based on historical patterns.
- **Long-Term Trends:** LSTMs are capable of capturing long-term trends in stock prices, making them suitable for forecasting over extended time horizons.
- **Predictive Performance:** The trained LSTM model is evaluated on testing data to assess its predictive performance. Metrics such as Mean Absolute Error (MAE) or Root Mean Squared Error (RMSE) may be used to quantify the accuracy of the predictions.

Schematic Diagram



Results

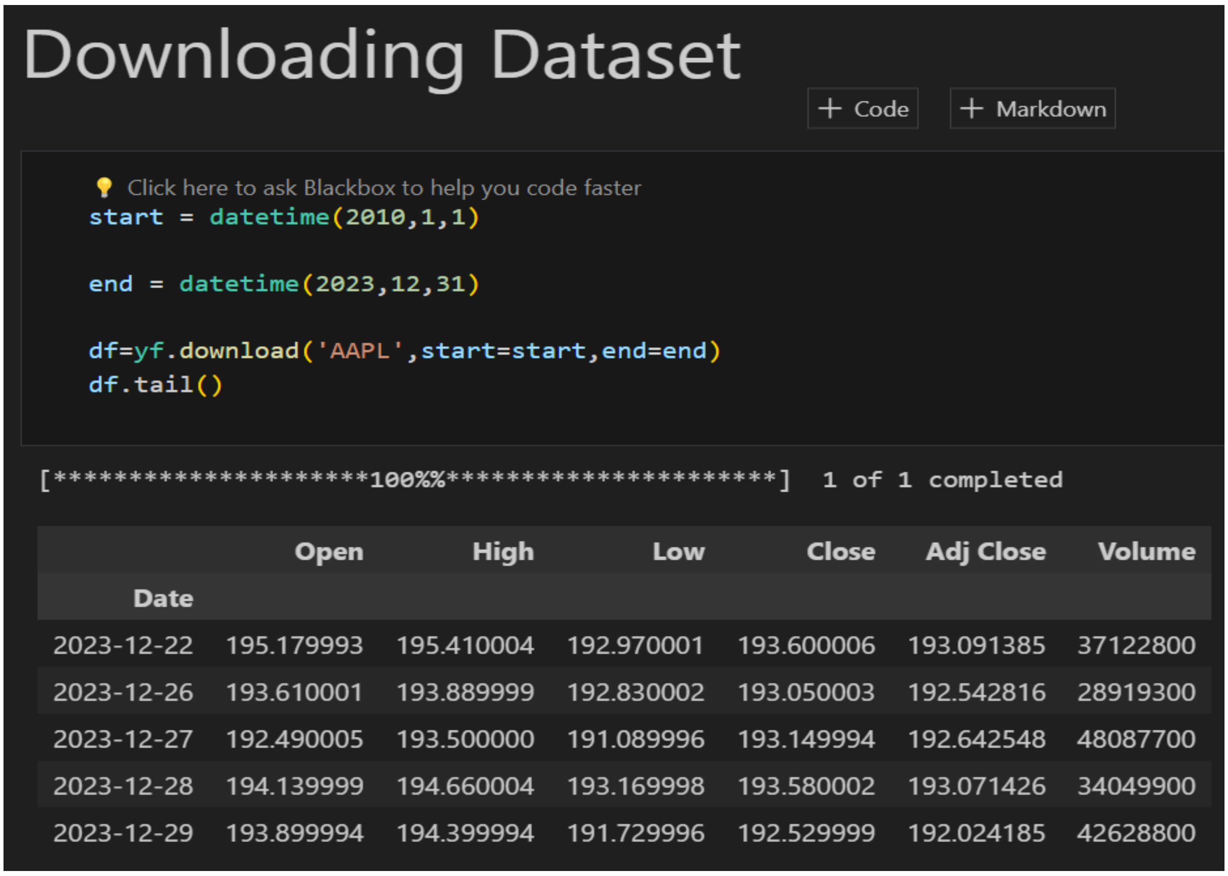
LSTM Model



Importing Libraries

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from datetime import datetime
import yfinance as yf
from sklearn.preprocessing import MinMaxScaler
from keras.layers import Dense, Dropout, LSTM
from keras.models import Sequential
```

Figure 2 Importing Libraries



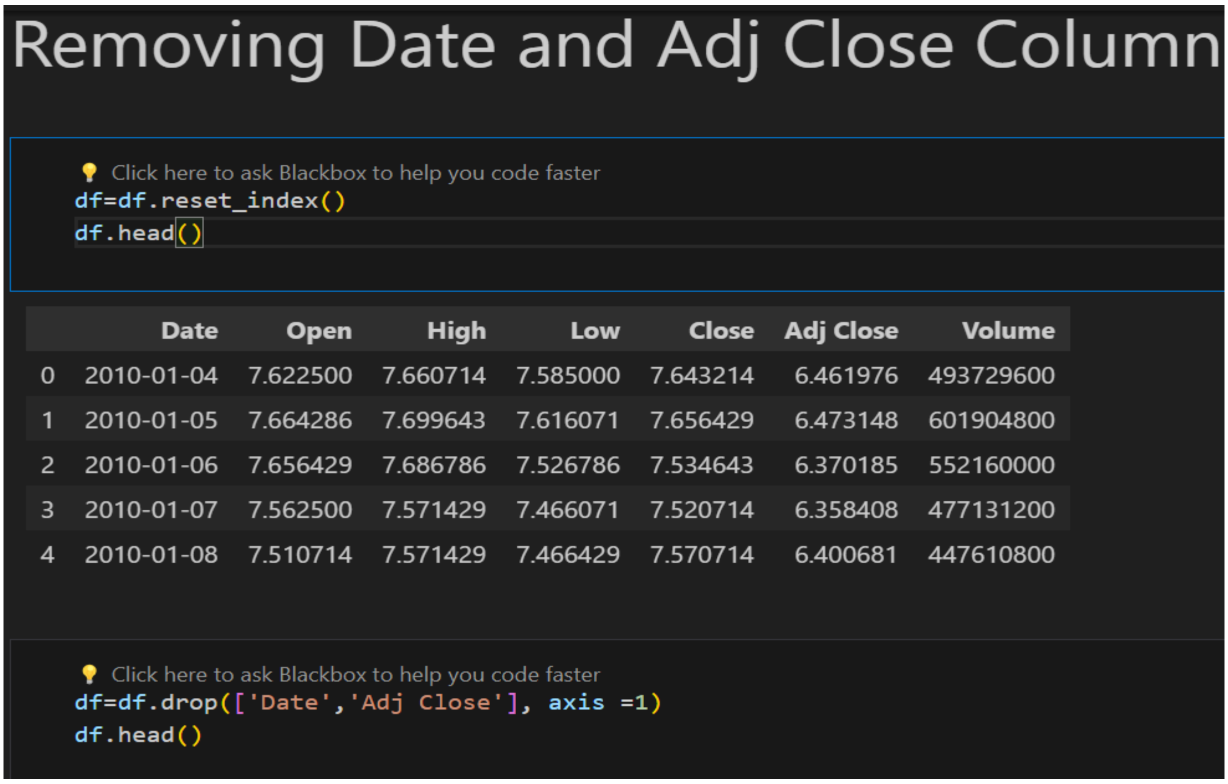
Downloading Dataset

```
start = datetime(2010,1,1)

end = datetime(2023,12,31)

df=yf.download('AAPL',start=start,end=end)
df.tail()
```

```
[*********************100%%**********************]  1 of 1 completed
```

	Open	High	Low	Close	Adj Close	Volume
Date						
2023-12-22	195.179993	195.410004	192.970001	193.600006	193.091385	37122800
2023-12-26	193.610001	193.889999	192.830002	193.050003	192.542816	28919300
2023-12-27	192.490005	193.500000	191.089996	193.149994	192.642548	48087700
2023-12-28	194.139999	194.660004	193.169998	193.580002	193.071426	34049900
2023-12-29	193.899994	194.399994	191.729996	192.529999	192.024185	42628800

Figure 3 Downloading Dataset



Removing Date and Adj Close Column

Click here to ask Blackbox to help you code faster

```
df=df.reset_index()
df.head()
```

	Date	Open	High	Low	Close	Adj Close	Volume
0	2010-01-04	7.622500	7.660714	7.585000	7.643214	6.461976	493729600
1	2010-01-05	7.664286	7.699643	7.616071	7.656429	6.473148	601904800
2	2010-01-06	7.656429	7.686786	7.526786	7.534643	6.370185	552160000
3	2010-01-07	7.562500	7.571429	7.466071	7.520714	6.358408	477131200
4	2010-01-08	7.510714	7.571429	7.466429	7.570714	6.400681	447610800

Click here to ask Blackbox to help you code faster

```
df=df.drop(['Date','Adj Close'], axis =1)
df.head()
```

Figure 4 Making Desired Dataframe

Click here to ask Blackbox to help you code faster

```
plt.plot(df.Close)
```

```
[<matplotlib.lines.Line2D at 0x1b8fd879ed0>]
```

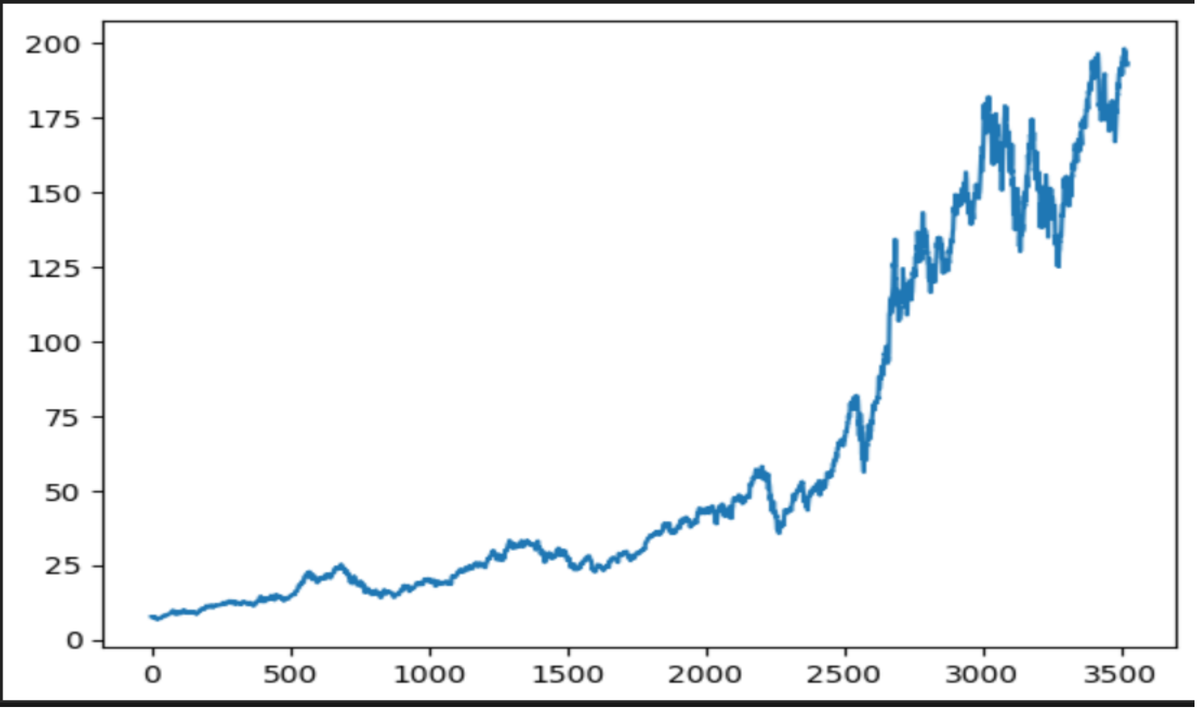
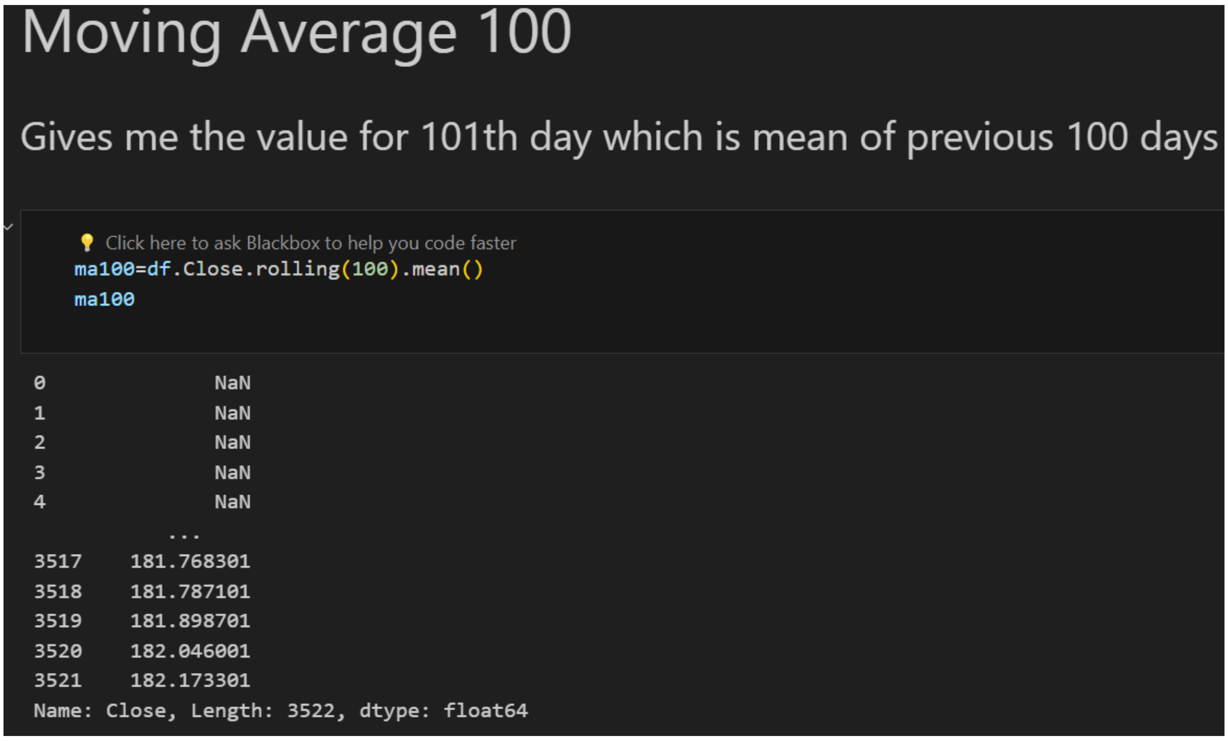


Figure 5 Plotting Graph on Closing Value



Moving Average 100

Gives me the value for 101th day which is mean of previous 100 days

💡 Click here to ask Blackbox to help you code faster

