APPLIED DATA SCIENCE –PHASE 5

STOCK PRICE PREDICTION

PROBLEM STATEMENT:

The problem is to build a predictive model that forecasts stock prices based on historical market data. The goal is to create a tool that assists investors in making well-informed decisions and optimizing their investment strategies. This project involves data collection, data preprocessing, feature engineering, model selection, training, and evaluation.

As the problem definition of the project tells us that , the main motive of our model is to predict prices of a given stock and then predict its price . As this will help the investors in understanding the market using appropriate data points and thus giving them a clear vision in making the right and well informed decisions.

DESIGN THINKING: This is the phase where in how the problem is going to be approached is discussed

Data Collection: In this step which is the primary step for any problem statement is the collection of appropriate data which will help us in making proper predictions based on the relevant data points.

Here , we can take the historical data of the stocks of whom we want to predict the price for and accordingly we will have to take in the right columns that help in the prediction process which would be date, open price, close price, volume, are some of the indicators which can help us in the predicting the price.

Data Preprocessing: This is a very crucial part of the project because the accuracy of the model is as good as the data on which it feeds to give us the desired outcome that means if we clean the dataset i.e, removing unwanted data or filling up the null/nan values and then keeping only those factors that actually help in predicting the price is what is to be done in this step.

Feature Engineering: There are many factors that affect the price of any stock so we need to use different parameters that help in enhancing the power of prediction of the price like Lag features, moving averages, relative strength index(RSI) ,calendar features,time based features, or more such ideas can be inco-operated.

Model Selection: Here we choose the most appropriate or suitable algorithm like linear regression , but then ARIMA or SARIMA which are time series model or even Long short term memory networks(LSTM) can be made use of for predicting the price of the stock.

Model Training: This is the point where in the model that we have created is checked against the predefined dataset (70:30 i.e,70 % data for processing/training and 30%data for testing).

Evaluation: here the model is evaluated and this part tells us the model's performance using appropriate time series forecasting metrics (e.g., Mean Absolute Error, Root Mean Squared Error).

PHASES OF DEVELOPMENT

Developing a stock price prediction model typically involves several phases, each crucial for ensuring the accuracy and reliability of the model. Here are the common phases involved in building such a model:

1.Problem Definition and Data Collection:

- Clearly define the problem you want to solve.

- Gather historical stock market data, including price, volume, and other relevant factors, from various reliable sources like financial APIs, databases, or online repositories. Here we have taken the dataset from kaggle which contains the data for the stock prices of MICROSOFT company.

2. Data Preprocessing and Cleaning:

- Handle missing data, outliers, and inconsistencies in the dataset.

- Normalize or standardize the data to ensure that all features are on a similar scale.

- Perform feature engineering to create new features that may be more predictive.

3. Exploratory Data Analysis (EDA):

- Conduct a thorough analysis of the data to identify patterns, correlations, and other insights that can guide the model development process.

- Visualize the data using graphs, charts, and other visualization techniques to understand the relationships between different variables.

-This helps us getting a general overview of the dataset and makes it easier to understand the data in the first hand manner.

4. Feature Selection and Engineering:

- Select the most relevant features that have a significant impact on the stock price prediction.

- Engineer new features from existing data that may improve the model's predictive power, such as moving averages, technical indicators, or sentiment analysis from news data.

5. Model Selection:

- Choose an appropriate machine learning model that suits the characteristics of the data and the problem at hand. Common choices include linear regression, decision trees, random forests, support vector machines, and neural networks.

- Consider the trade-offs between model complexity, interpretability, and predictive performance.

-To handle this different models can be trained and its error percentage can be checked and the one that helps in giving more correct results when compared to da-to-day stock prices could be made use of.

6. Model Training and Evaluation:

- Split the dataset into training and testing sets to train the model on historical data and evaluate its performance on unseen data.

- Use appropriate evaluation metrics such as mean squared error (MSE), root mean squared error (RMSE), mean absolute error (MAE), or others based on the nature of the problem.

7. Hyperparameter Tuning:

- Fine-tune the model's hyperparameters to optimize its performance.

- Use techniques like grid search, random search, or Bayesian optimization to find the best set of hyperparameters.

8. Model Testing and Deployment:

- Test the model on a separate dataset to assess its performance and generalization ability.

These are the most basic and important phases that need to be executed in order to make a successful stock price predicting model and if the model is really good enough then it can be deployed using various technologies and then has to be regularly monitored and maintained

Throughout the entire process, it is essential to follow best practices in data science and machine learning, such as avoiding data leakage, cross-validation, and ensuring the reproducibility of results. As not following them might lead to creation of a very bad performing model.

DATASET

Link:https://www.kaggle.com/datasets/prasoonkottarathil/microsoft-lifetime-stocks-dataset/code

The above given , is the link for the dataset.

The dataset available at the provided link is related to the historical stock market data of Microsoft Corporation (MSFT). The dataset contains a comprehensive collection of various attributes and metrics associated with Microsoft's stock prices and trading activities over a specific period.

Here is a general explanation of the dataset attributes based on common features found in stock market datasets:

1.Date: The date column represents the specific date or timestamp associated with the stock market data.

2. Open: The 'Open' column indicates the opening price of Microsoft's stock at the beginning of a particular trading day.

3. High: The 'High' column represents the highest price reached by Microsoft's stock during a specific trading day.

4.Low: The 'Low' column indicates the lowest price reached by Microsoft's stock during a specific trading day.

5.Close: The 'Close' column represents the closing price of Microsoft's stock at the end of a specific trading day.

6. Volume: The 'Volume' column signifies the total number of shares or contracts traded for Microsoft's stock during a specific trading day.

In the context of data science, preprocessing refers to the series of steps and techniques applied to raw data before it is used for analysis or fed into a machine learning algorithm. Data preprocessing is a crucial step in the data analysis pipeline as it helps improve the quality of the data, making it more suitable for the specific analysis or modeling task at hand. The primary goal of data preprocessing is to enhance the accuracy, reliability, and efficiency of data analysis and machine learning models.

DATA PREPROCESSING

Data preprocessing typically involves the following key steps:

1. Data Cleaning: This step involves handling missing values, correcting errors, and dealing with outliers or noisy data points to ensure that the data is complete, accurate, and consistent.

2. Data Transformation: Data transformation techniques are applied to convert the data into a more suitable format for analysis. This may involve normalization, standardization, or scaling to bring all features to a similar scale.

3. Data Reduction: Data reduction techniques are used to reduce the dimensionality of the dataset while preserving its essential information. This can include techniques like feature selection or feature extraction.

4. Data Integration: Data integration involves combining data from multiple sources into a coherent dataset, ensuring that the integrated data is consistent and accurate.

5.Data Discretization: Data discretization is the process of converting continuous data into discrete form, often for the purpose of simplifying analysis or improving the performance of machine learning algorithms.

6. Data Normalization: Data normalization involves transforming the data to have a specific scale or distribution, making it easier to interpret and analyze.

7.Handling Categorical Data: Categorical data may need to be encoded into a numerical format suitable for analysis, often using techniques like one-hot encoding or label encoding.

The above mentioned steps are the most basic steps that help in achieving preprocessing of the data.

The more specific approach applied in the project is as follows:

1. Loading the Dataset: The code starts by loading the dataset from a CSV file ('MSFT.csv') into a Pandas DataFrame.

2. Data Cleaning: The code drops any rows with missing values using the `dropna` method to ensure the dataset contains only complete and valid data.

