IBM Applied Data Science Project

Stock Price Prediction

PHASE 1

PROBLEM DEFINITION:

The primary goal of this project is to create a predictive model for stock price forecasting using

historical market data. Specifically, we aim to predict the future stock prices of Microsoft (MSFT) to

help investors make more informed decisions and potentially improve their investment strategies.

DESIGN THINKING:

1. Data Collection:

- Collect historical stock market data for Microsoft from the provided dataset.
- Key features include date, open price, close price, volume, and other relevant indicators.

2. Data Preprocessing:

- Clean the data: Handle missing values, outliers, and inconsistencies.
- Feature selection: Identify the most relevant features for forecasting.
- Data transformation: Convert categorical features (if any) into numerical representations.
- Time series data formatting: Ensure data is organized by date and sorted chronologically.

3. Feature Engineering:

- Create additional features that can enhance the predictive power of the model.
- Moving averages (e.g., 7-day, 30-day, etc.)
- Technical indicators (e.g., RSI, MACD, Bollinger Bands)
- Lagged variables (using past stock prices and indicators as features)

4. Model Selection:

Choose appropriate time series forecasting algorithms for stock price prediction.

Potential choices include:

- ARIMA (Auto Regressive Integrated Moving Average)
- LSTM (Long Short-Term Memory)
- Other machine learning models like Random Forest, XG Boost (for comparison)

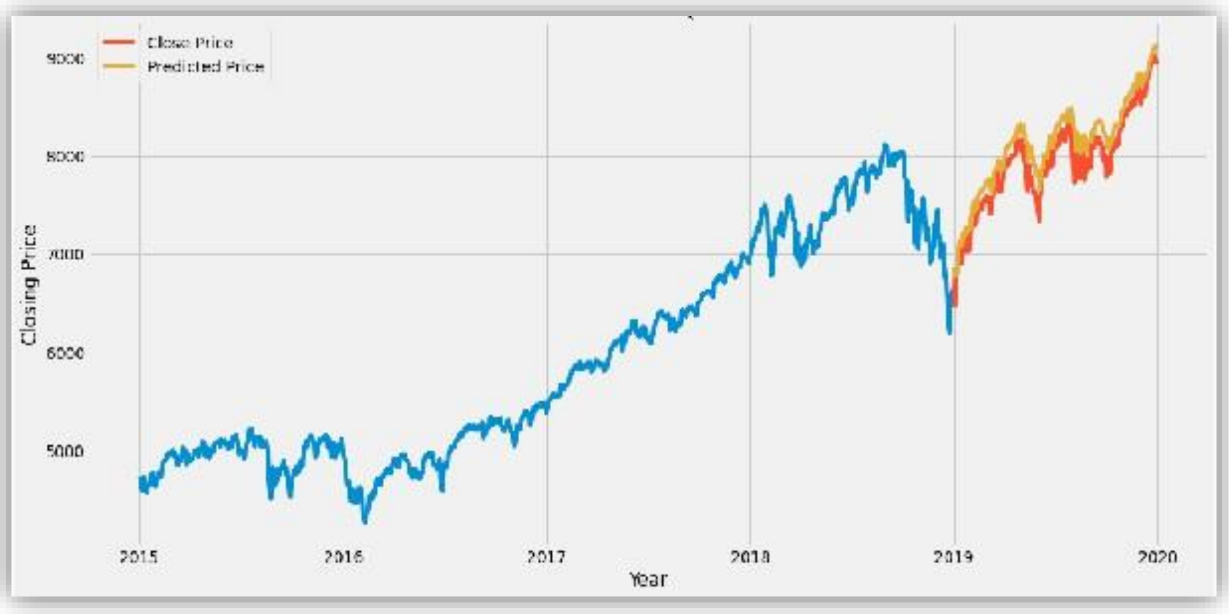
5. Model Training:

- Split the data into training and validation sets, maintaining chronological order.
- Train the selected model(s) using the preprocessed data.

6. Evaluation:

Assess the model performance using relevant time series forecasting metrics, such as Mean Absolute Error (MAE), Root Mean Squared Error (RMSE), and possibly others.

Compare the performance of different models to select the best-performing one.



Stock Price Prediction Analysis Approach

1. Problem Understanding:

Clearly define the problem you aim to solve: Predict future stock prices.

Understand the business context: What are the objectives and constraints of the project?

Define the scope: Which stocks or indices are you forecasting? What is the prediction horizon (e.g., daily, weekly, monthly)?

2. Data Collection:

Identify and gather historical stock market data from reliable sources (e.g., Yahoo Finance, Alpha Vantage).