```
ma100=df.Close.rolling(100).mean()
ma100
```

```
0              NaN
1              NaN
2              NaN
3              NaN
4              NaN
           ...
3517    181.768301
3518    181.787101
3519    181.898701
3520    182.046001
3521    182.173301
Name: Close, Length: 3522, dtype: float64
```

Figure 6 Moving Average 100

💡 Click here to ask Blackbox to help you code faster

```
plt.figure(figsize=(8,4))
plt.plot(df.Close)
plt.plot(ma100,'red')
```

✓ 0.3s

```
[<matplotlib.lines.Line2D at 0x1b88f4d6210>]
```

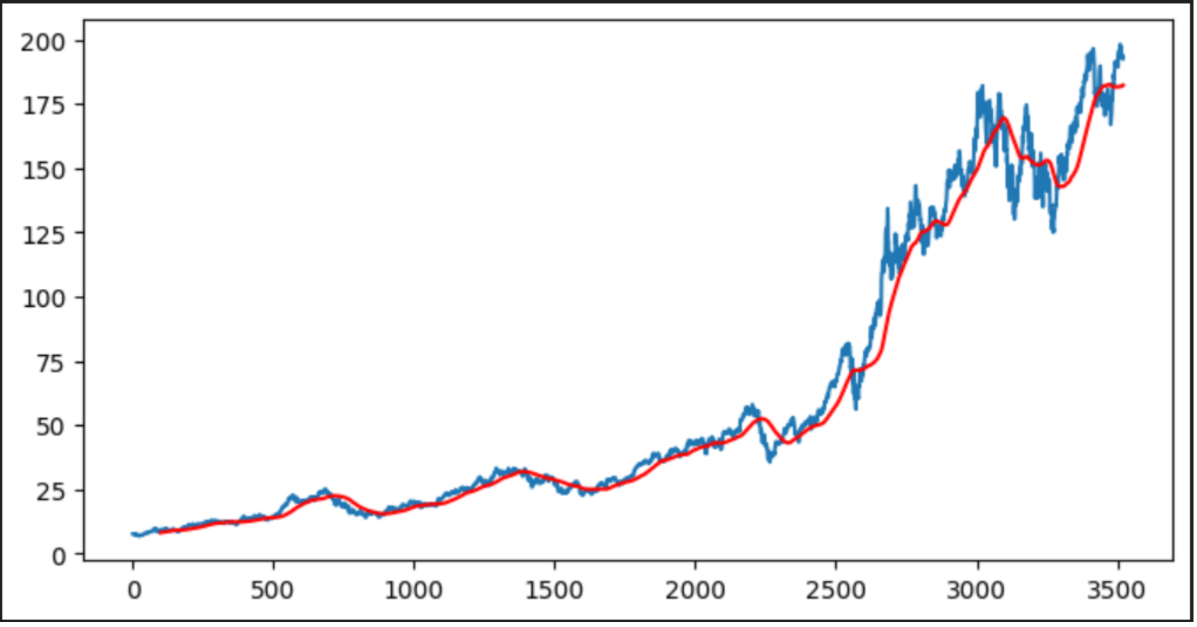
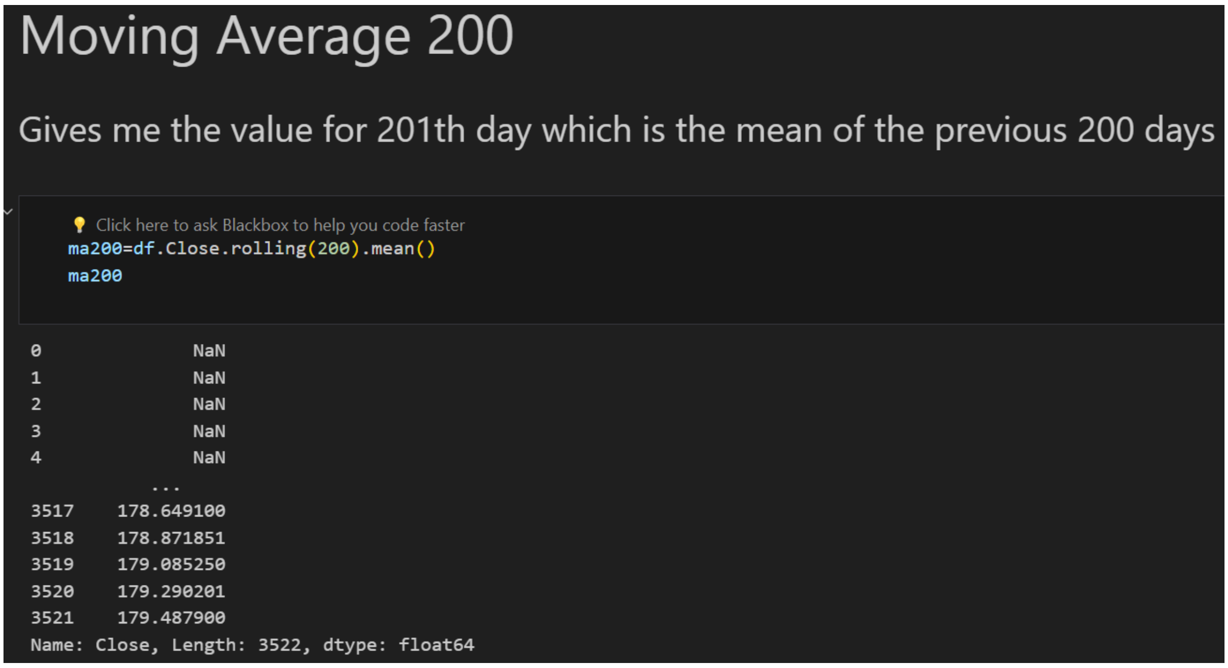


Figure 7 Graph of ma100 with Closing Value



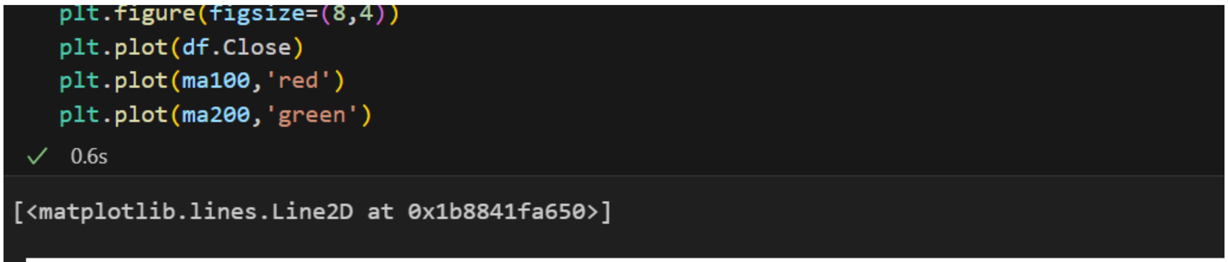
Moving Average 200

Gives me the value for 201th day which is the mean of the previous 200 days

```
ma200=df.Close.rolling(200).mean()
ma200
```

```
0              NaN
1              NaN
2              NaN
3              NaN
4              NaN
           ...
3517    178.649100
3518    178.871851
3519    179.085250
3520    179.290201
3521    179.487900
Name: Close, Length: 3522, dtype: float64
```

Figure 8 Moving Average 200



```
plt.figure(figsize=(8,4))
plt.plot(df.Close)
plt.plot(ma100,'red')
plt.plot(ma200,'green')
```

```
[<matplotlib.lines.Line2D at 0x1b8841fa650>]
```

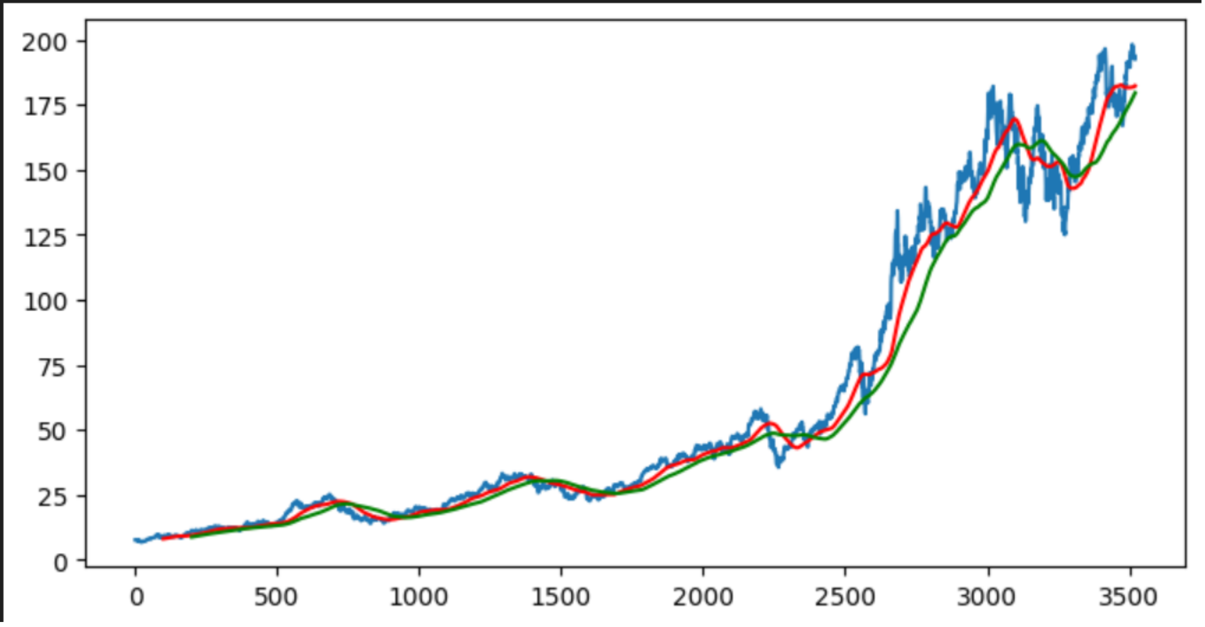
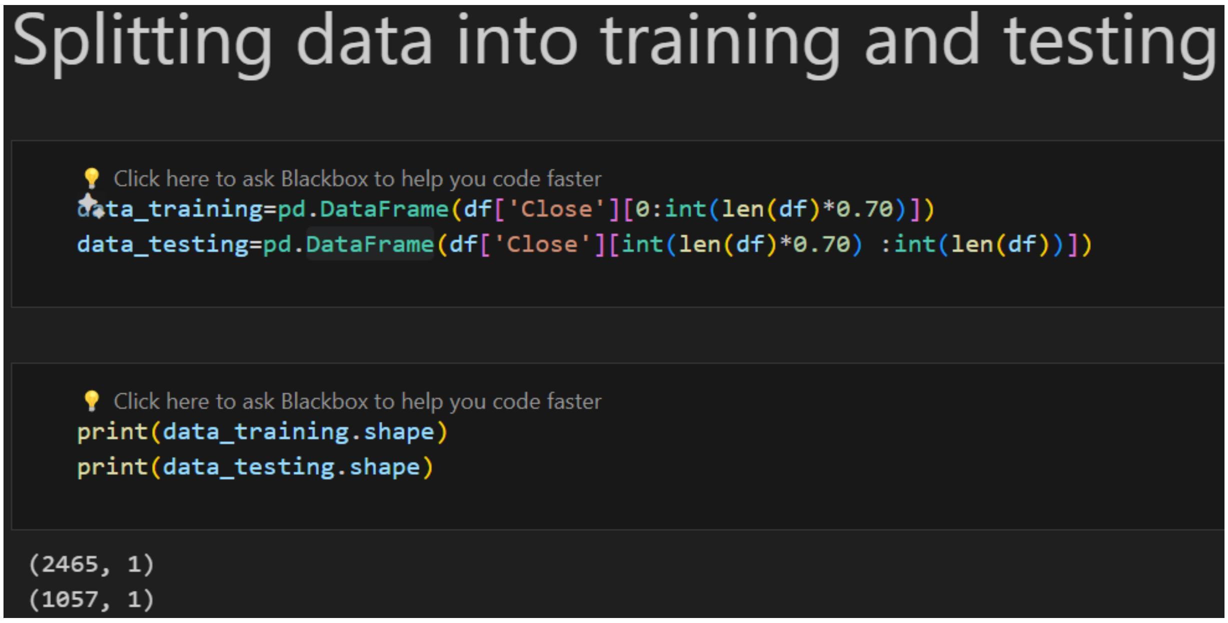


Figure 9 Graph of ma100 and ma200 with Closing Value



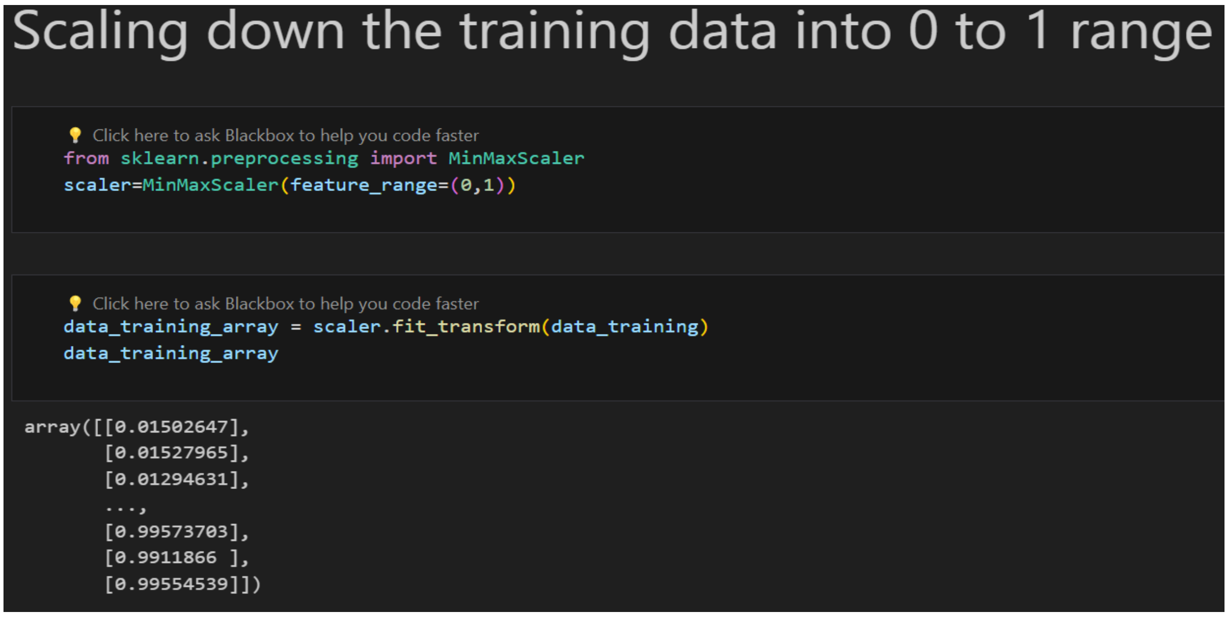
Splitting data into training and testing

💡 Click here to ask Blackbox to help you code faster
```python
data_training=pd.DataFrame(df['Close'][0:int(len(df)*0.70)])
data_testing=pd.DataFrame(df['Close'][int(len(df)*0.70) :int(len(df))])
```

💡 Click here to ask Blackbox to help you code faster
```python
print(data_training.shape)
print(data_testing.shape)
```