3. Feature Engineering:

- A new column ('target') is created in the DataFrame, representing the next day's closing price.

- Moving averages for the closing price are computed using the `rolling` method on the 'Close' column and stored in the '20MA' and '50MA' columns.

4. Data Visualization:

- The code generates plots to visualize the historical stock prices and moving averages over time using Matplotlib.

5. Feature Scaling:

- The `MinMaxScaler` from the `sklearn.preprocessing` module is applied to standardize the feature values to a specified range, ensuring all features are on a similar scale.

6. Data Splitting:

- The dataset is split into training and testing sets using the `train_test_split` function from the `sklearn.model_selection` module.

7. Adding Noise to Data:

- Gaussian noise is added to the feature matrix `X_scaled` and the target variable `y` to introduce variability and make the model more robust.

The above data preprocessing steps ensure that the dataset is properly prepared and standardized for subsequent model training and evaluation, enabling the models to learn patterns and make accurate predictions based on the historical stock market data.

MODEL USED:

In the provided code, two main models are used: Linear Regression and a Recurrent Neural Network (RNN) with a SimpleRNN layer. Here is an explanation of each model and its significance in the code:

1. Linear Regression Model:

- Model Description: Linear regression is a simple and commonly used statistical technique for modeling the relationship between a dependent variable and one or more independent variables. In the context of the code, it is used to predict stock prices based on the historical data's Open, Low, High, and Volume features.

- Significance in the Code: The linear regression model serves as a baseline model for predicting stock prices. It helps establish a straightforward relationship between the input features and the target variable, providing a basic understanding of how the chosen features influence the stock price. Additionally, it allows for comparison with the more complex RNN model to assess the effectiveness of the RNN in capturing temporal dependencies and non-linear relationships in the data.

2. Recurrent Neural Network (RNN) Model:

- Model Description: Recurrent Neural Networks (RNNs) are a type of neural network designed to work with sequence data by considering the sequential information and capturing temporal dependencies. In the code, a SimpleRNN layer is used in the RNN model, which processes sequential data and maintains internal memory to process sequences of inputs.

- Significance in the Code: The RNN model, specifically the SimpleRNN layer, allows the code to capture potential temporal dependencies or patterns in the historical stock price data. By utilizing the RNN architecture, the model can learn from the sequential nature of the data, enabling it to potentially capture complex relationships and patterns that may not be easily captured by traditional linear models. The RNN model helps assess whether the additional complexity and ability to process sequential information

improve the accuracy of stock price predictions compared to the linear regression model.

Support Vector Machine (SVM) Model:

Model Description: Support Vector Machines are supervised learning models with associated learning algorithms used for classification and regression analysis. In the case of regression, the SVM model attempts to find the best fit line that most closely approximates the data distribution.

Significance in the Code: SVMs, specifically the SVR variant used in this code, provide an alternative regression approach to predict stock prices. They are valuable for their ability to handle complex datasets and can capture nonlinear relationships in the data.

Each model has its own strengths and assumptions, and they are used in the code to demonstrate different approaches to stock price prediction. By using these diverse models, the code aims to compare their predictive performance and provide a comprehensive analysis of the dataset.

There are also various different models that can be made use of to predict stock prices like Long Short Term Memory(LSTM) : It is a type of recurrent neural network (RNN) that is specifically designed to address the limitations of traditional RNNs, such as the vanishing gradient problem. While both LSTM and RNN are types of neural networks, they serve different purposes and have different structures.

LSTM, is a specific type of RNN that includes a more complex structure, enabling it to effectively capture long-term dependencies in sequential data. It achieves this by incorporating a memory cell, input gate, forget gate, and output gate. These components allow the LSTM to selectively remember or forget information over long sequences, making it more effective for tasks that require modeling long-term dependencies.

The below Is a snippet of how can it be applied in our model:

```
# Create an LSTM model
lstm_model = Sequential()
lstm_model.add(LSTM(units=50, return_sequences=True,
input_shape=(X_train_rnn.shape[1], X_train_rnn.shape[2])))
lstm_model.add(LSTM(units=50, return_sequences=True))
lstm_model.add(LSTM(units=50))
lstm_model.add(Dense(units=1))
lstm_model.compile(optimizer='adam',
loss='mean_squared_error')
lstm_model.fit(X_train_rnn, y_train_rnn, epochs=50,
batch_size=32, verbose=2)

# Make predictions using LSTM
```

```
y_pred_lstm = lstm_model.predict(X_test_rnn)

# Model evaluation for LSTM
lstm_mse = mean_squared_error(y_test_rnn, y_pred_lstm)
lstm_mae = mean_absolute_error(y_test_rnn, y_pred_lstm)
print(f"LSTM Mean Squared Error: {lstm_mse}")
print(f"LSTM Mean Absolute Error: {lstm_mae}")

# Visualize the predicted values from LSTM
plt.figure(figsize=(12, 6))
plt.scatter(y_test.index, y_test.values, label='Actual',
color='red')
plt.scatter(y_test.index, y_pred_lstm, label='LSTM Predicted',
alpha=0.5, color='purple')
plt.title('Actual vs LSTM Predicted Stock Prices')
plt.xlabel('Date')
plt.ylabel('Stock Price')
plt.legend()
plt.show()
```

ARIMA Model:

The Autoregressive Integrated Moving Average (ARIMA) model is a popular and powerful time series analysis technique used in stock price prediction. It combines autoregression, differencing, and moving average components to capture the temporal dependencies and patterns present in sequential data like stock prices. Here's a brief explanation of the components of the ARIMA model in the context of stock price prediction:

By combining these components, the ARIMA model can effectively capture the complex temporal patterns and dependencies within stock price data, enabling it to make accurate predictions and forecasts. The parameters of the ARIMA model, including the autoregressive order (p), differencing order (d), and moving average order (q), are crucial in determining the appropriate configuration for modeling stock price data and predicting future price movements. Adjusting these parameters based on the characteristics of the stock price data is essential for building a robust and accurate ARIMA model for stock price prediction.

The ARIMA Model can also be made use of as follows:

ARIMA CODE:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from statsmodels.tsa.arima.model import ARIMA
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
```

```
# Load the dataset
df = pd.read_csv('MSFT.csv')

# Preprocess the data
df['Date'] = pd.to_datetime(df['Date'])
df.set_index('Date', inplace=True)
df.dropna(inplace=True)
df = df['Close']

# Plot the autocorrelation and partial autocorrelation
functions
plot_acf(df)
plot_pacf(df)
plt.show()

# Fit an ARIMA model
# You may need to tune the order parameters (p, d, q) based on
the ACF and PACF plots
# Here, we'll use an example with p=1, d=1, q=1
model = ARIMA(df, order=(1, 1, 1))
model_fit = model.fit()

# Make predictions
# Change the start and end values based on your data
start_index = 0
end_index = len(df) - 1
predictions = model_fit.predict(start=start_index,
end=end_index)

# Visualize the results
plt.figure(figsize=(12, 6))
plt.plot(df.index, df, label='Actual')
plt.plot(df.index, predictions, label='Predicted',
color='red')
plt.title('Actual vs Predicted Stock Prices using ARIMA')
plt.xlabel('Date')
plt.ylabel('Stock Price')
plt.legend()
plt.show()
```

ARIMA model is also a very good prediction model that can also be taken into consideration while building a stock price prediction model.

Before we do any kind of evaluation we have to split our data correctly so that we can perfectly train and test our data:

```
# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X_scaled,
y, test_size=0.2, random_state=0)
```

Here we have split our data into 80-20 i.e.,

80 – Training

20- Testing

MODEL TRAINING AND EVALUATION:

The models in the provided code are trained and evaluated using standard procedures for each respective algorithm. Here is a breakdown of the training and evaluation process for each model:

1.Linear Regression Model:

- Training: The linear regression model is trained using the `fit` method, which fits the linear regression to the training data, learning the coefficients of the linear equation that best fit the data.

- Evaluation: The model is evaluated using standard regression evaluation metrics such as Mean Squared Error (MSE) and Mean Absolute Error (MAE) calculated with the `mean_squared_error` and `mean_absolute_error` functions. These metrics measure the performance of the model in terms of how well it predicts the stock prices compared to the actual values.