Collect a diverse set of features, including price, volume, and potentially external factors (e.g., economic indicators, news sentiment) that might impact stock prices.

3. Data Exploration and Preprocessing:

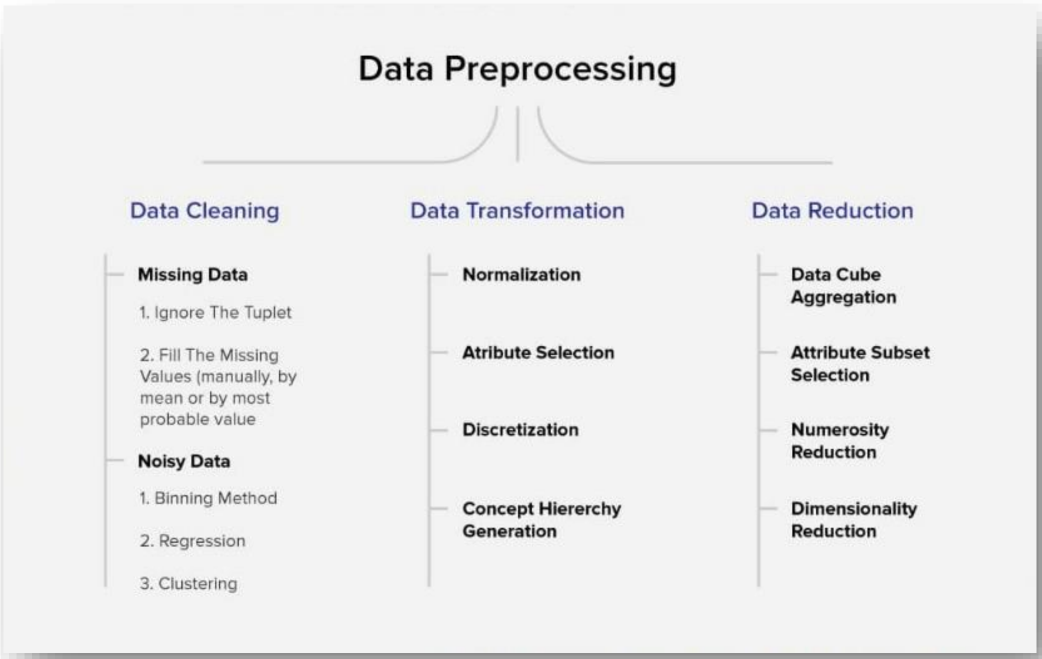
Perform initial data exploration to understand the dataset structure and quality.

Check for missing data, outliers, and data integrity issues.

Visualize time series data to identify trends, seasonality, and patterns.

Convert date columns into datetime objects and set them as the index.

Calculate returns or log-returns, as these are often used in stock price prediction.



4. Feature Engineering:

Create additional relevant features to improve model performance, such as moving averages, technical indicators (e.g., RSI, MACD), and lagged variables.

Consider encoding categorical variables or sentiment scores.

Normalize or scale features to ensure they have similar ranges.

5. Data Splitting:

Split the dataset into training, validation, and test sets.

Ensure that the validation and test sets are future timestamps, maintaining a chronological order.

6. Model Selection:

Choose appropriate machine learning or time series forecasting models:

For traditional time series data: ARIMA, SARIMA, Prophet.

For deep learning: LSTM, GRU, or other RNN-based models.

For machine learning: Random Forest, XG Boost, or SVR.

Consider multiple models for improved accuracy.

7. Model Training:

Train the selected models using the training dataset.

Experiment with different hyperparameters and architectures.

Monitor training progress and avoid overfitting.

8. Model Evaluation:

Evaluate model performance using relevant metrics such as Mean Absolute Error (MAE), Root Mean Squared Error (RMSE), Mean Absolute Percentage Error (MAPE), and directional accuracy.

Visualize predictions against actual stock prices to assess model fit.

Compare different models and select the best-performing one.