```
(2465, 1)
(1057, 1)
```

Figure 10 Data Splitting



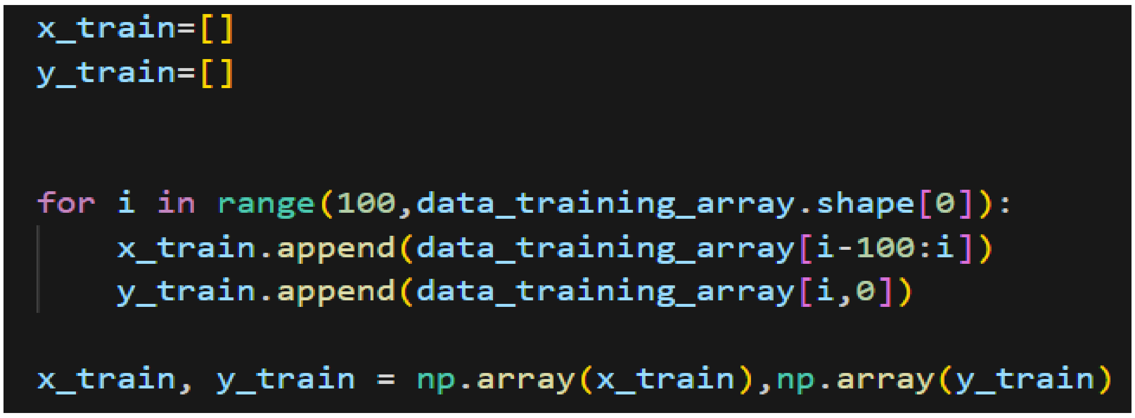
Scaling down the training data into 0 to 1 range

💡 Click here to ask Blackbox to help you code faster
```python
from sklearn.preprocessing import MinMaxScaler
scaler=MinMaxScaler(feature_range=(0,1))
```

💡 Click here to ask Blackbox to help you code faster
```python
data_training_array = scaler.fit_transform(data_training)
data_training_array
```

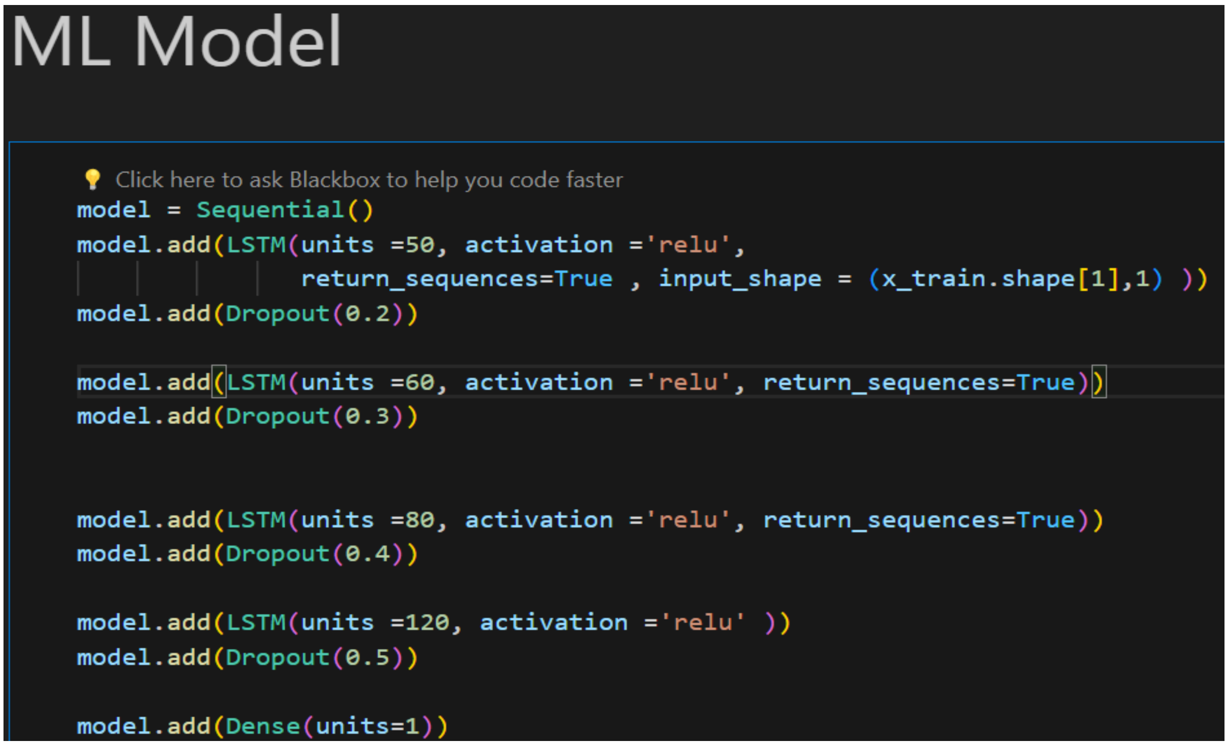
```
array([[0.01502647],
       [0.01527965],
       [0.01294631],
       ...,
       [0.99573703],
       [0.9911866 ],
       [0.99554539]])
```

Figure 11 Scaling Down Data



```python
x_train=[]
y_train=[]


for i in range(100,data_training_array.shape[0]):
    x_train.append(data_training_array[i-100:i])
    y_train.append(data_training_array[i,0])

x_train, y_train = np.array(x_train),np.array(y_train)
```

Figure 12 Making x_train and y_train

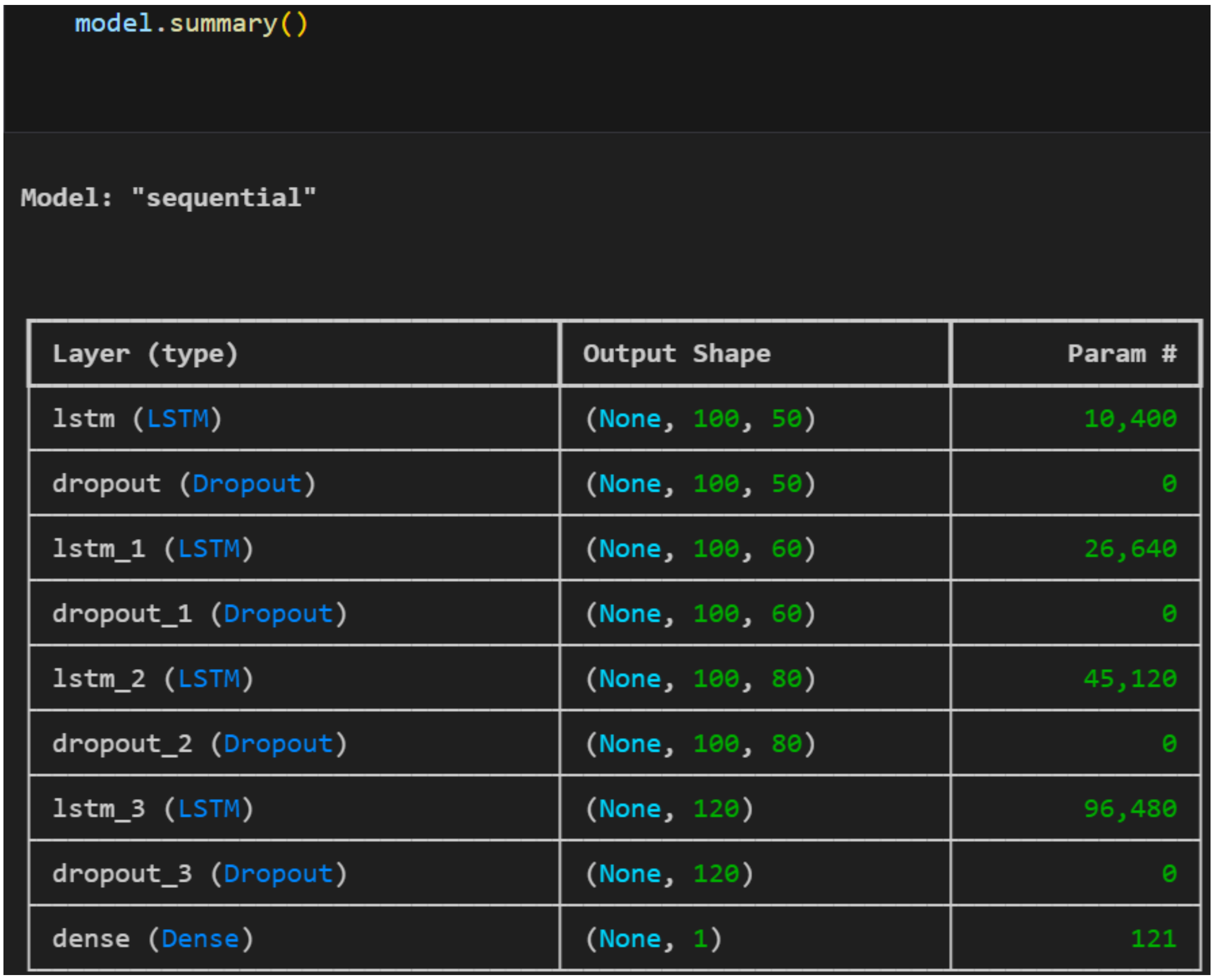


ML Model

Click here to ask Blackbox to help you code faster