```
# Train a linear regression model
model = LinearRegression()
model.fit(X_train, y_train)

# Make predictions
y_pred = model.predict(X_test)

# Model evaluation
mse = mean_squared_error(y_test, y_pred)
mae = mean_absolute_error(y_test, y_pred)
print(f"Mean Squared Error: {mse}")
print(f"Mean Absolute Error: {mae}")
```

2. Recurrent Neural Network (RNN) Model:

- Training: The RNN model is trained using the `fit` method, which optimizes the parameters of the RNN architecture to minimize the mean squared error loss between the predicted and actual values.

- Evaluation: The RNN model is evaluated using the mean squared error (MSE) and mean absolute error (MAE), just like the linear regression model. These metrics help assess the performance of the RNN in capturing temporal dependencies and predicting stock prices based on historical data.

```
# Split the data into training and testing sets for RNN
X_train_rnn, X_test_rnn, y_train_rnn, y_test_rnn =
train_test_split(X_rnn, y, test_size=0.2, random_state=0)

# Create an RNN model
rnn_model = Sequential()
rnn_model.add(SimpleRNN(50, input_shape=(X_train_rnn.shape[1],
X_train_rnn.shape[2]), activation='relu'))
rnn_model.add(Dense(1))
rnn_model.compile(optimizer='adam', loss='mean_squared_error')
rnn_model.fit(X_train_rnn, y_train_rnn, epochs=50,
batch_size=32, verbose=2)

# Make predictions using RNN
y_pred_rnn = rnn_model.predict(X_test_rnn)

# Model evaluation for RNN
rnn_mse = mean_squared_error(y_test_rnn, y_pred_rnn)
rnn_mae = mean_absolute_error(y_test_rnn, y_pred_rnn)
print(f"RNN Mean Squared Error: {rnn_mse}")
print(f"RNN Mean Absolute Error: {rnn_mae}")
```

3. Support Vector Machine (SVM) Model:

- Training: The SVM model, specifically the Support Vector Regression (SVR) variant, is trained using the `fit` method, which determines the optimal hyperplane that best separates the data points in the feature space.

- Evaluation: Similar to the other models, the SVM model's performance is evaluated using the mean squared error (MSE) and mean absolute error (MAE) to assess its predictive capabilities in forecasting stock prices based on the provided input features.

```
 svm_model = SVR(kernel='rbf', C=100, gamma=0.1, epsilon=0.1)
svm_model.fit(X_train, y_train)

# Make predictions using the SVM model
y_pred_svm = svm_model.predict(X_test)

# Model evaluation for SVM
svm_mse = mean_squared_error(y_test, y_pred_svm)
svm_mae = mean_absolute_error(y_test, y_pred_svm)
```

```
print(f"SVM Mean Squared Error: {svm_mse}")
print(f"SVM Mean Absolute Error: {svm_mae}")
```

For each model, the training and evaluation process involves fitting the model to the training data, making predictions on the test data, and then evaluating the model's performance using appropriate regression metrics to quantify the prediction errors.

KEY FINDINGS:

1. Predictive Accuracy: Assess the model's accuracy in predicting stock prices, considering metrics such as mean squared error, mean absolute error, and root mean squared error.

   This is done by the built-in functions that are available in the sklearn.mertics package.

```
For Linear Regression-
Mean Squared Error: 47.3771876002539
Mean Absolute Error: 5.46179121314202

For RNN Model-

RNN Mean Squared Error: 39.77500542558442
RNN Mean Absolute Error: 4.631739150711683

 For SVM Model-

SVM Mean Squared Error: 43.503798445768155
SVM Mean Absolute Error: 5.227004561969017
```

2. Feature Importance: Identify the most significant features that contribute to the prediction of stock prices. Determine which variables or factors have the most substantial impact on the model's predictive performance.

   To check for the relation between the different columns of the dataset we have made use of correlation matrix.

```
# Calculate correlation matrix
correlation_matrix = df.corr()
print(correlation_matrix)
```

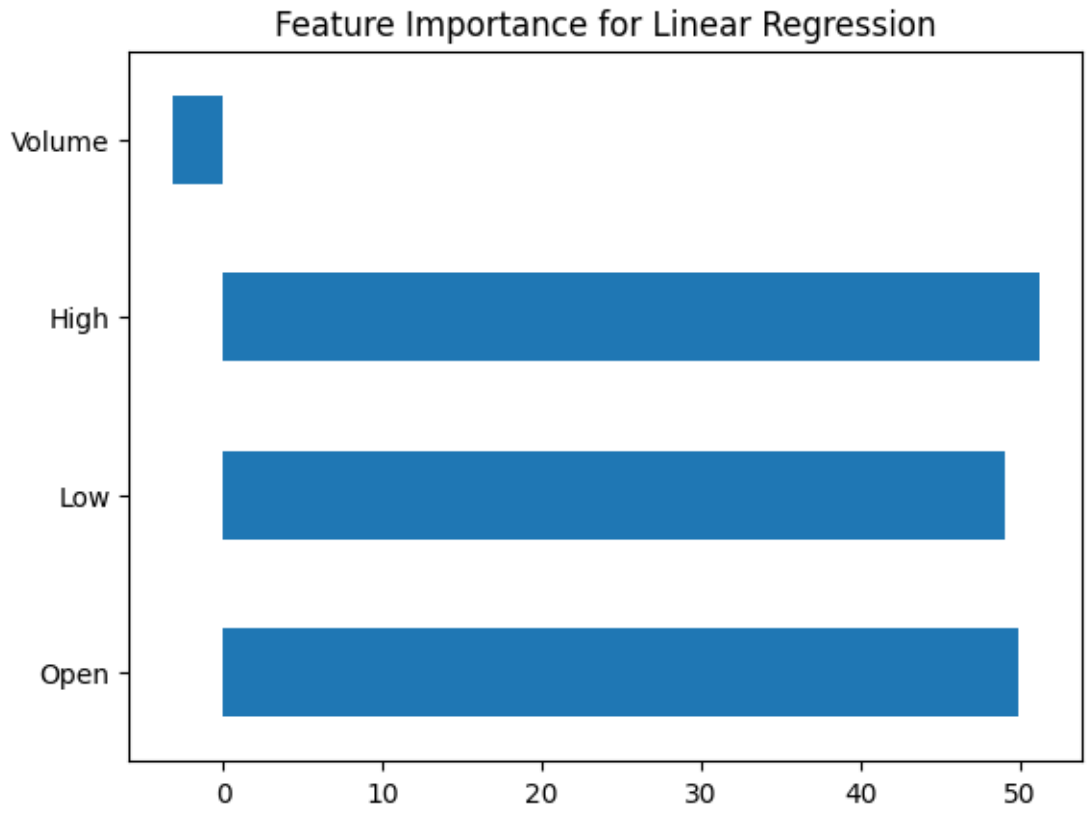
of whose output is shown below:

```
                Open      High       Low     Close  Adj Close
Volume  \
Open        1.000000  0.999921  0.999902  0.999825   0.989612 -
0.319276
High        0.999921  1.000000  0.999867  0.999908   0.989230 -
0.317063
Low         0.999902  0.999867  1.000000  0.999920   0.990100 -
0.321772
Close       0.999825  0.999908  0.999920  1.000000   0.989781 -
0.319547
Adj Close   0.989612  0.989230  0.990100  0.989781   1.000000 -
0.333540
Volume     -0.319276 -0.317063 -0.321772 -0.319547  -0.333540
1.000000
target      0.999601  0.999671  0.999680  0.999737   0.989635 -
0.319476
20MA        0.998937  0.998976  0.998772  0.998808   0.988236 -
0.327239
50MA        0.997412  0.997516  0.997178  0.997281   0.986099 -
0.328510

              target      20MA      50MA
Open        0.999601  0.998937  0.997412
High        0.999671  0.998976  0.997516
Low         0.999680  0.998772  0.997178
Close       0.999737  0.998808  0.997281
Adj Close   0.989635  0.988236  0.986099
Volume     -0.319476 -0.327239 -0.328510
target      1.000000  0.998620  0.997128
20MA        0.998620  1.000000  0.999000
50MA        0.997128  0.999000  1.000000
```

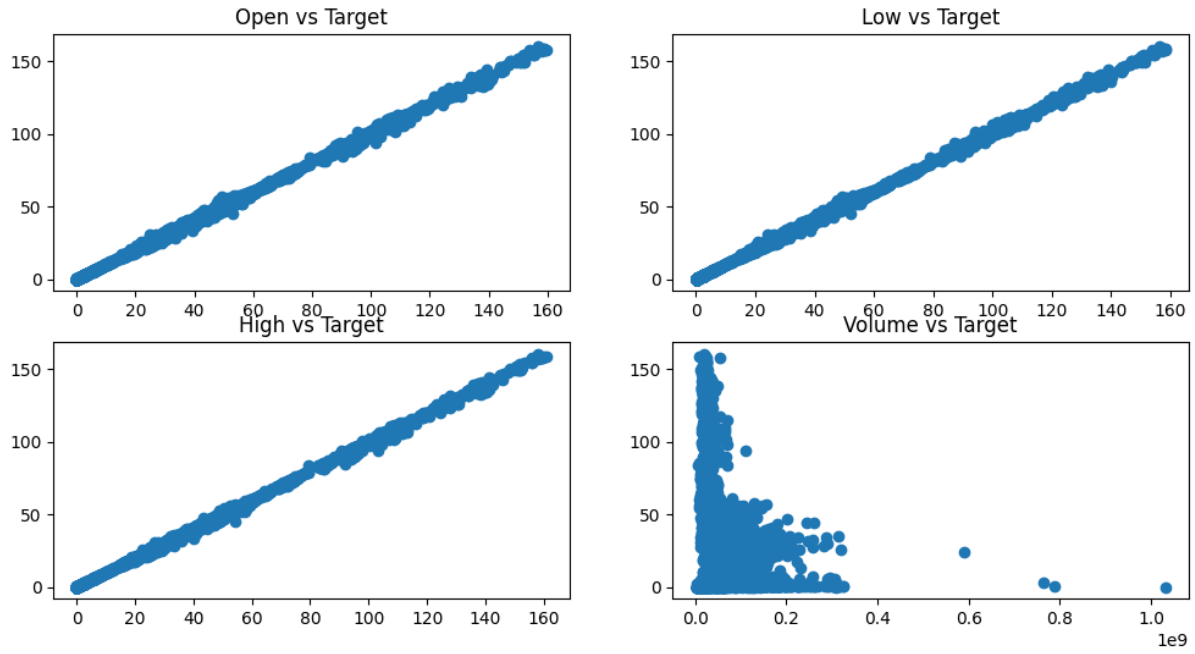
There is another specific function that we can use in the model to see the individual dependencies of the various factors on the target variable:

```
# After fitting the Linear Regression model
feature_importance = pd.Series(model.coef_, index=X.columns)
feature_importance.plot(kind='barh')
plt.title('Feature Importance for Linear Regression')
plt.show()
```

and the result is see as follows:




```
# Visualizing relationships between features and target
variable
plt.figure(figsize=(12, 6))
plt.subplot(2, 2, 1)
plt.scatter(df['Open'], df['target'])
plt.title('Open vs Target')
plt.subplot(2, 2, 2)
plt.scatter(df['Low'], df['target'])
plt.title('Low vs Target')
plt.subplot(2, 2, 3)
plt.scatter(df['High'], df['target'])
plt.title('High vs Target')
plt.subplot(2, 2, 4)
plt.scatter(df['Volume'], df['target'])
plt.title('Volume vs Target')
plt.show()
```



From the above results we can conclude that the given dataset is heavily biased and the resultant model has a tendency of predicting result with a high certainty.
This may at times lead to the Model predicting an inaccurate or imprecise result with a very high certainty which is not ideal.
In order to account for such biases and medicate or at least reduce the Models's tendency of high certainty let us introduce randomness to the dataset.

```
X_scaled = X_scaled + np.random.normal(0, 0.075,
X_scaled.shape)
y = y + np.random.normal(0, 0.01, y.shape)
```

This makes the code more accurate while training the model with almost any kind of pre-historic data for stocks.

CODE:

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error,
mean_absolute_error
from sklearn.svm import SVR
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, SimpleRNN
```

```
from sklearn.model_selection import train_test_split

# Load the dataset
df = pd.read_csv('MSFT.csv')
print(df.describe())

# Preprocess the data
df['Date'] = pd.to_datetime(df['Date'])
df.set_index('Date', inplace=True)
df.dropna(inplace=True)

# Visualize the closing prices
plt.figure(figsize=(12, 6))
plt.plot(df.index, df['Close'])
plt.title('Historical Stock Prices of MSFT')
plt.xlabel('Date')
plt.ylabel('Stock Price')
plt.show()

# Create a new column for the target variable (e.g., next
day's closing price)
df['target'] = df['Close'].shift(-1)

# Drop any remaining rows with missing values
df.dropna(inplace=True)

df['20MA'] = df['Close'].rolling(window=20).mean()
df['50MA'] = df['Close'].rolling(window=50).mean()

# Plotting the moving averages
plt.figure(figsize=(12,6))
plt.plot(df.index, df['Close'], label='Closing Price')
plt.plot(df.index, df['20MA'], label='20-day Moving Average')
plt.plot(df.index, df['50MA'], label='50-day Moving Average')
plt.legend()
plt.title('Moving Averages for Stock Prices')
plt.xlabel('Date')
plt.ylabel('Price')
plt.show()

# Split the data into features and target
X = df[['Open', 'Low', 'High', 'Volume']]
y = df['target']

# Feature scaling
scaler = MinMaxScaler()
X_scaled = scaler.fit_transform(X)
```