9. Hyperparameter Tuning:

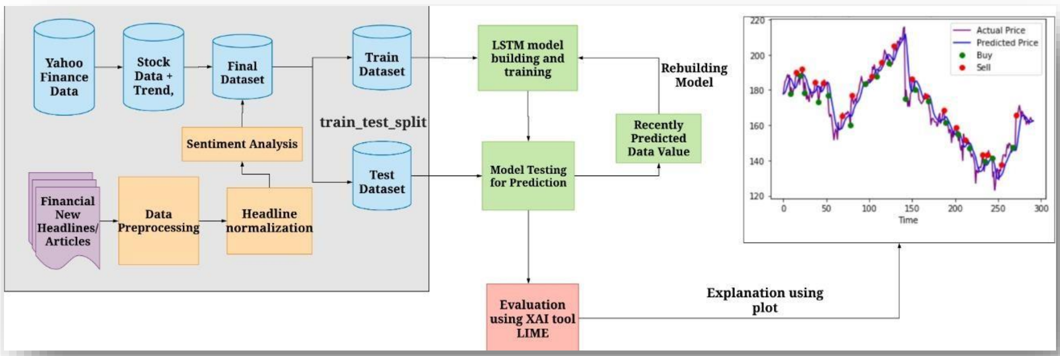
Fine-tune model hyperparameters using techniques like grid search or Bayesian optimization.

Re-train the model with optimized hyperparameters.

10. Model Interpretation:

Understand the importance of features using feature importance scores (e.g., SHAP values for tree-based models).

Explore model predictions to identify patterns and trends.



11. Deployment and Monitoring:

Deploy the trained model in a production environment if applicable.

Set up monitoring to track model performance and retrain as necessary.

12. Reporting and Communication:

Summarize the project in a report or presentation.

Explain the model predictions and limitations.

Provide actionable insights for investors based on the model forecasts.

Communicate results effectively to stakeholders.

13. Maintenance and Updates:

Regularly update the model with new data to keep it relevant.

Continuously monitor model performance and retrain as needed to adapt to changing market conditions.

This analysis approach provides a structured framework for developing and deploying a stock price prediction model, ensuring that we cover essential steps from data collection to model deployment and maintenance.

IBM APPLIED DATA SCIENCE PROJECTSTOCK PRICE PREDICTION (PHASE 2)

INNOVATION:

Innovation Idea: **Stock vector** - a high-dimensional representation of multiple stock-relatedfeatures or characteristics used to improve stock market predictions.

1. Innovative Stock Market Prediction Approach:

The research introduces an innovative neural network-based approach for stock marketprediction, departing from traditional methods. It leverages deep learning techniques toimprove the accuracy of predictions.

2. Real-time and Historical Data:

The study utilizes a combination of real-time and historical data from the stock market. Thisapproach allows for more comprehensive and dynamic analysis, considering both long-term trends and immediate market conditions.

3. Visualization and Analytics:

The research presents results in the form of visualizations and analytics. This can aid in understanding the dynamics of the stock market, making it accessible and interpretable forinvestors and analysts.

4. Internet of Multimedia of Things (IMMT):

The research incorporates the concept of the Internet of Multimedia Things (IMMT) for stock analysis. This suggests a multidisciplinary approach to studying stock market dynamics, potentially including multimedia data sources in the analysis.

5. Challenges of Traditional Neural Networks:

Traditional neural network algorithms are mentioned as having limitations in predicting thestock market, particularly due to random weight initialization. This highlights the need for more robust techniques.

6. Introduction of "Stock Vector":

The research introduces the concept of a "stock vector." Instead of using a single index or stock data, the model works with multi-stock, high-dimensional historical data. This reflects amore complex and data-rich approach to stock prediction.

7. Use of Deep Learning Models:

The study employs deep learning models, specifically the Deep Long Short-Term Memory (LSTM) neural network and the LSTM with an autoencoder. These models are designed to capture complex temporal dependencies in stock data, which can be challenging for

traditional models

8. Embedded Layer and Autoencoder:

The research employs an embedded layer and an autoencoder for data vectorization. Thesetechniques are used to convert the complex stock data into a format that can be effectivelyused by the LSTM models.

9. Model Comparison:

The research compares the performance of the deep LSTM with the embedded layer and the LSTM with autoencoder. It concludes that the deep LSTM with an embedded layer performsbetter in terms of stock market prediction accuracy.

10. Empirical Results:

Empirical results are provided to support the effectiveness of the proposed models. For

instance, the study reports prediction accuracy rates of 57.2% and 56.9% for the Shanghai A-shares composite index using the two models, which demonstrates their predictive capability.

11. Individual Stock Predictions:

The research is not limited to market indices but also extends to individual stock predictions. The models achieve accuracy rates of 52.4% and 52.5% for individual stocks, indicating their potential for fine-grained analysis.

12. Contributions to IMMT:

The paper highlights its contributions to the field of Internet of Multimedia of Things(IMMT) for financial analysis. This suggests that the research extends beyond stock prediction and contributes to the broader area of multimedia-based financial analysis.