```
model = Sequential()
model.add(LSTM(units =50, activation ='relu',
               return_sequences=True , input_shape = (x_train.shape[1],1) ))
model.add(Dropout(0.2))

model.add(LSTM(units =60, activation ='relu', return_sequences=True))
model.add(Dropout(0.3))


model.add(LSTM(units =80, activation ='relu', return_sequences=True))
model.add(Dropout(0.4))

model.add(LSTM(units =120, activation ='relu' ))
model.add(Dropout(0.5))

model.add(Dense(units=1))
```

Figure 13 Making the Model

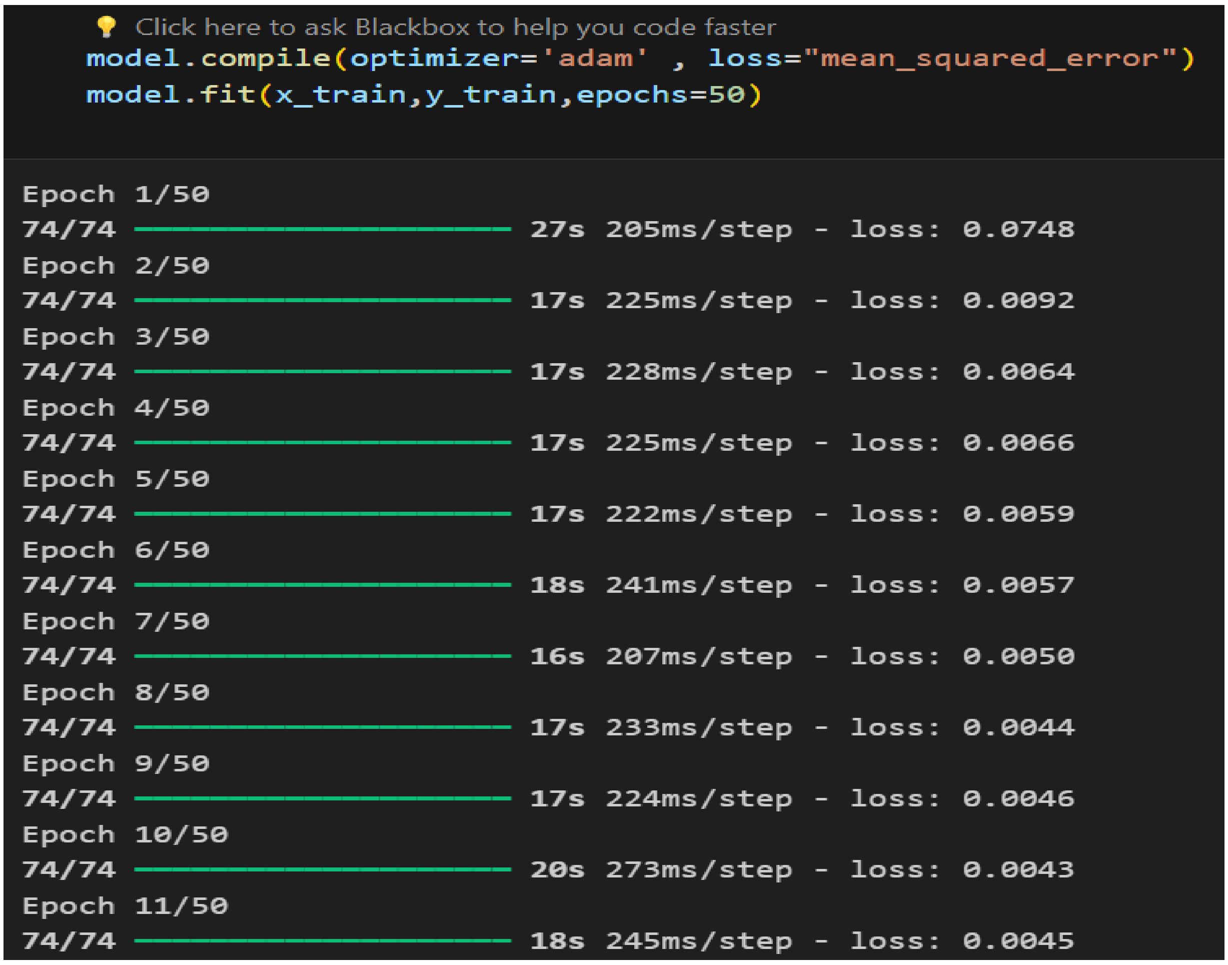


```
model.summary()
```

```
Model: "sequential"
```

Layer (type)	Output Shape	Param #
lstm (LSTM)	(None, 100, 50)	10,400
dropout (Dropout)	(None, 100, 50)	0
lstm_1 (LSTM)	(None, 100, 60)	26,640
dropout_1 (Dropout)	(None, 100, 60)	0
lstm_2 (LSTM)	(None, 100, 80)	45,120
dropout_2 (Dropout)	(None, 100, 80)	0
lstm_3 (LSTM)	(None, 120)	96,480
dropout_3 (Dropout)	(None, 120)	0
dense (Dense)	(None, 1)	121

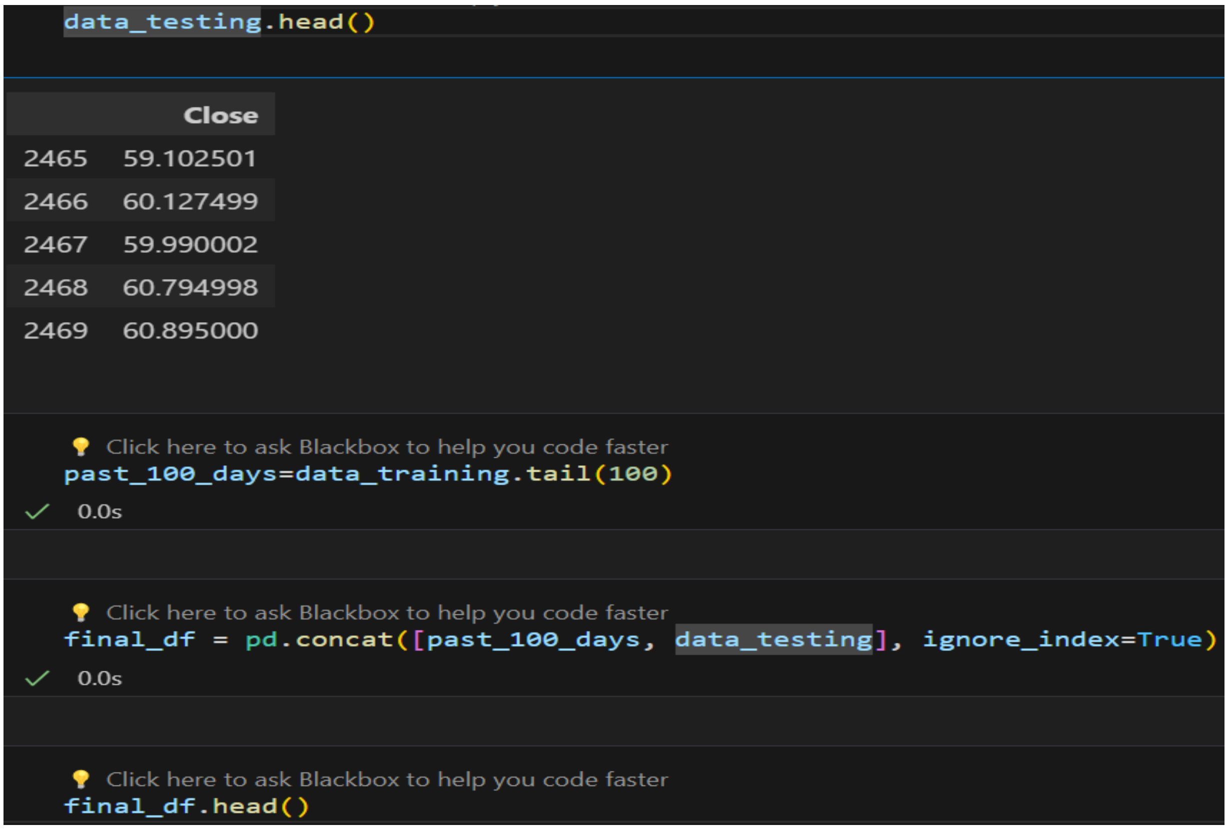
Figure 14 Model Summary



```
💡 Click here to ask Blackbox to help you code faster
model.compile(optimizer='adam' , loss="mean_squared_error")
model.fit(x_train,y_train,epochs=50)
```

```
Epoch 1/50
74/74 ━━━━━━━━━━━━━━━━━━━━ 27s 205ms/step - loss: 0.0748
Epoch 2/50
74/74 ━━━━━━━━━━━━━━━━━━━━ 17s 225ms/step - loss: 0.0092
Epoch 3/50
74/74 ━━━━━━━━━━━━━━━━━━━━ 17s 228ms/step - loss: 0.0064
Epoch 4/50
74/74 ━━━━━━━━━━━━━━━━━━━━ 17s 225ms/step - loss: 0.0066
Epoch 5/50
74/74 ━━━━━━━━━━━━━━━━━━━━ 17s 222ms/step - loss: 0.0059
Epoch 6/50
74/74 ━━━━━━━━━━━━━━━━━━━━ 18s 241ms/step - loss: 0.0057
Epoch 7/50
74/74 ━━━━━━━━━━━━━━━━━━━━ 16s 207ms/step - loss: 0.0050
Epoch 8/50
74/74 ━━━━━━━━━━━━━━━━━━━━ 17s 233ms/step - loss: 0.0044
Epoch 9/50
74/74 ━━━━━━━━━━━━━━━━━━━━ 17s 224ms/step - loss: 0.0046
Epoch 10/50
74/74 ━━━━━━━━━━━━━━━━━━━━ 20s 273ms/step - loss: 0.0043
Epoch 11/50
74/74 ━━━━━━━━━━━━━━━━━━━━ 18s 245ms/step - loss: 0.0045
```

Figure 15 Model Compilation



```
data_testing.head()
```

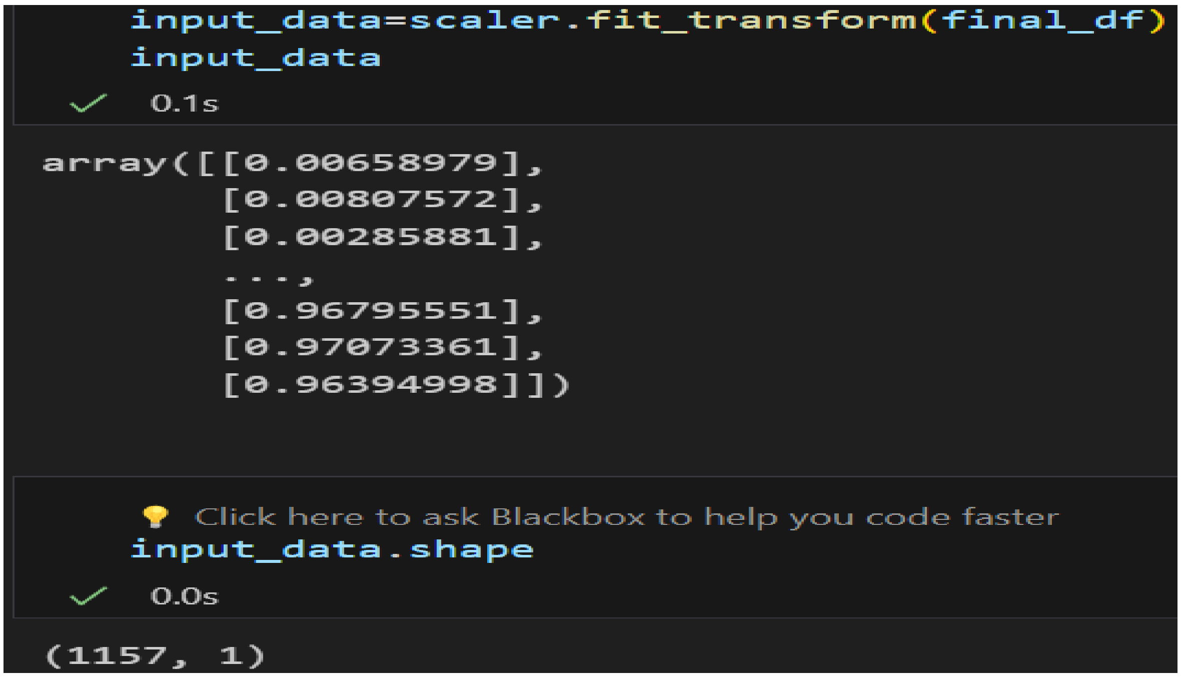
	Close
2465	59.102501
2466	60.127499
2467	59.990002
2468	60.794998
2469	60.895000

```
💡 Click here to ask Blackbox to help you code faster
past_100_days=data_training.tail(100)
✓ 0.0s
```