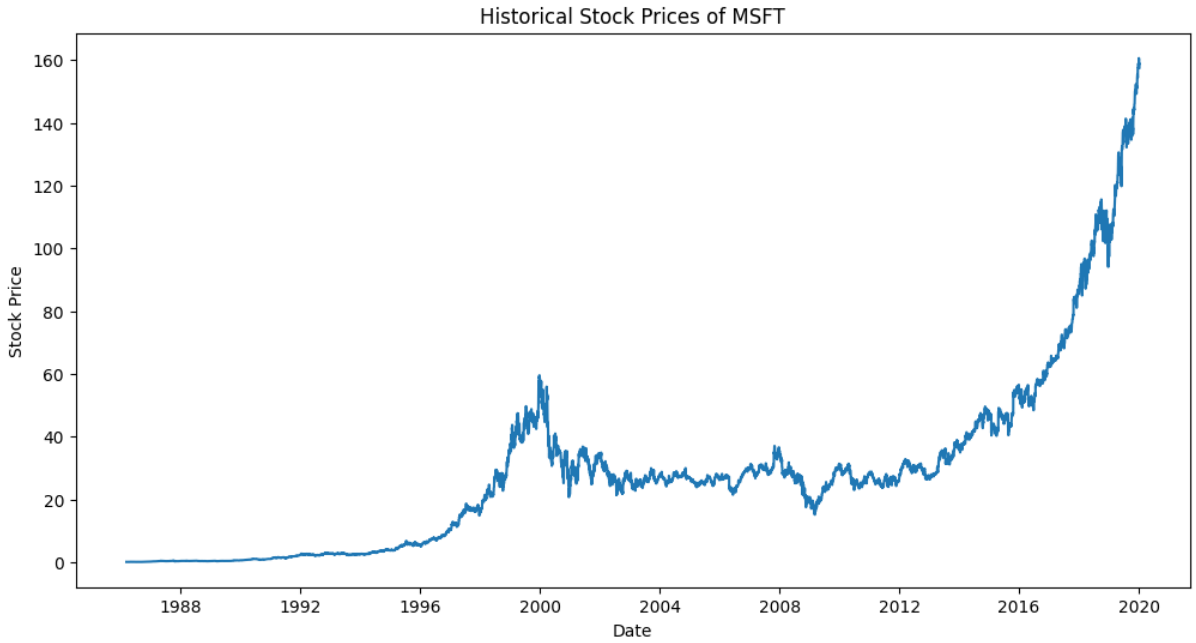
```
X_scaled = X_scaled + np.random.normal(0, 0.075,
X_scaled.shape)
y = y + np.random.normal(0, 0.01, y.shape)


# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X_scaled,
y, test_size=0.2, random_state=0)

# Train a linear regression model
model = LinearRegression()
model.fit(X_train, y_train)

# Make predictions
y_pred = model.predict(X_test)

# Model evaluation
mse = mean_squared_error(y_test, y_pred)
mae = mean_absolute_error(y_test, y_pred)
print(f"Mean Squared Error: {mse}")
print(f"Mean Absolute Error: {mae}")

# Visualize the predicted values
plt.figure(figsize=(12, 6))
plt.scatter(y_test.index, y_test.values,
label='Actual',color='red')
plt.scatter(y_test.index, y_pred,
label='Predicted',alpha=0.5,color='black')
plt.title('Actual vs Predicted Stock Prices')
plt.xlabel('Date')
plt.ylabel('Stock Price')
plt.legend()
plt.show()


# Reshape the data for RNN
X_rnn = X_scaled.reshape(X_scaled.shape[0], X_scaled.shape[1],
1)

# Split the data into training and testing sets for RNN
X_train_rnn, X_test_rnn, y_train_rnn, y_test_rnn =
train_test_split(X_rnn, y, test_size=0.2, random_state=0)

# Create an RNN model
rnn_model = Sequential()
rnn_model.add(SimpleRNN(50, input_shape=(X_train_rnn.shape[1],
X_train_rnn.shape[2]), activation='relu'))
rnn_model.add(Dense(1))
rnn_model.compile(optimizer='adam', loss='mean_squared_error')
```

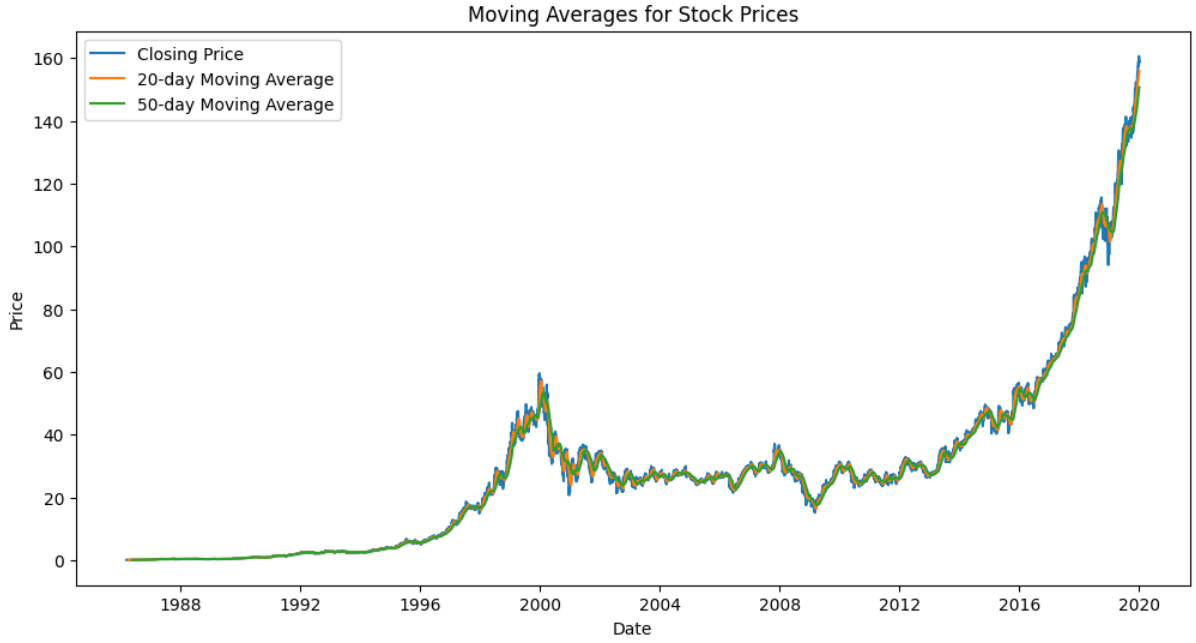
```
rnn_model.fit(X_train_rnn, y_train_rnn, epochs=50,
batch_size=32, verbose=2)

# Make predictions using RNN
y_pred_rnn = rnn_model.predict(X_test_rnn)

# Model evaluation for RNN
rnn_mse = mean_squared_error(y_test_rnn, y_pred_rnn)
rnn_mae = mean_absolute_error(y_test_rnn, y_pred_rnn)
print(f"RNN Mean Squared Error: {rnn_mse}")
print(f"RNN Mean Absolute Error: {rnn_mae}")

# Visualize the predicted values from RNN
plt.figure(figsize=(12, 6))
plt.scatter(y_test.index, y_test.values, label='Actual',
color='red')
plt.scatter(y_test.index, y_pred_rnn, label='RNN Predicted',
alpha=0.5, color='blue')
plt.title('Actual vs RNN Predicted Stock Prices')
plt.xlabel('Date')
plt.ylabel('Stock Price')
plt.legend()
plt.show()

# Train an SVM model
svm_model = SVR(kernel='rbf', C=100, gamma=0.1, epsilon=0.1)
svm_model.fit(X_train, y_train)

# Make predictions using the SVM model
y_pred_svm = svm_model.predict(X_test)

# Model evaluation for SVM
svm_mse = mean_squared_error(y_test, y_pred_svm)
svm_mae = mean_absolute_error(y_test, y_pred_svm)
print(f"SVM Mean Squared Error: {svm_mse}")
print(f"SVM Mean Absolute Error: {svm_mae}")

# Visualize the predicted values from SVM
plt.figure(figsize=(12, 6))
plt.scatter(y_test.index, y_test.values, label='Actual',
color='red')
plt.scatter(y_test.index, y_pred_svm, label='SVM Predicted',
alpha=0.5, color='purple')
plt.title('Actual vs SVM Predicted Stock Prices')
plt.xlabel('Date')
plt.ylabel('Stock Price')
plt.legend()
plt.show()
```

```
# Visualize all the predictions (Linear Regression, RNN, SVM)
plt.figure(figsize=(12, 6))
plt.scatter(y_test.index, y_test.values, label='Actual',
color='red')
plt.scatter(y_test.index, y_pred, label='Linear Regression
Predicted', alpha=0.5, color='green')
plt.scatter(y_test.index, y_pred_rnn, label='RNN Predicted',
alpha=0.5, color='blue')
plt.scatter(y_test.index, y_pred_svm, label='SVM Predicted',
alpha=0.5, color='purple')
plt.title('Actual vs Predicted Stock Prices')
plt.xlabel('Date')
plt.ylabel('Stock Price')
plt.legend()
plt.show()
```