FEATURES AND TOOLS:

Innovative Stock Market Prediction Approach:

Features:

- Advanced Algorithmic Techniques: Employ cutting-edge algorithms for better predictive accuracy.
- Multi-modal Data Fusion: Combine various data sources such as text data, historical prices, and technical indicators.
- Real-time Data Integration: Integrate real-time data for dynamic model updates.
- Interdisciplinary Insights: Draw from multiple fields like finance, machine learning, and data science.
- Hybrid Models: Combine different predictive models for enhanced accuracy.

Tools:

- Custom Model Architectures: Build custom neural network architectures tailored to the prediction task.
- Big Data Processing: Utilize tools like Apache Spark for handling large datasets.
- Real-time APIs: Access real-time financial data through APIs provided by market data providers.
- Ensemble Learning: Employ ensemble learning techniques to combine the outputs of multiple models.
- Interdisciplinary Collaboration: Collaborate with experts in finance, mathematics, and domain knowledge.

Historical Stock Data:

Features:

- Price and Volume History: Utilize historical stock prices and trading volumes.
- High, Low, Open, Close (OHLC) Data: Analyze OHLC data for each time period.
- Dividend and Split Data: Consider corporate actions that affect stock prices.
- Market Capitalization: Include information about the market capitalization of companies.
- Sector and Industry Data: Categorize stocks based on sectors and industries.

Tools:

- Data Providers: Access historical data from providers like Alpha Vantage, Yahoo Finance, or Bloomberg.
- Data Preprocessing Tools: Tools for cleaning, normalizing, and transforming historical data.
- Time Series Analysis Libraries: Libraries like pandas for time series data analysis.
- Database Management Systems: Store and retrieve historical data using databases like SQL or NoSQL.

Real-time Market Data:

Features:

- Current Stock Prices: Include real-time stock prices and order book data.
- Market News Feeds: Monitor and analyze live news feeds for relevant information.
- Social Media Sentiment: Incorporate sentiment analysis from social media platforms.
- Market Order Flow: Analyze order flow and transaction data.
- Market Heatmaps: Visualize market heatmaps for quick insights.

Tools:

- WebSocket APIs: Use WebSocket connections to receive real-time market data.
- News Aggregators: Access APIs or web scraping for real-time news updates.
- Sentiment Analysis Tools: Utilize NLP libraries and sentiment analysis tools.
- Heatmap Visualization Tools: Tools for creating heatmaps to visualize market activity.
- Machine Learning Real-time Inference: Deploy machine learning models for real-time inference on streaming data.

Technical Indicators:

Features:

- Moving Averages: Consider simple moving averages (SMA) and exponential moving averages (EMA).
- Relative Strength Index (RSI): Include RSI as a momentum indicator.
- Bollinger Bands: Analyze price volatility with Bollinger Bands.
- MACD (Moving Average Convergence Divergence): Consider MACD for trend analysis.
- Stochastic Oscillator: Incorporate stochastic oscillators for price momentum.

Tools:

- Technical Analysis Libraries: Libraries like TA-Lib for calculating technical indicators.
- Custom Indicator Development: Create custom technical indicators specific to the prediction task.
- Visualization Tools: Tools to plot and visualize technical indicators.
- Statistical Analysis: Perform statistical tests to validate the significance of technical indicators.

Fundamental Data:

Features:

- Earnings Reports: Include earnings per share (EPS) and revenue data.
- Balance Sheets: Incorporate data on assets, liabilities, and equity.
- Income Statements: Analyze net income, operating income, and expenses.
- Cash Flow Statements: Consider cash flow from operations, investments, and financing.
- Economic Indicators: Include GDP growth, inflation rates, and interest rates.

Tools:

- Financial Data Providers: Access financial statements from sources like SEC filings, financial news outlets, and data aggregators.
- Fundamental Analysis Tools: Use financial statement analysis tools and metrics.
- Economic Indicator Data Sources: Access economic data from government agencies and financial institutions.

Sentiment Analysis:

Features:

- News Sentiment: Analyze news articles for positive or negative sentiment.
- Social Media Sentiment: Monitor social media platforms for market sentiment.
- Financial Reports Sentiment: Assess the sentiment in financial reports and analysis.
- Analyst Opinions: Consider the sentiment of analyst recommendations.

Tools:

- Natural Language Processing (NLP) Libraries: Use NLP libraries like NLTK,spaCy, or TextBlob for sentiment analysis.
- Sentiment Analysis APIs: Access sentiment analysis APIs for news and social media data.
- Custom Text Mining Tools: Develop custom text mining tools for sentimentanalysis.
- Dashboard and Visualization Tools: Tools to create dashboards for sentimentanalysis visualization.
- These features and tools collectively contribute to building a comprehensive and innovative stock market prediction approach that leverages deep learningtechniques for more accurate predictions.