```
💡 Click here to ask Blackbox to help you code faster
final_df = pd.concat([past_100_days, data_testing], ignore_index=True)
✓ 0.0s
```

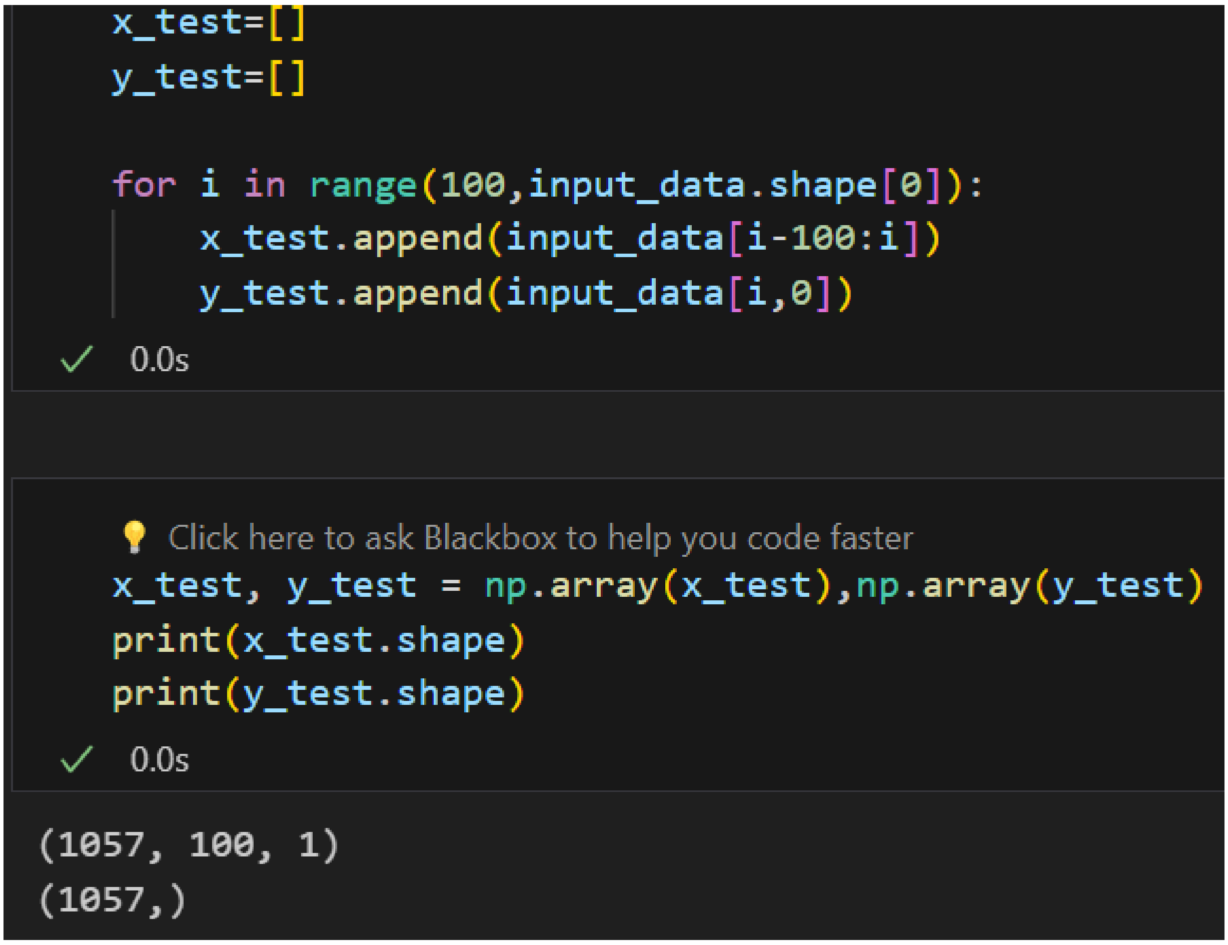
```
💡 Click here to ask Blackbox to help you code faster
final_df.head()
```

Figure 16



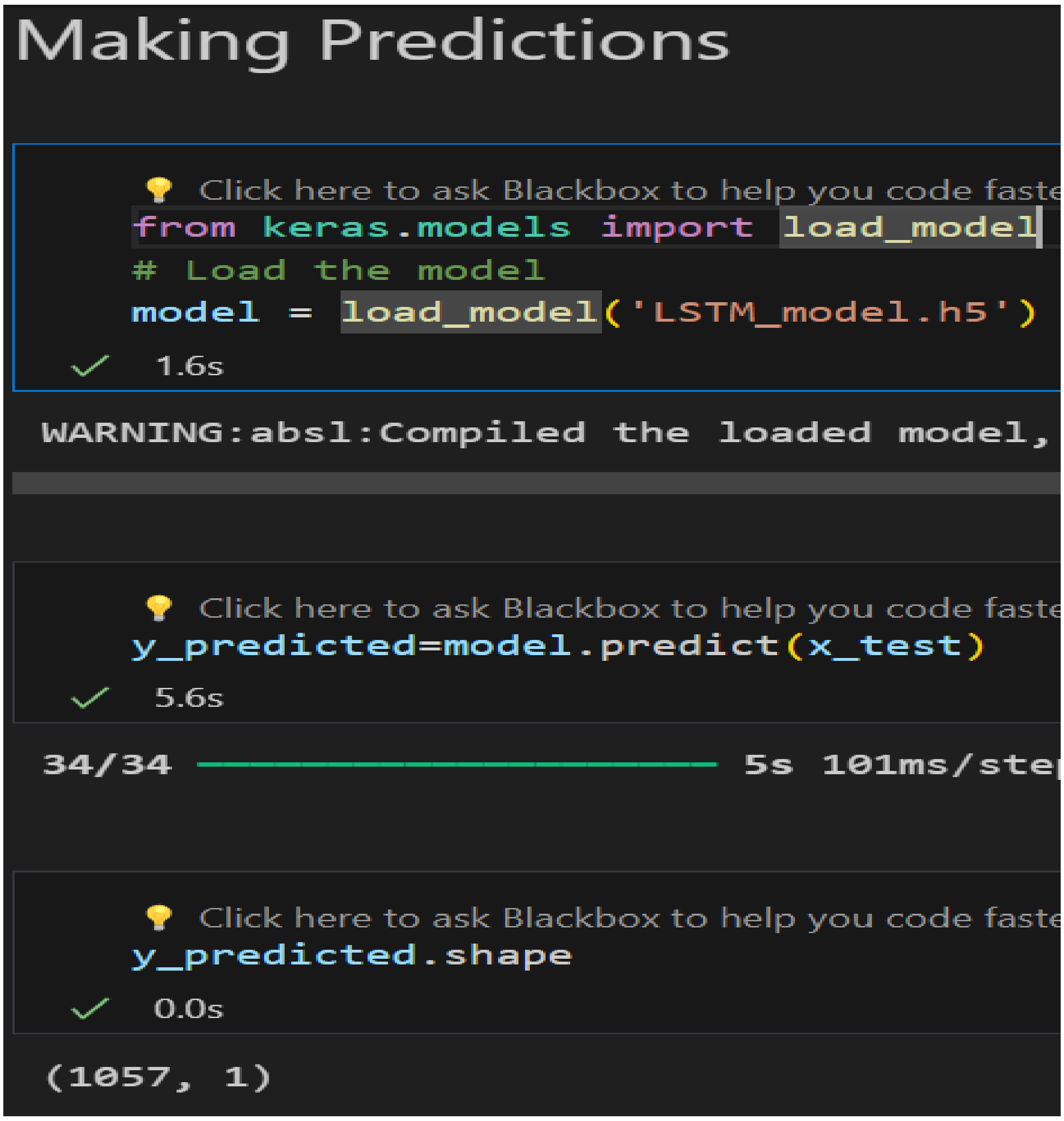
```
input_data=scaler.fit_transform(final_df)
input_data
✓ 0.1s

array([[0.00658979],
       [0.00807572],
       [0.00285881],
       ...,
       [0.96795551],
       [0.97073361],
       [0.96394998]])

💡 Click here to ask Blackbox to help you code faster
input_data.shape
✓ 0.0s

(1157, 1)
```

Figure 17 Input Data



```
x_test=[]
y_test=[]

for i in range(100,input_data.shape[0]):
    x_test.append(input_data[i-100:i])
    y_test.append(input_data[i,0])
✓ 0.0s

💡 Click here to ask Blackbox to help you code faster
x_test, y_test = np.array(x_test),np.array(y_test)
print(x_test.shape)
print(y_test.shape)
✓ 0.0s

(1057, 100, 1)
(1057,)
```

Figure 18 x_test and y_test on Input Data



Making Predictions

```
💡 Click here to ask Blackbox to help you code faste
from keras.models import load_model
# Load the model
model = load_model('LSTM_model.h5')
✓ 1.6s
```

```
WARNING:absl:Compiled the loaded model,
```

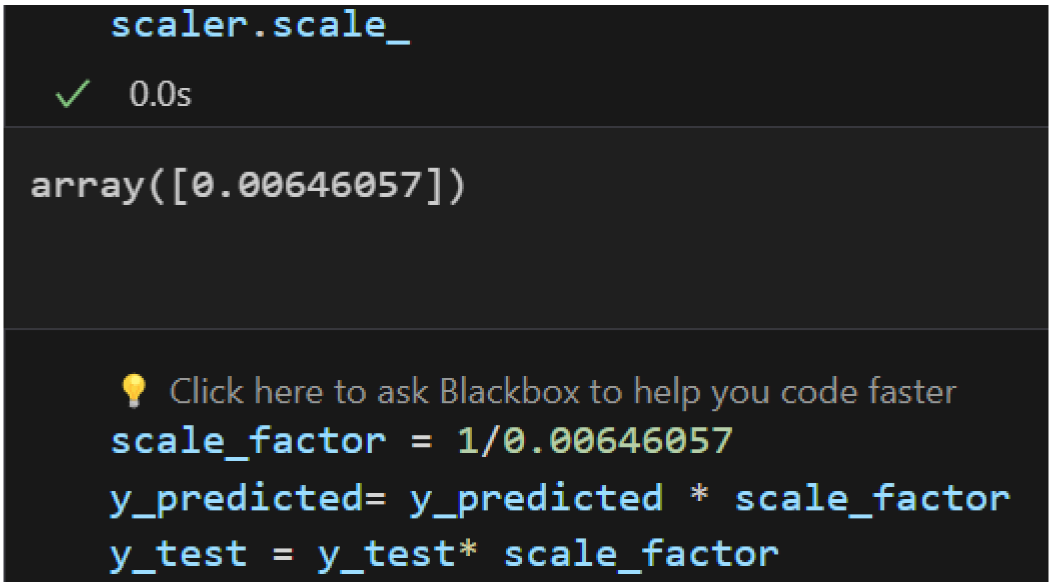
```
💡 Click here to ask Blackbox to help you code faste
y_predicted=model.predict(x_test)
✓ 5.6s
```

```
34/34 ━━━━━━━━━━━━━━━━━━━━ 5s 101ms/ste
```

```
💡 Click here to ask Blackbox to help you code faste
y_predicted.shape
✓ 0.0s
```

```
(1057, 1)
```

Figure 19 Making Predictions

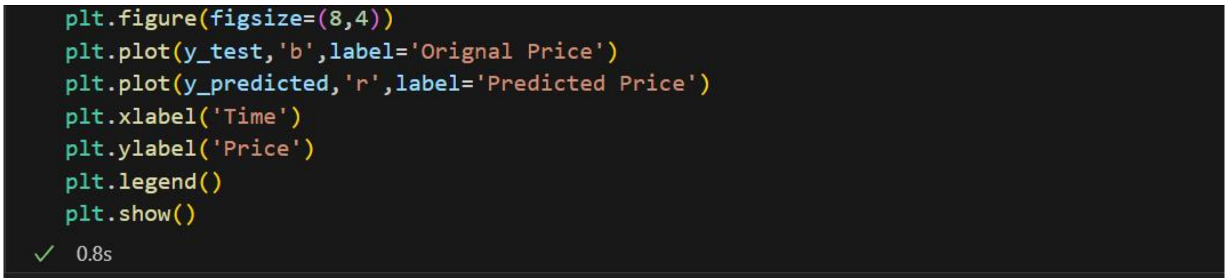


```
scaler.scale_
✓ 0.0s
```

```
array([0.00646057])
```

```
💡 Click here to ask Blackbox to help you code faster
scale_factor = 1/0.00646057
y_predicted= y_predicted * scale_factor
y_test = y_test* scale_factor
```

Figure 20 Scale Factor