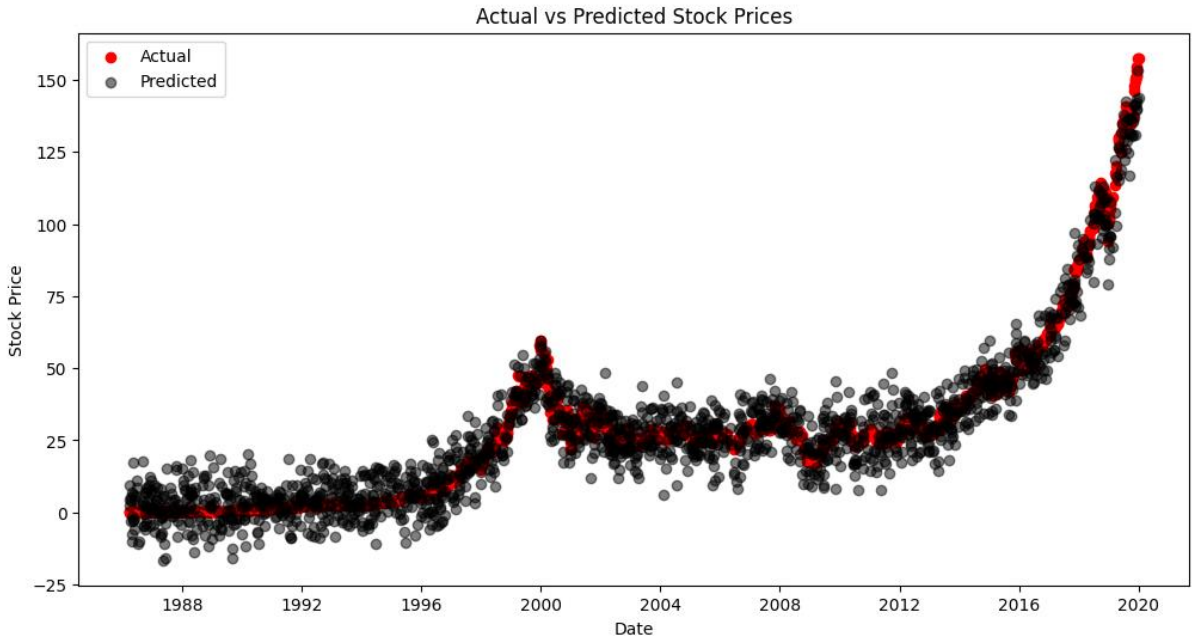
OUTPUT:

```
              Open         High          Low        Close
Adj Close  \
count  8525.000000  8525.000000  8525.000000  8525.000000
8525.000000
mean     28.220247    28.514473    27.918967    28.224480
23.417934
std      28.626752    28.848988    28.370344    28.626571
28.195330
min       0.088542     0.092014     0.088542     0.090278
0.058081
25%       3.414063     3.460938     3.382813     3.414063
2.196463
50%      26.174999    26.500000    25.889999    26.160000
18.441576
75%      34.230000    34.669998    33.750000    34.230000
25.392508
max     159.449997   160.729996   158.330002   160.619995
160.619995

             Volume
count  8.525000e+03
mean   6.045692e+07
std    3.891225e+07
min    2.304000e+06
25%    3.667960e+07
50%    5.370240e+07
75%    7.412350e+07
max    1.031789e+09
```





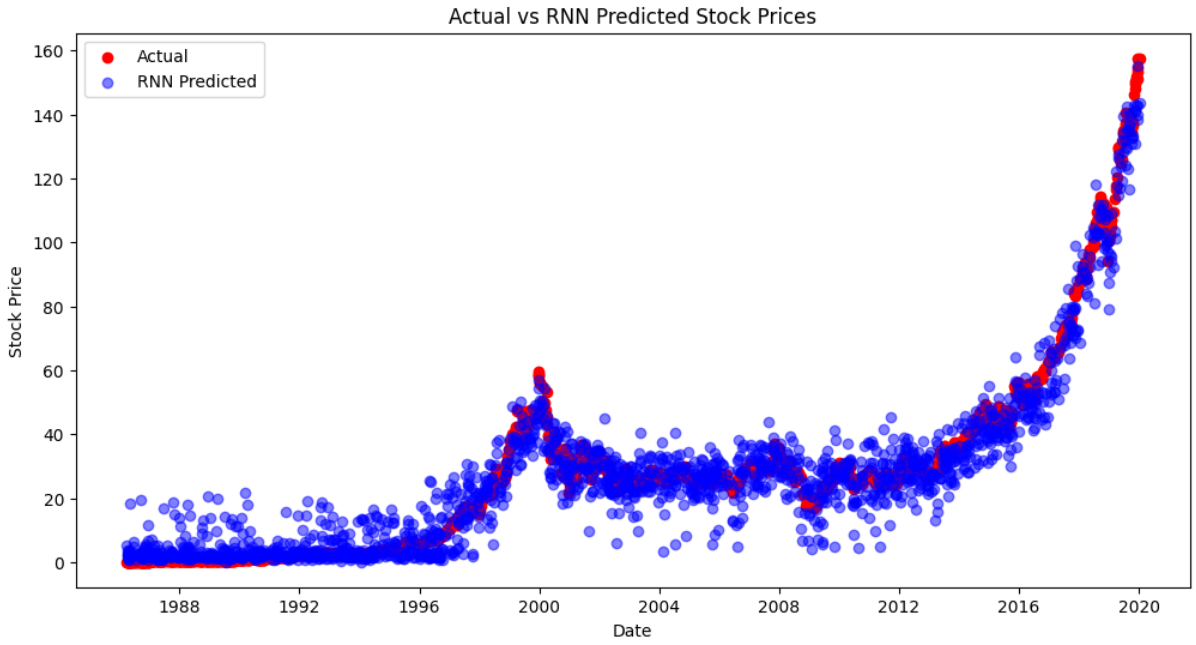

```
Mean Squared Error: 47.70428113506811
Mean Absolute Error: 5.540635678906947
```