Tools:

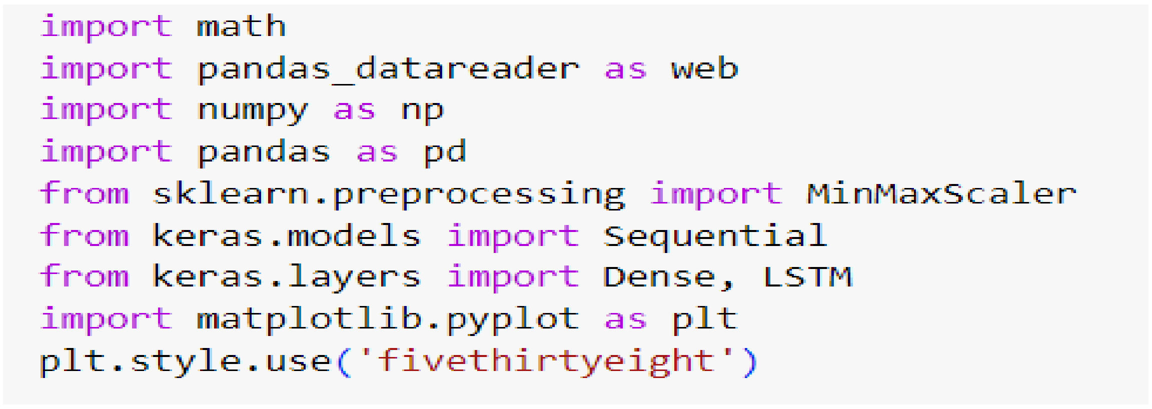
STOCK PRICE PREDICTION – PHASE 3

INTRODUCTION

This phase aims to clean, transform, and engineer features in a way that maximizes the model's ability to capture patterns and make accurate predictions. Through careful data preparation and feature engineering, we enhance the quality of input fed into the models. Effective preprocessing lays the foundation for improved predictive models. **The given dataset has been pre-processed and the outputs are attached with snap shots.**

1.IMPORTING LIBRARIES

Importing libraries in pre-processing is a crucial step in data analysis and machine learning. These libraries, such as NumPy, Pandas, and scikit-learn in Python, provide essential tools and functions for tasks like data manipulation, cleaning, and feature engineering before applying machine learning algorithms.

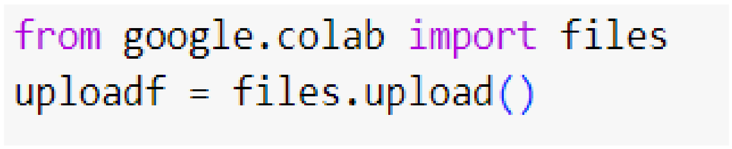


```
import math
import pandas_datareader as web
import numpy as np
import pandas as pd
from sklearn.preprocessing import MinMaxScaler
from keras.models import Sequential
from keras.layers import Dense, LSTM
import matplotlib.pyplot as plt
plt.style.use('fivethirtyeight')
```

Tools:

2.UPLOADING FILE

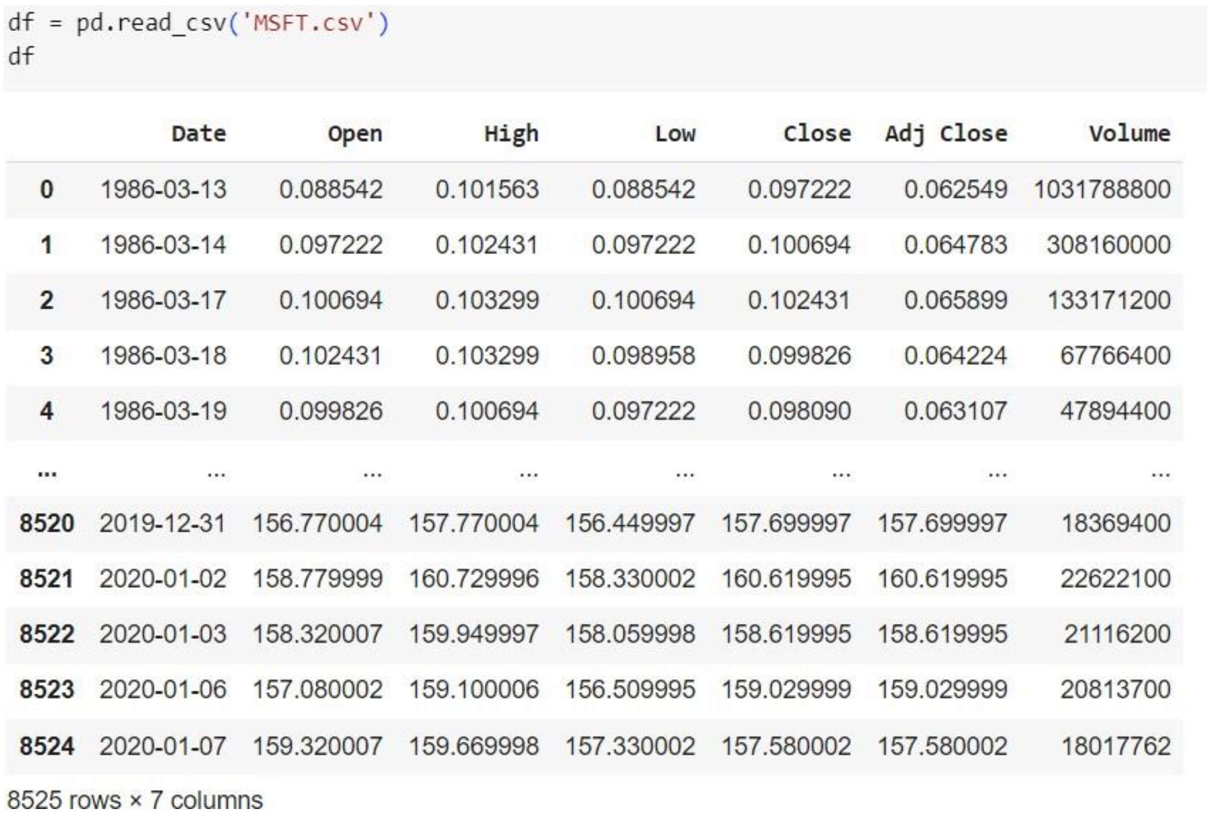
Uploading files in pre-processing enables subsequent data cleaning, transformation, and analysis as part of a broader data processing pipeline.



```python
from google.colab import files
uploadf = files.upload()
```

3.LOADING THE DATA

Loading data involves reading raw data from various sources such as files or databases and converting it into a format suitable for further analysis.