```
plt.figure(figsize=(8,4))
plt.plot(y_test,'b',label='Orignal Price')
plt.plot(y_predicted,'r',label='Predicted Price')
plt.xlabel('Time')
plt.ylabel('Price')
plt.legend()
plt.show()
```
✓ 0.8s

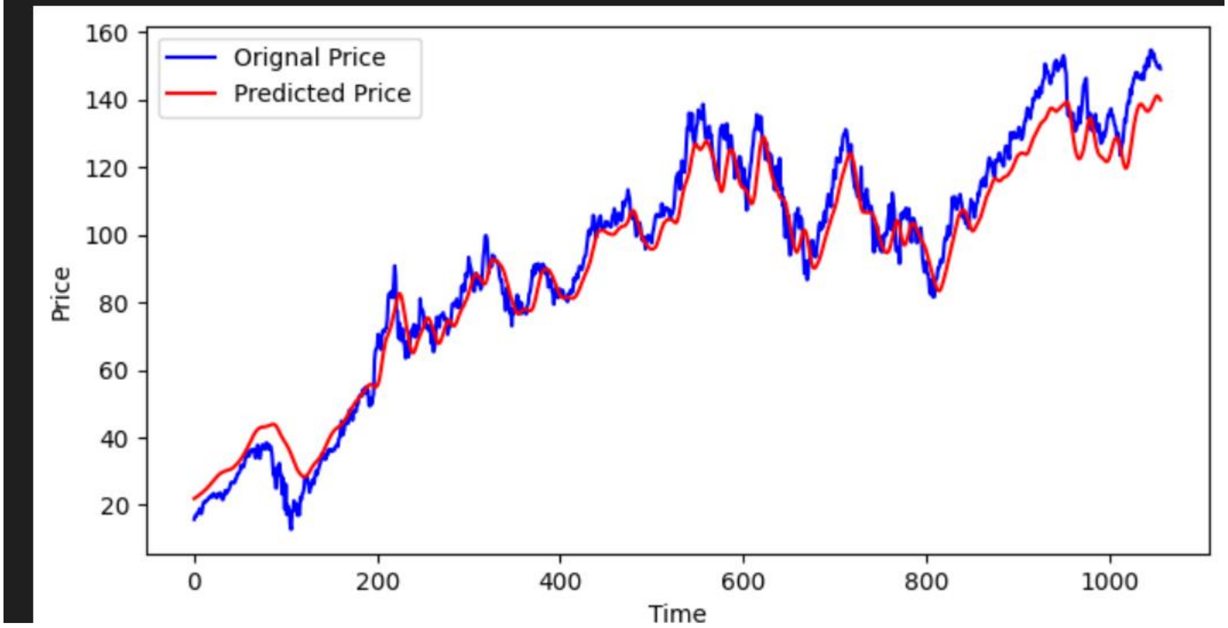
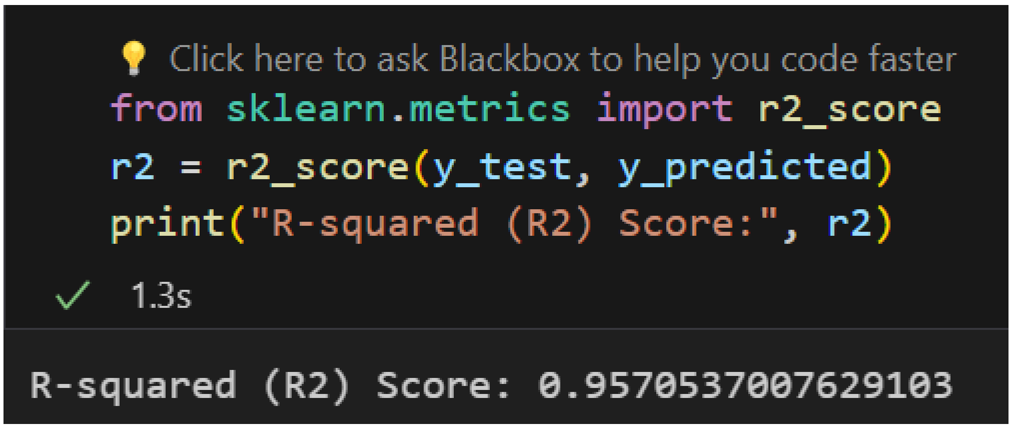


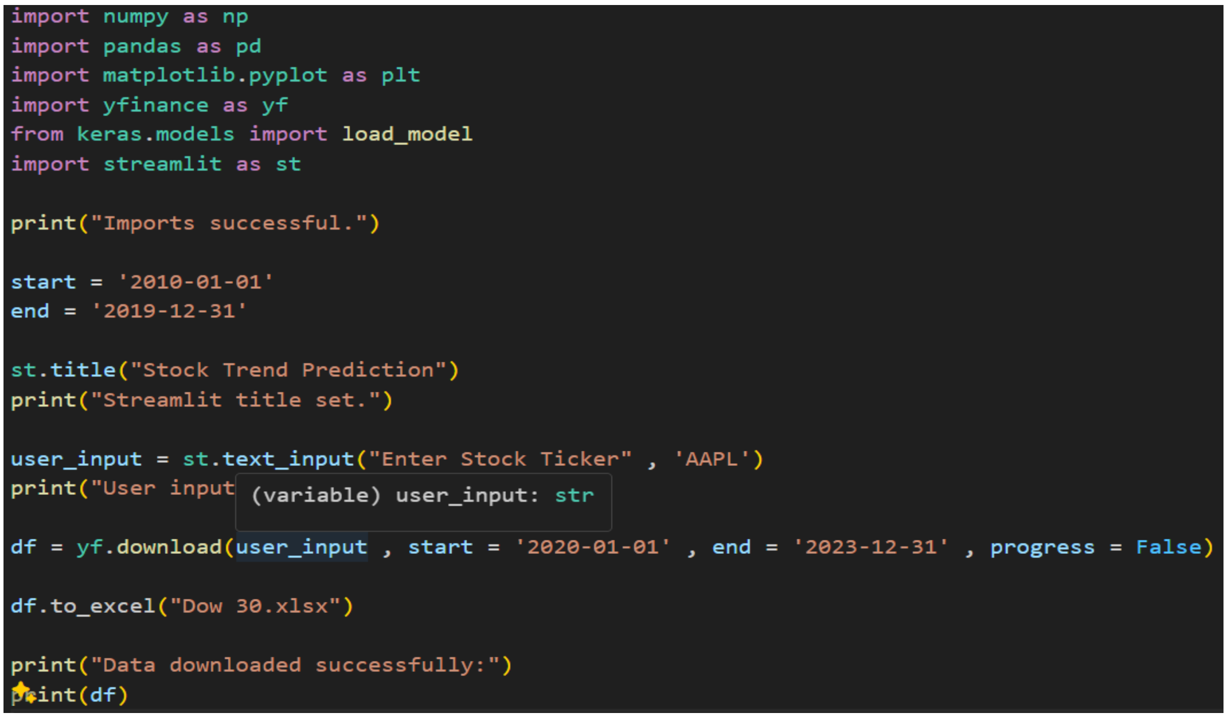
Figure 21 Graph Original Vs Predicted



Click here to ask Blackbox to help you code faster
```
from sklearn.metrics import r2_score
r2 = r2_score(y_test, y_predicted)
print("R-squared (R2) Score:", r2)
```
✓ 1.3s

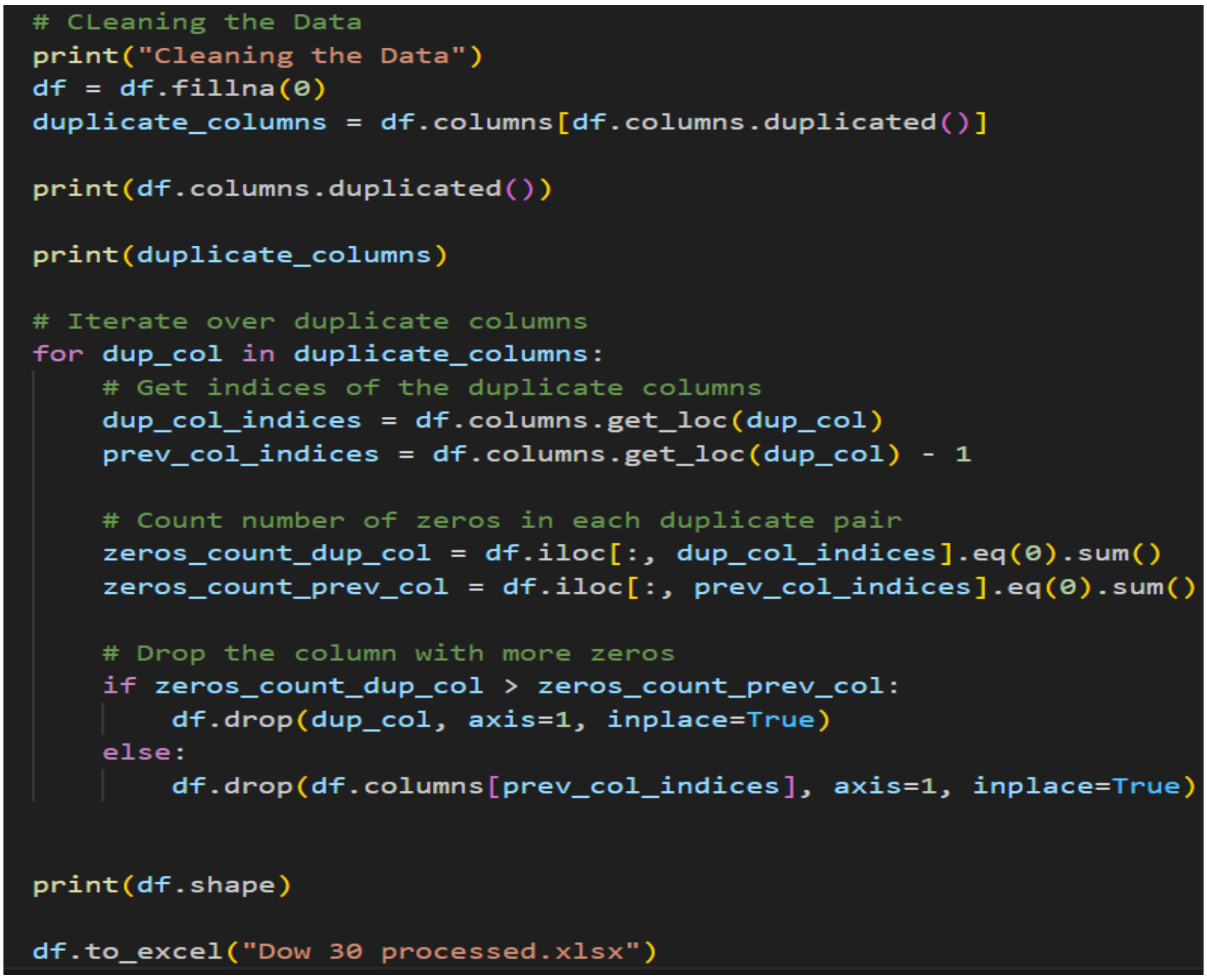
```
R-squared (R2) Score: 0.9570537007629103
```
Figure 22 Accuracy Check

Streamlit App



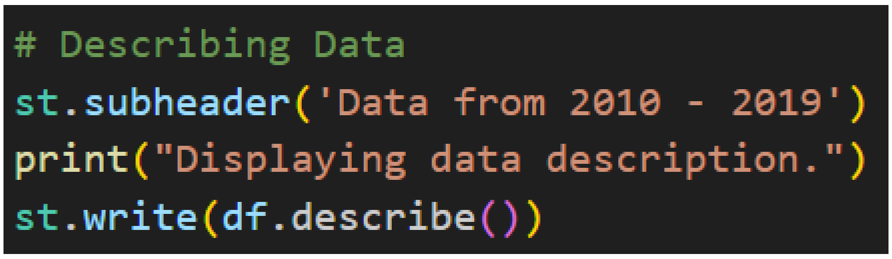
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import yfinance as yf
from keras.models import load_model
import streamlit as st

print("Imports successful.")

start = '2010-01-01'
end = '2019-12-31'

st.title("Stock Trend Prediction")
print("Streamlit title set.")

user_input = st.text_input("Enter Stock Ticker" , 'AAPL')
print("User input (variable) user_input: str

df = yf.download(user_input , start = '2020-01-01' , end = '2023-12-31' , progress = False)

df.to_excel("Dow 30.xlsx")

print("Data downloaded successfully:")
print(df)
```

Figure 23



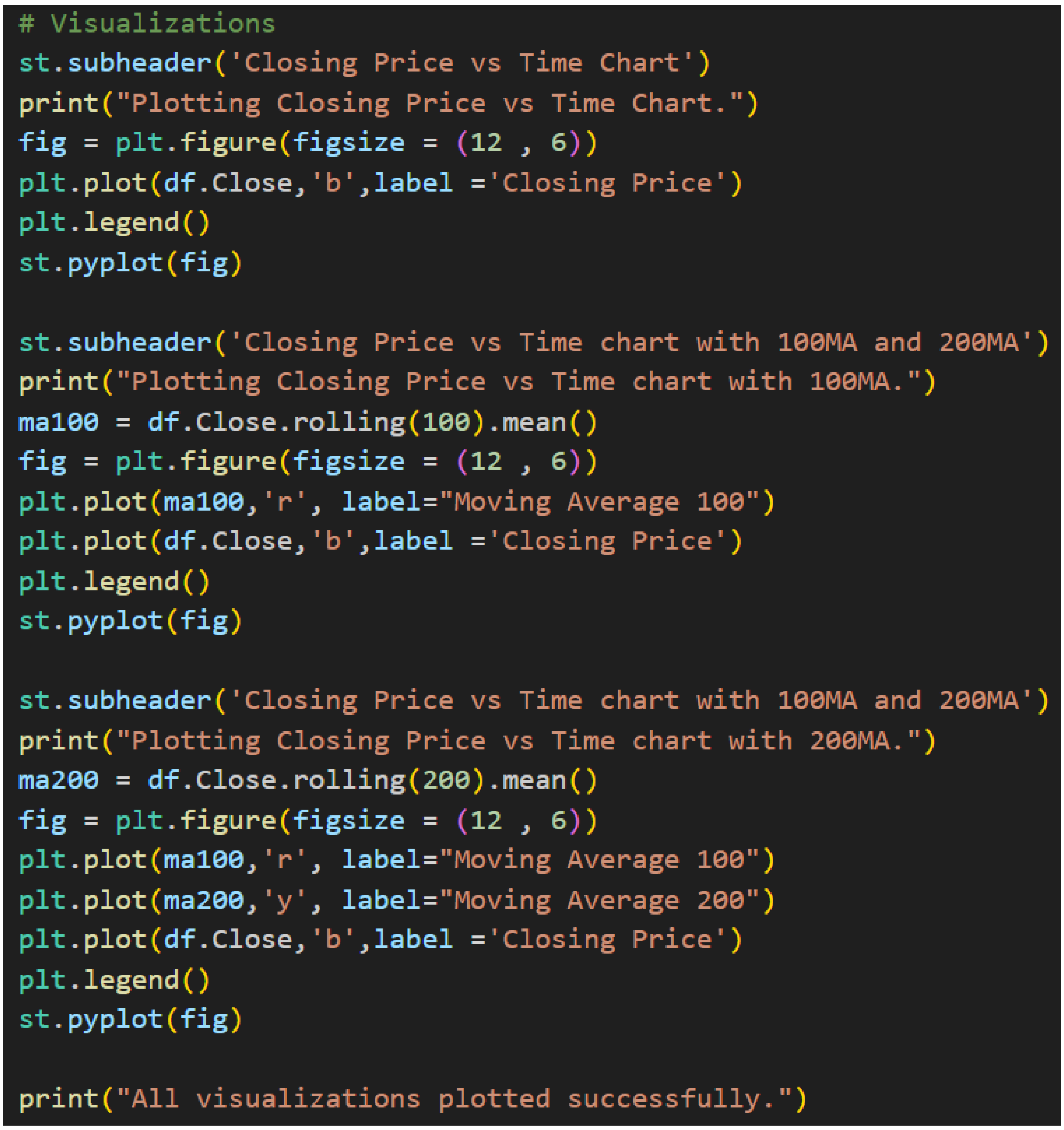
```
# CLeaning the Data
print("Cleaning the Data")
df = df.fillna(0)
duplicate_columns = df.columns[df.columns.duplicated()]

print(df.columns.duplicated())

print(duplicate_columns)

# Iterate over duplicate columns
for dup_col in duplicate_columns:
    # Get indices of the duplicate columns
    dup_col_indices = df.columns.get_loc(dup_col)
    prev_col_indices = df.columns.get_loc(dup_col) - 1

    # Count number of zeros in each duplicate pair
    zeros_count_dup_col = df.iloc[:, dup_col_indices].eq(0).sum()
    zeros_count_prev_col = df.iloc[:, prev_col_indices].eq(0).sum()

    # Drop the column with more zeros
    if zeros_count_dup_col > zeros_count_prev_col:
        df.drop(dup_col, axis=1, inplace=True)
    else:
        df.drop(df.columns[prev_col_indices], axis=1, inplace=True)


print(df.shape)

df.to_excel("Dow 30 processed.xlsx")
```