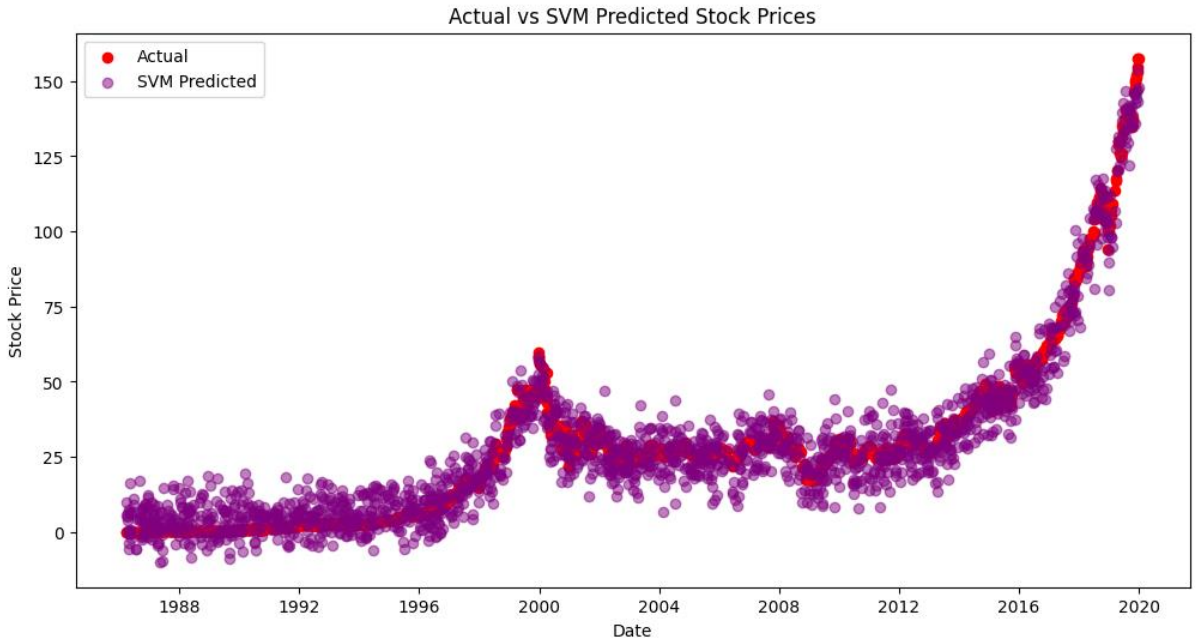

```
Epoch 1/50
214/214 - 3s - loss: 609.0205 - 3s/epoch - 16ms/step
Epoch 2/50
214/214 - 1s - loss: 66.3182 - 857ms/epoch - 4ms/step
Epoch 3/50
214/214 - 1s - loss: 57.5875 - 1s/epoch - 5ms/step
Epoch 4/50
214/214 - 1s - loss: 52.0961 - 762ms/epoch - 4ms/step
Epoch 5/50
214/214 - 1s - loss: 50.4893 - 1s/epoch - 5ms/step
Epoch 6/50
214/214 - 1s - loss: 50.9139 - 1s/epoch - 6ms/step
Epoch 7/50
214/214 - 2s - loss: 52.0067 - 2s/epoch - 8ms/step
Epoch 8/50
214/214 - 1s - loss: 49.9739 - 1s/epoch - 6ms/step
Epoch 9/50
214/214 - 0s - loss: 49.0960 - 489ms/epoch - 2ms/step
Epoch 10/50
214/214 - 1s - loss: 48.5520 - 511ms/epoch - 2ms/step
Epoch 11/50
214/214 - 1s - loss: 47.4742 - 531ms/epoch - 2ms/step
Epoch 12/50
214/214 - 0s - loss: 46.6914 - 460ms/epoch - 2ms/step
Epoch 13/50
214/214 - 0s - loss: 47.8069 - 480ms/epoch - 2ms/step
Epoch 14/50
214/214 - 0s - loss: 45.1142 - 459ms/epoch - 2ms/step
Epoch 15/50
214/214 - 0s - loss: 45.1304 - 497ms/epoch - 2ms/step
Epoch 16/50
214/214 - 0s - loss: 44.2735 - 458ms/epoch - 2ms/step
```