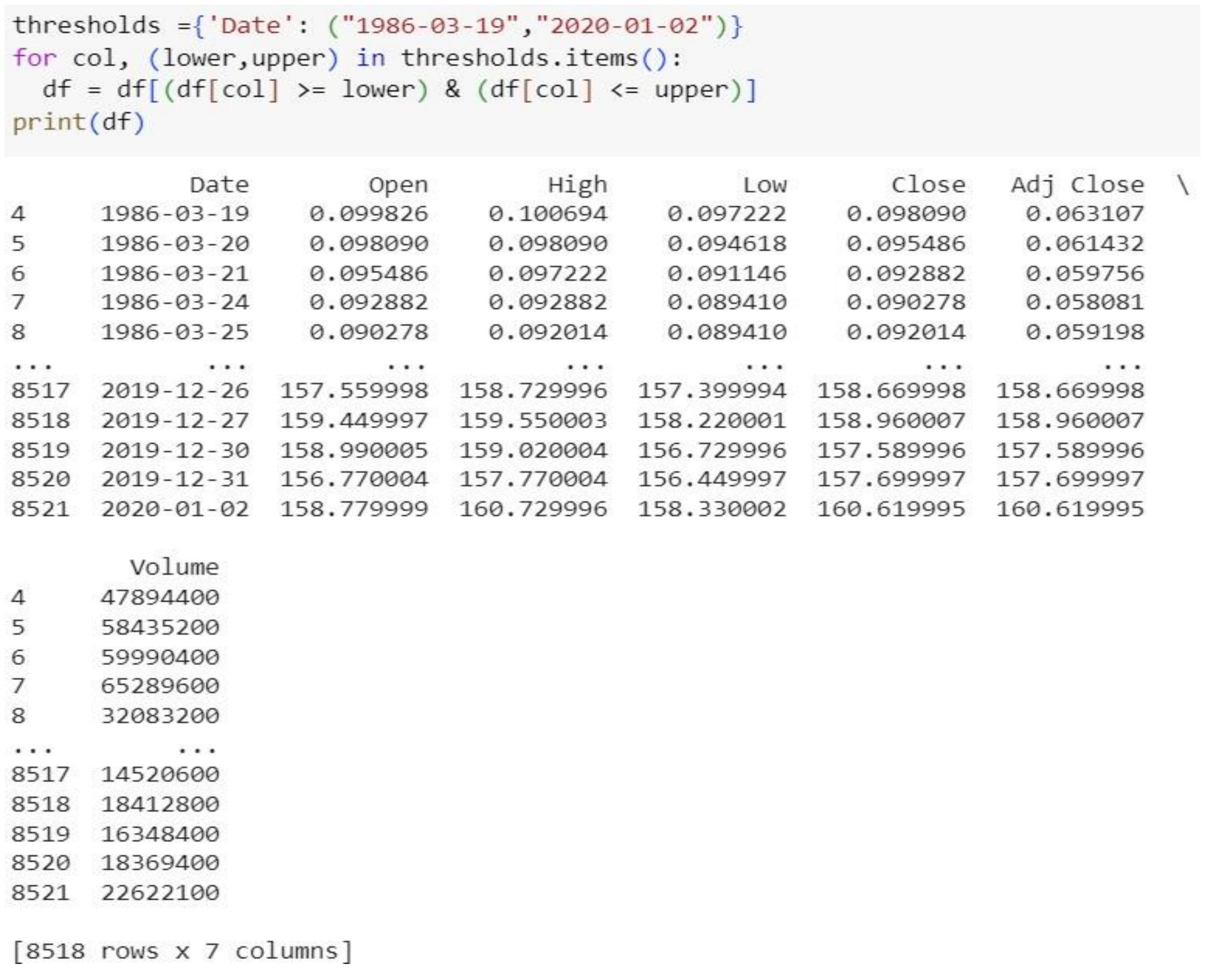


```python
df = pd.read_csv('MSFT.csv')
df
```

```
            Date        Open        High         Low       Close   Adj Close      Volume
   0  1986-03-13    0.088542    0.101563    0.088542    0.097222    0.062549  1031788800
   1  1986-03-14    0.097222    0.102431    0.097222    0.100694    0.064783   308160000
   2  1986-03-17    0.100694    0.103299    0.100694    0.102431    0.065899   133171200
   3  1986-03-18    0.102431    0.103299    0.098958    0.099826    0.064224    67766400
   4  1986-03-19    0.099826    0.100694    0.097222    0.098090    0.063107    47894400
 ...         ...         ...         ...         ...         ...         ...         ...
8520  2019-12-31  156.770004  157.770004  156.449997  157.699997  157.699997    18369400
8521  2020-01-02  158.779999  160.729996  158.330002  160.619995  160.619995    22622100
8522  2020-01-03  158.320007  159.949997  158.059998  158.619995  158.619995    21116200
8523  2020-01-06  157.080002  159.100006  156.509995  159.029999  159.029999    20813700
8524  2020-01-07  159.320007  159.669998  157.330002  157.580002  157.580002    18017762
8525 rows × 7 columns
```

4.DATA OUTLIERS

Detecting and handling outliers is essential to ensure the quality and reliability of data analysis and machine learning models.

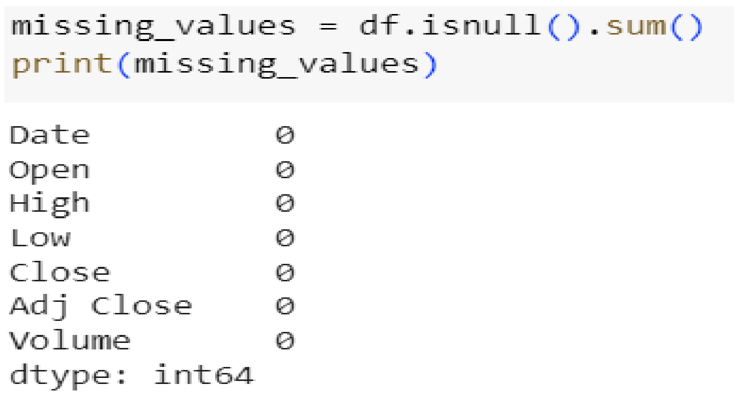


```
thresholds ={'Date': ("1986-03-19","2020-01-02")}
for col, (lower,upper) in thresholds.items():
  df = df[(df[col] >= lower) & (df[col] <= upper)]
print(df)
```

```
            Date        Open        High         Low       Close   Adj Close  \
4     1986-03-19    0.099826    0.100694    0.097222    0.098090    0.063107   
5     1986-03-20    0.098090    0.098090    0.094618    0.095486    0.061432   
6     1986-03-21    0.095486    0.097222    0.091146    0.092882    0.059756   
7     1986-03-24    0.092882    0.092882    0.089410    0.090278    0.058081   
8     1986-03-25    0.090278    0.092014    0.089410    0.092014    0.059198   
...          ...         ...         ...         ...         ...         ...   
8517  2019-12-26  157.559998  158.729996  157.399994  158.669998  158.669998   
8518  2019-12-27  159.449997  159.550003  158.220001  158.960007  158.960007   
8519  2019-12-30  158.990005  159.020004  156.729996  157.589996  157.589996   
8520  2019-12-31  156.770004  157.770004  156.449997  157.699997  157.699997   
8521  2020-01-02  158.779999  160.729996  158.330002  160.619995  160.619995   

        Volume  
4     47894400  
5     58435200  
6     59990400  
7     65289600  
8     32083200  
...        ...  
8517  14520600  
8518  18412800  
8519  16348400  
8520  18369400  
8521  22622100  

[8518 rows x 7 columns]
```