Figure 24 Data Cleaning



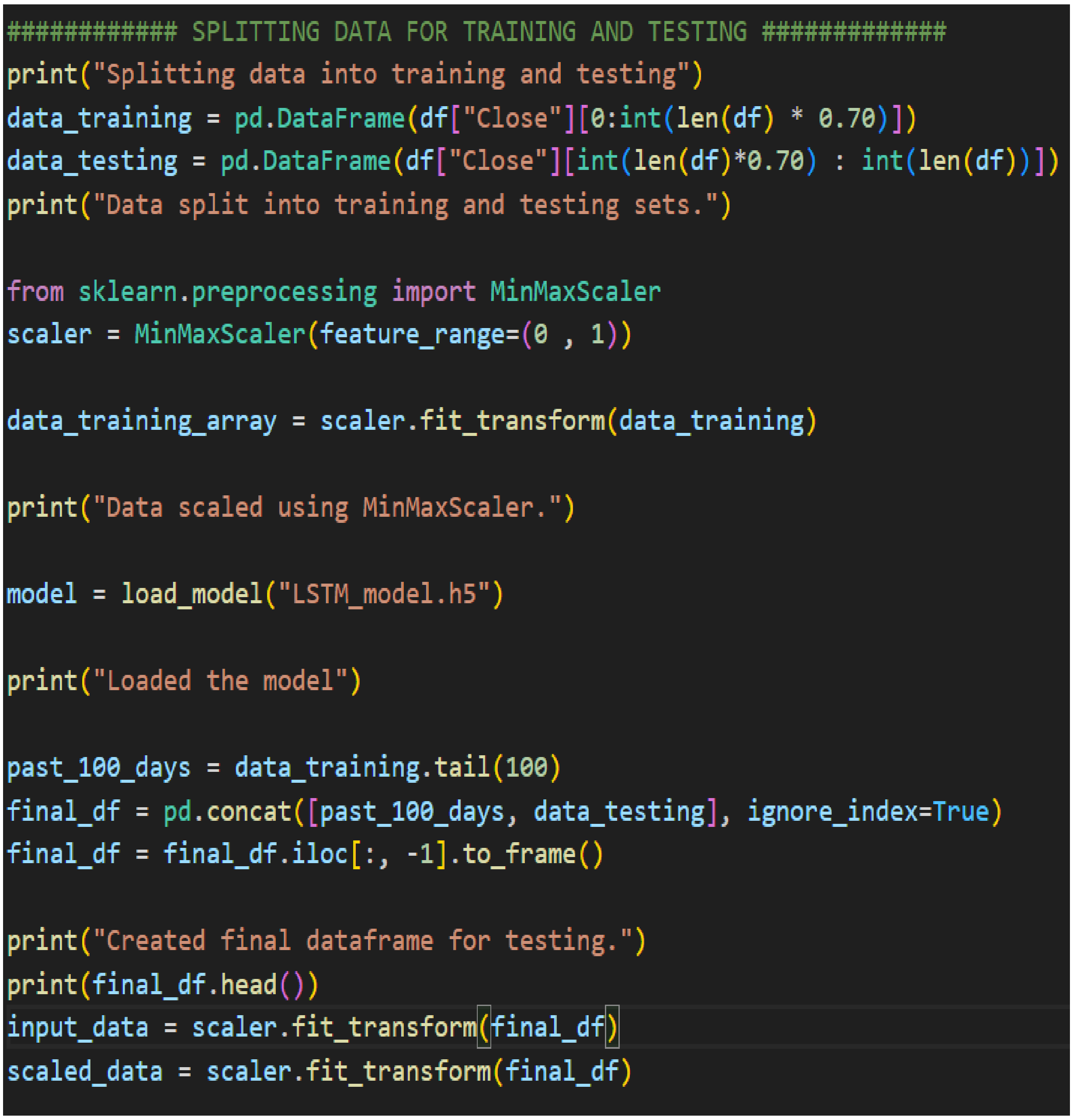
```python
# Describing Data
st.subheader('Data from 2010 - 2019')
print("Displaying data description.")
st.write(df.describe())
```

Figure 25 Describing Data



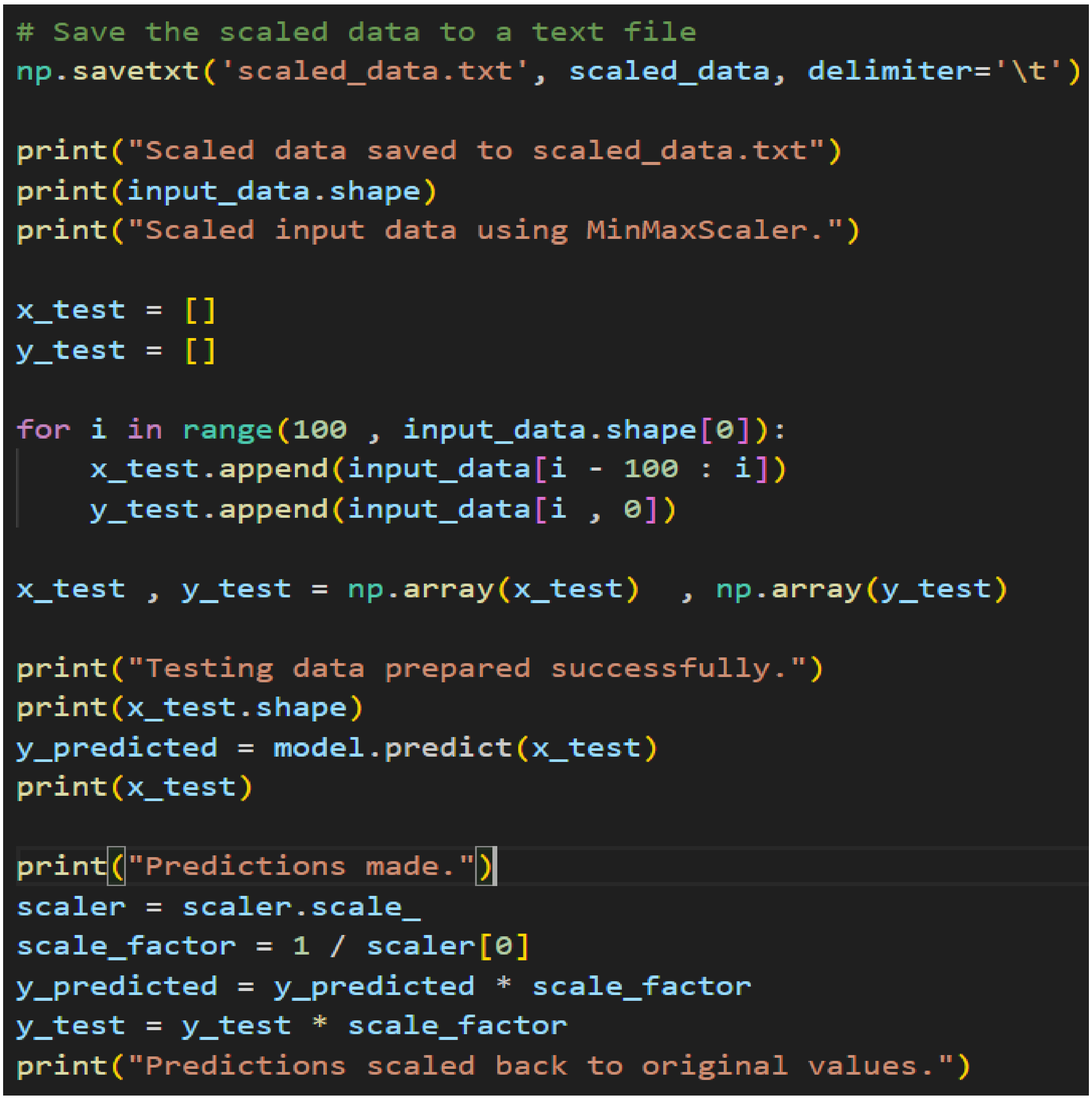
```python
# Visualizations
st.subheader('Closing Price vs Time Chart')
print("Plotting Closing Price vs Time Chart.")
fig = plt.figure(figsize = (12 , 6))
plt.plot(df.Close,'b',label ='Closing Price')
plt.legend()
st.pyplot(fig)

st.subheader('Closing Price vs Time chart with 100MA and 200MA')
print("Plotting Closing Price vs Time chart with 100MA.")
ma100 = df.Close.rolling(100).mean()
fig = plt.figure(figsize = (12 , 6))
plt.plot(ma100,'r', label="Moving Average 100")
plt.plot(df.Close,'b',label ='Closing Price')
plt.legend()
st.pyplot(fig)

st.subheader('Closing Price vs Time chart with 100MA and 200MA')
print("Plotting Closing Price vs Time chart with 200MA.")
ma200 = df.Close.rolling(200).mean()
fig = plt.figure(figsize = (12 , 6))
plt.plot(ma100,'r', label="Moving Average 100")
plt.plot(ma200,'y', label="Moving Average 200")
plt.plot(df.Close,'b',label ='Closing Price')
plt.legend()
st.pyplot(fig)

print("All visualizations plotted successfully.")
```

Figure 26 Visualizations



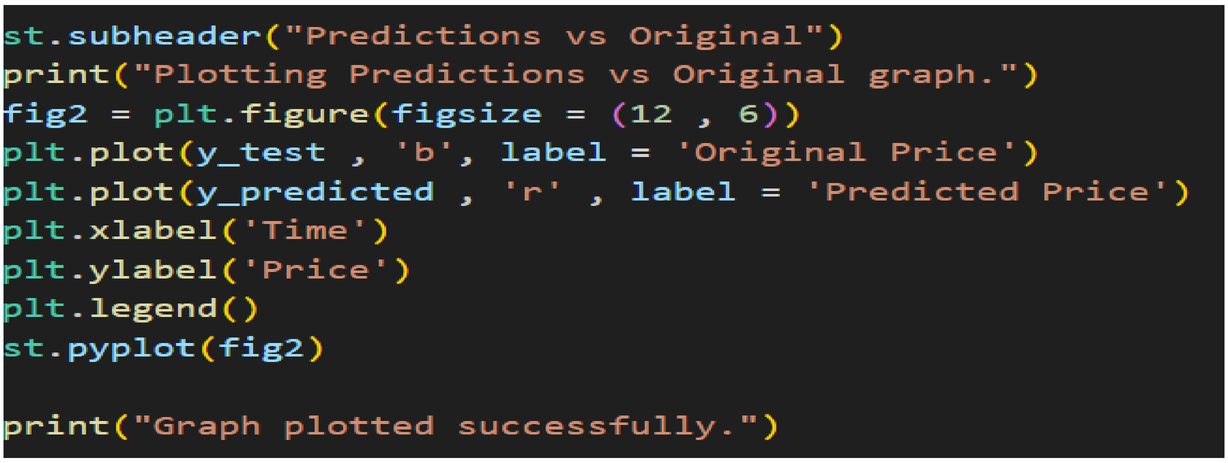
```
############ SPLITTING DATA FOR TRAINING AND TESTING #############
print("Splitting data into training and testing")
data_training = pd.DataFrame(df["Close"][0:int(len(df) * 0.70)])
data_testing = pd.DataFrame(df["Close"][int(len(df)*0.70) : int(len(df))])
print("Data split into training and testing sets.")

from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler(feature_range=(0 , 1))

data_training_array = scaler.fit_transform(data_training)

print("Data scaled using MinMaxScaler.")

model = load_model("LSTM_model.h5")

print("Loaded the model")

past_100_days = data_training.tail(100)
final_df = pd.concat([past_100_days, data_testing], ignore_index=True)
final_df = final_df.iloc[:, -1].to_frame()

print("Created final dataframe for testing.")
print(final_df.head())
input_data = scaler.fit_transform(final_df)
scaled_data = scaler.fit_transform(final_df)
```

Figure 27 Splitting Data



```python
# Save the scaled data to a text file
np.savetxt('scaled_data.txt', scaled_data, delimiter='\t')

print("Scaled data saved to scaled_data.txt")
print(input_data.shape)
print("Scaled input data using MinMaxScaler.")

x_test = []
y_test = []

for i in range(100 , input_data.shape[0]):
    x_test.append(input_data[i - 100 : i])
    y_test.append(input_data[i , 0])

x_test , y_test = np.array(x_test)  , np.array(y_test)

print("Testing data prepared successfully.")
print(x_test.shape)
y_predicted = model.predict(x_test)
print(x_test)

print("Predictions made.")
scaler = scaler.scale_
scale_factor = 1 / scaler[0]
y_predicted = y_predicted * scale_factor
y_test = y_test * scale_factor
print("Predictions scaled back to original values.")
```