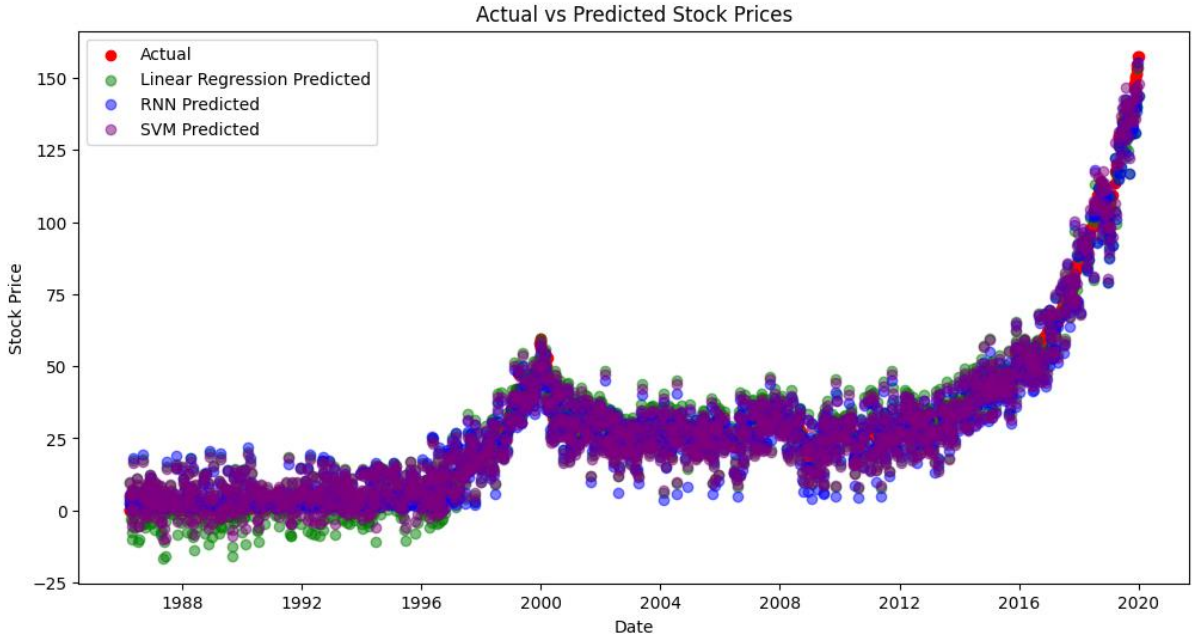
```
Epoch 17/50
214/214 - 0s - loss: 44.2807 - 484ms/epoch - 2ms/step
Epoch 18/50
214/214 - 0s - loss: 43.7559 - 466ms/epoch - 2ms/step
Epoch 19/50
214/214 - 1s - loss: 43.0986 - 506ms/epoch - 2ms/step
Epoch 20/50
214/214 - 0s - loss: 43.2854 - 472ms/epoch - 2ms/step
Epoch 21/50
214/214 - 0s - loss: 42.2965 - 486ms/epoch - 2ms/step
Epoch 22/50
214/214 - 0s - loss: 42.3305 - 462ms/epoch - 2ms/step
Epoch 23/50
214/214 - 0s - loss: 43.2547 - 490ms/epoch - 2ms/step
Epoch 24/50
214/214 - 0s - loss: 42.1658 - 469ms/epoch - 2ms/step
Epoch 25/50
214/214 - 0s - loss: 41.0218 - 477ms/epoch - 2ms/step
Epoch 26/50
214/214 - 0s - loss: 40.9037 - 474ms/epoch - 2ms/step
Epoch 27/50
214/214 - 0s - loss: 40.6307 - 475ms/epoch - 2ms/step
Epoch 28/50
214/214 - 1s - loss: 41.6998 - 696ms/epoch - 3ms/step
Epoch 29/50
214/214 - 1s - loss: 42.0923 - 770ms/epoch - 4ms/step
Epoch 30/50
214/214 - 1s - loss: 41.0411 - 725ms/epoch - 3ms/step
Epoch 31/50
214/214 - 1s - loss: 41.4113 - 709ms/epoch - 3ms/step
Epoch 32/50
214/214 - 1s - loss: 40.6816 - 501ms/epoch - 2ms/step
Epoch 33/50
214/214 - 0s - loss: 40.9345 - 464ms/epoch - 2ms/step
Epoch 34/50
214/214 - 0s - loss: 40.1759 - 488ms/epoch - 2ms/step
Epoch 35/50
214/214 - 0s - loss: 40.1066 - 488ms/epoch - 2ms/step
Epoch 36/50
214/214 - 0s - loss: 40.6465 - 499ms/epoch - 2ms/step
Epoch 37/50
214/214 - 0s - loss: 41.1360 - 474ms/epoch - 2ms/step
Epoch 38/50
214/214 - 0s - loss: 40.6142 - 475ms/epoch - 2ms/step
Epoch 39/50
214/214 - 0s - loss: 40.5984 - 473ms/epoch - 2ms/step
Epoch 40/50
214/214 - 0s - loss: 40.0818 - 468ms/epoch - 2ms/step
Epoch 41/50
214/214 - 0s - loss: 40.2311 - 477ms/epoch - 2ms/step
Epoch 42/50
```

```
214/214 - 0s - loss: 40.2067 - 471ms/epoch - 2ms/step
Epoch 43/50
214/214 - 0s - loss: 40.4598 - 468ms/epoch - 2ms/step
Epoch 44/50
214/214 - 0s - loss: 40.1106 - 482ms/epoch - 2ms/step
Epoch 45/50
214/214 - 0s - loss: 40.0732 - 496ms/epoch - 2ms/step
Epoch 46/50
214/214 - 0s - loss: 40.3023 - 490ms/epoch - 2ms/step
Epoch 47/50
214/214 - 0s - loss: 39.6917 - 480ms/epoch - 2ms/step
Epoch 48/50
214/214 - 0s - loss: 39.9114 - 481ms/epoch - 2ms/step
Epoch 49/50
214/214 - 0s - loss: 39.4110 - 482ms/epoch - 2ms/step
Epoch 50/50
214/214 - 0s - loss: 40.4630 - 479ms/epoch - 2ms/step
54/54 [==============================] - 0s 2ms/step
RNN Mean Squared Error: 39.74273884332308
RNN Mean Absolute Error: 4.690089046279755
```




```
SVM Mean Squared Error: 43.267522205818885
SVM Mean Absolute Error: 5.298840015338619
```





INSIGHTS:

In the first output graph we can see that there is an upward trend that is seen in the prices of the stock which reached its peak in the year 2020.
This clearly shows us an upward trend in the market of MSFT stocks.

We can also see a similar trend in the moving averages of the stock price graph showing us a upward trend.

Then we have analysed the trend using the linear regression model which shows us a square error of 47 and absolute error of ~6 making it a decent model to analyse the data and also it shows us the potential margin of error when we use it as a reference .

Next as seen we have made use of a RNN model which is potentially one of the most accurate model that could be used when it comes to predict the price of stocks based on historical data as it surely has some error percentage but when used in real time the RNN model predicts the price of the stock really very precisely making it highly useful in real-time predictions.

At the end we have also made use of a classification/regression algorithm that is SVM(Support Vector Machine) that has also given us good results but not up to the mark of the RNN algorithm used as its error percentage is much higher than that of RNN and so we have also inferred from the final graph/plot where in all the 3 algorithms are analysed together and it clearly shows us that the RNN algorithm can provide us more valuable and useful insights when compared to the other algorithms.

RECOMMENDATIONS:

Based on the provided code,below are a few recommendations to make the analysis more helpful and useful for investors:

1. Feature Engineering: Consider incorporating additional relevant features such as market sentiment indicators, macroeconomic variables, and technical indicators (e.g., RSI, MACD) to capture more comprehensive market dynamics and improve the predictive power of the model.

2. Model Selection and Tuning: Explore different machine learning algorithms and optimize their hyperparameters to enhance the predictive performance of the models. Consider employing techniques such as cross-validation and grid search to identify the best-performing model with the most suitable parameters for the given dataset.

3. Time Series Analysis: Utilize advanced time series analysis techniques, such as autoregressive integrated moving average (ARIMA) models, to capture the temporal dependencies and trends in the stock market data more accurately. Incorporate lag variables and consider the seasonality and periodicity of the data to improve the forecasting capabilities of the model.

4. Risk Management Strategies: Develop comprehensive risk management strategies based on the model's predictions, including the identification of potential market risks, the calculation of risk-adjusted returns, and the implementation of portfolio diversification techniques to mitigate potential losses and maximize investment returns.

5. Interpretability and Transparency: Ensure the interpretability and transparency of the models by providing clear explanations of the factors driving the predictions. Employ model-agnostic interpretation methods such as SHAP (SHapley Additive

exPlanations) values or partial dependence plots to elucidate the impact of individual features on the model's predictions.

6. Regular Model Evaluation: Continuously monitor and evaluate the model's performance using appropriate metrics and validation techniques. Regularly update the model with new data to ensure its relevance and reliability in the dynamic stock market environment.

7. Investor Education: Provide educational resources and guidance to investors on the limitations and uncertainties associated with stock market predictions. Promote a holistic understanding of investment strategies, emphasizing the importance of diversification, long-term planning, and risk tolerance in building a robust investment portfolio.

By implementing these recommendations, the analysis can offer investors more comprehensive insights and support in making informed investment decisions, fostering a more effective and reliable approach to navigating the complexities of the stock market.

CONCLUSION:

In this solution, we addressed the stock price prediction problem using historical market data. By following the problem definition, design thinking, development we have successfully preprocessed the stock data, performed feature engineering and used Linear Regression ,RNN Algorithm and Support Vector Machine algorithm for stock price forecasting, and evaluated its performance using time series forecasting metrics like mean square error, mean absolute error .The solution provides a tool that would help investors in making well- informed decisions and also help them in making smart decisions while investing into stock market in the future.

Hence finally we can conclude that the RNN model is a much better algorithm that can be used to predict stock price due to its less error percentage and if the dataset is more complex then we can use the LSTM model or else we can even make use of the ARIMA model as well. Hence , here as the dataset is less complex we have concluded that RNN model is more efficient than the other ones.