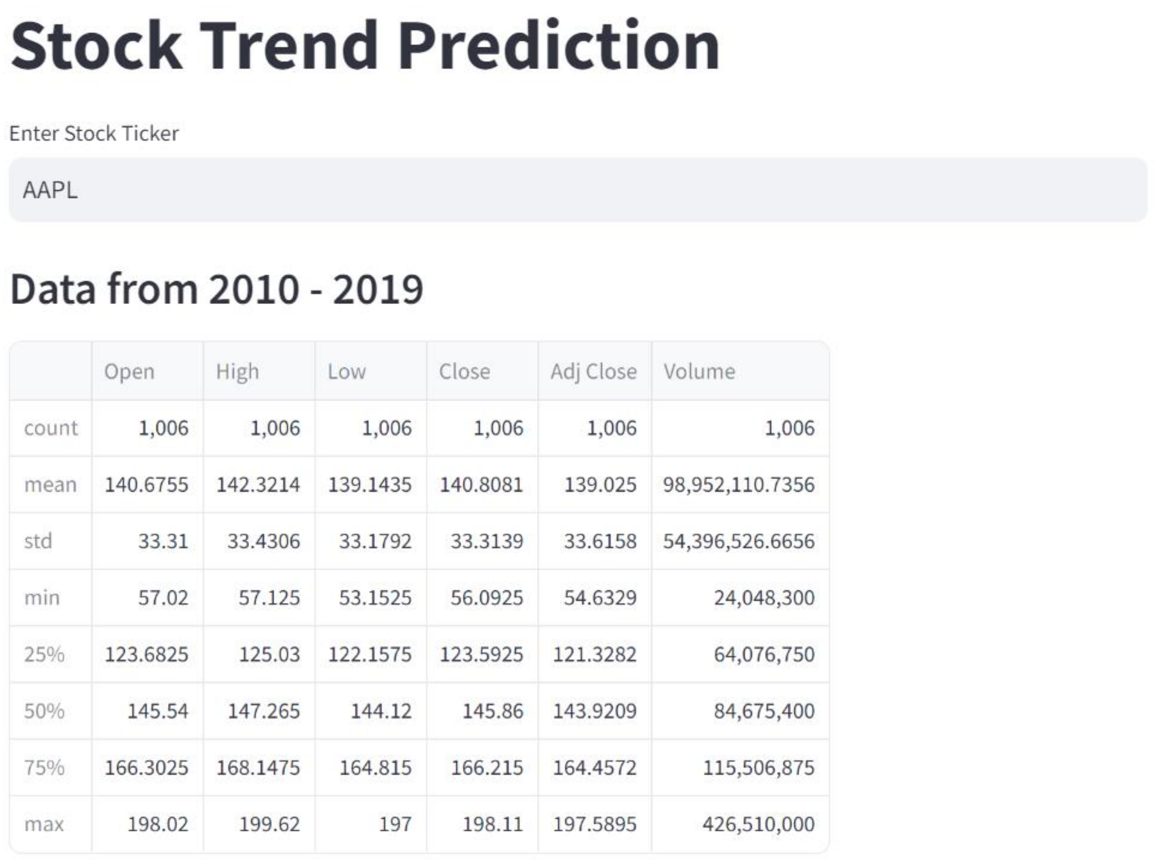
Figure 28 Scaled Data



```python
st.subheader("Predictions vs Original")
print("Plotting Predictions vs Original graph.")
fig2 = plt.figure(figsize = (12 , 6))
plt.plot(y_test , 'b', label = 'Original Price')
plt.plot(y_predicted , 'r' , label = 'Predicted Price')
plt.xlabel('Time')
plt.ylabel('Price')
plt.legend()
st.pyplot(fig2)

print("Graph plotted successfully.")
```

Figure 29 Final Graph

<u>WebApp</u>



Stock Trend Prediction

Enter Stock Ticker

AAPL

Data from 2010 - 2019

	Open	High	Low	Close	Adj Close	Volume
count	1,006	1,006	1,006	1,006	1,006	1,006
mean	140.6755	142.3214	139.1435	140.8081	139.025	98,952,110.7356
std	33.31	33.4306	33.1792	33.3139	33.6158	54,396,526.6656
min	57.02	57.125	53.1525	56.0925	54.6329	24,048,300
25%	123.6825	125.03	122.1575	123.5925	121.3282	64,076,750
50%	145.54	147.265	144.12	145.86	143.9209	84,675,400
75%	166.3025	168.1475	164.815	166.215	164.4572	115,506,875
max	198.02	199.62	197	198.11	197.5895	426,510,000

Figure 30

Closing Price vs Time Chart

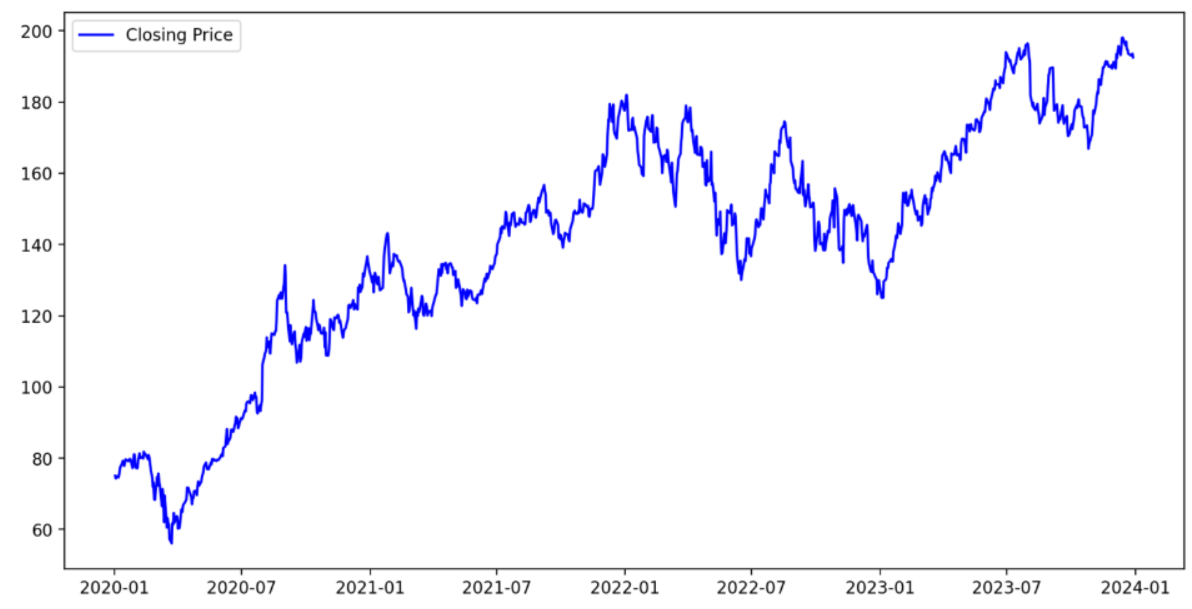


Figure 31

Closing Price vs Time chart with 100MA

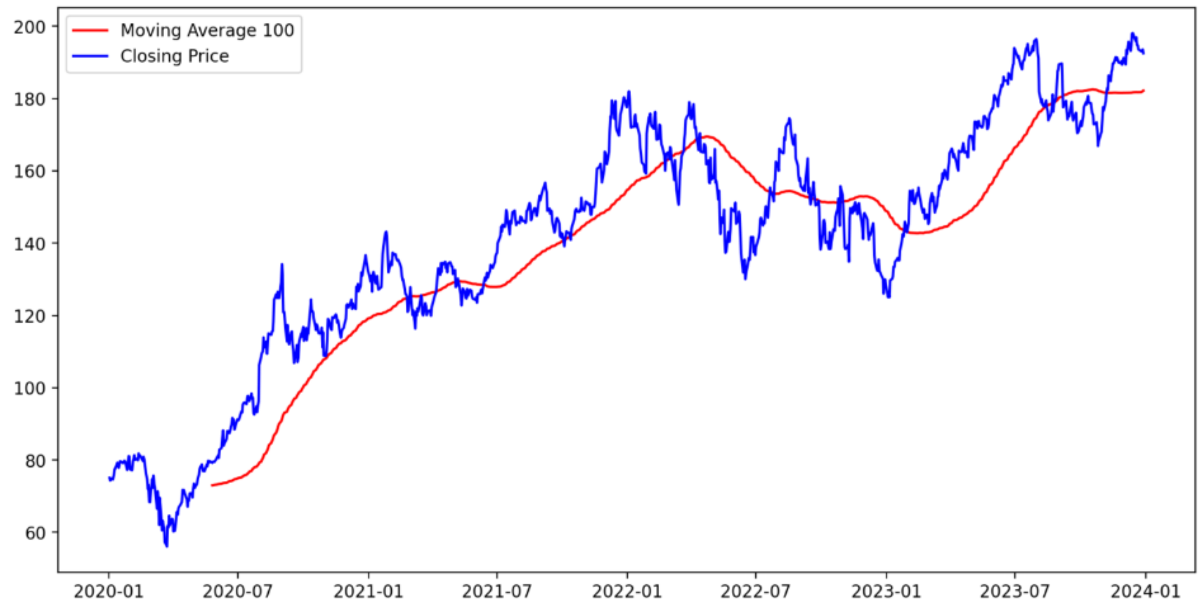


Figure 32

Closing Price vs Time chart with 100MA and 200MA

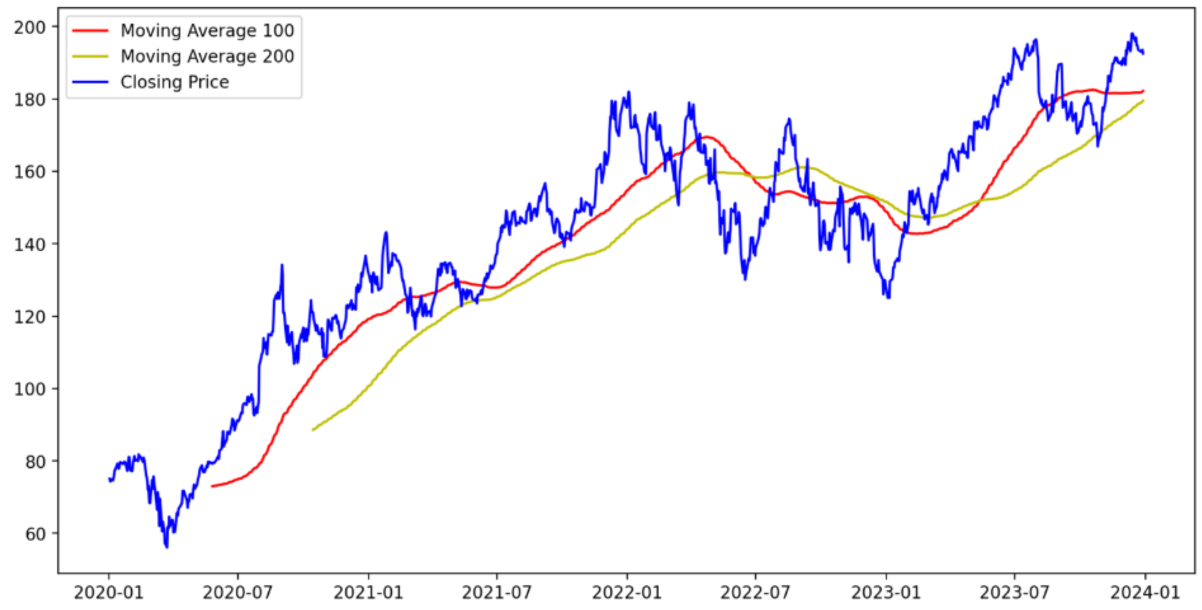


Figure 33

Predictions vs Original

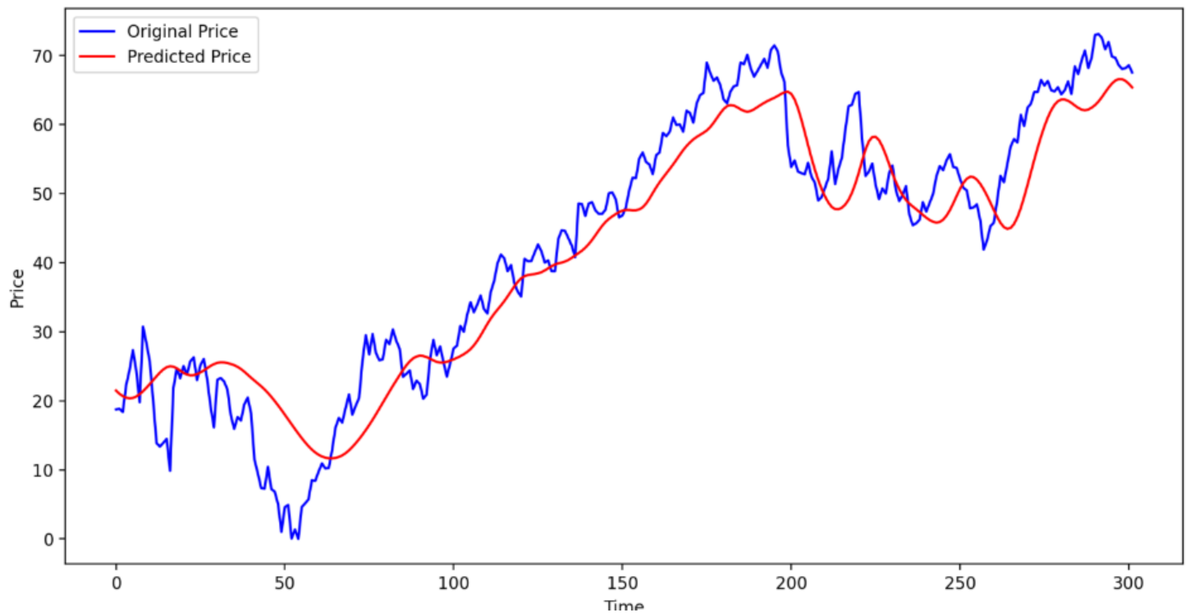


Figure 34

References

1. https://www.kaggle.com/
2. https://finance.yahoo.com/
3. https://streamlit.io/