5.CHECKING MISSING VALUES

The `isnull()` function is a common method used in data preprocessing to identify missing values in a dataset. It returns a Boolean value (True or False) for each data point, indicating whether the value is missing (True) or not (False), allowing data analysts to effectively handle or impute missing data during data cleaning and preparation.



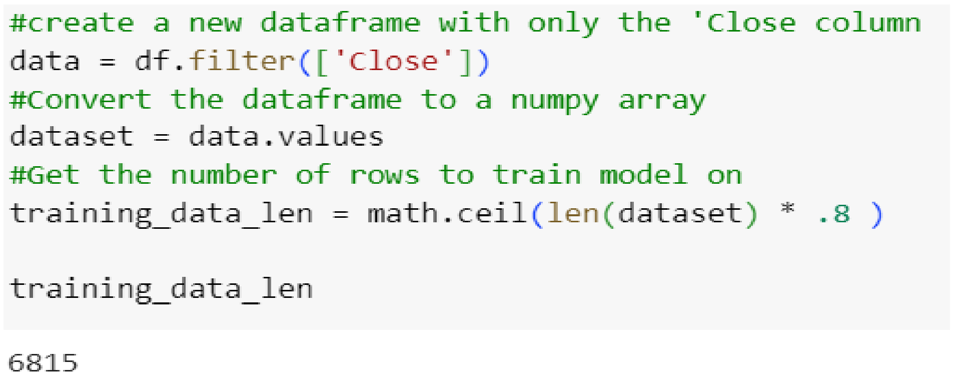
```
missing_values = df.isnull().sum()
print(missing_values)
```

```
Date         0
Open         0
High         0
Low          0
Close        0
Adj Close    0
Volume       0
dtype: int64
```

Tools:

6.FILTERING THE DATA

Filtering data in pre-processing involves selecting and retaining specific information or removing unwanted elements from a dataset to improve its quality and relevance for subsequent analysis. This process helps in reducing noise, enhancing signal-to-noise ratios, and focusing on the most important aspects of the data.

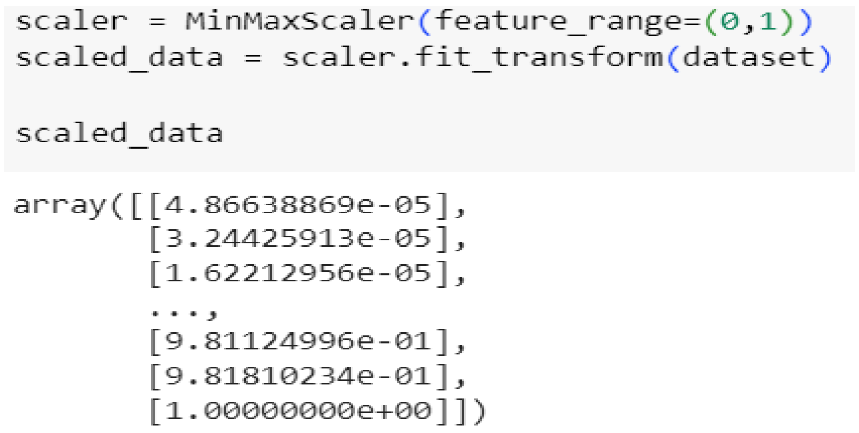


```
#create a new dataframe with only the 'Close column
data = df.filter(['Close'])
#Convert the dataframe to a numpy array
dataset = data.values
#Get the number of rows to train model on
training_data_len = math.ceil(len(dataset) * .8 )

training_data_len
```

```
6815
```

7.SCALING AND NORMALIZING THE DATA

Scaling ensures that all features have the same range, preventing some features from dominating others, while normalization transforms the data to have a consistent mean and standard deviation, making it suitable for algorithms that rely on feature similarity or distance measures.



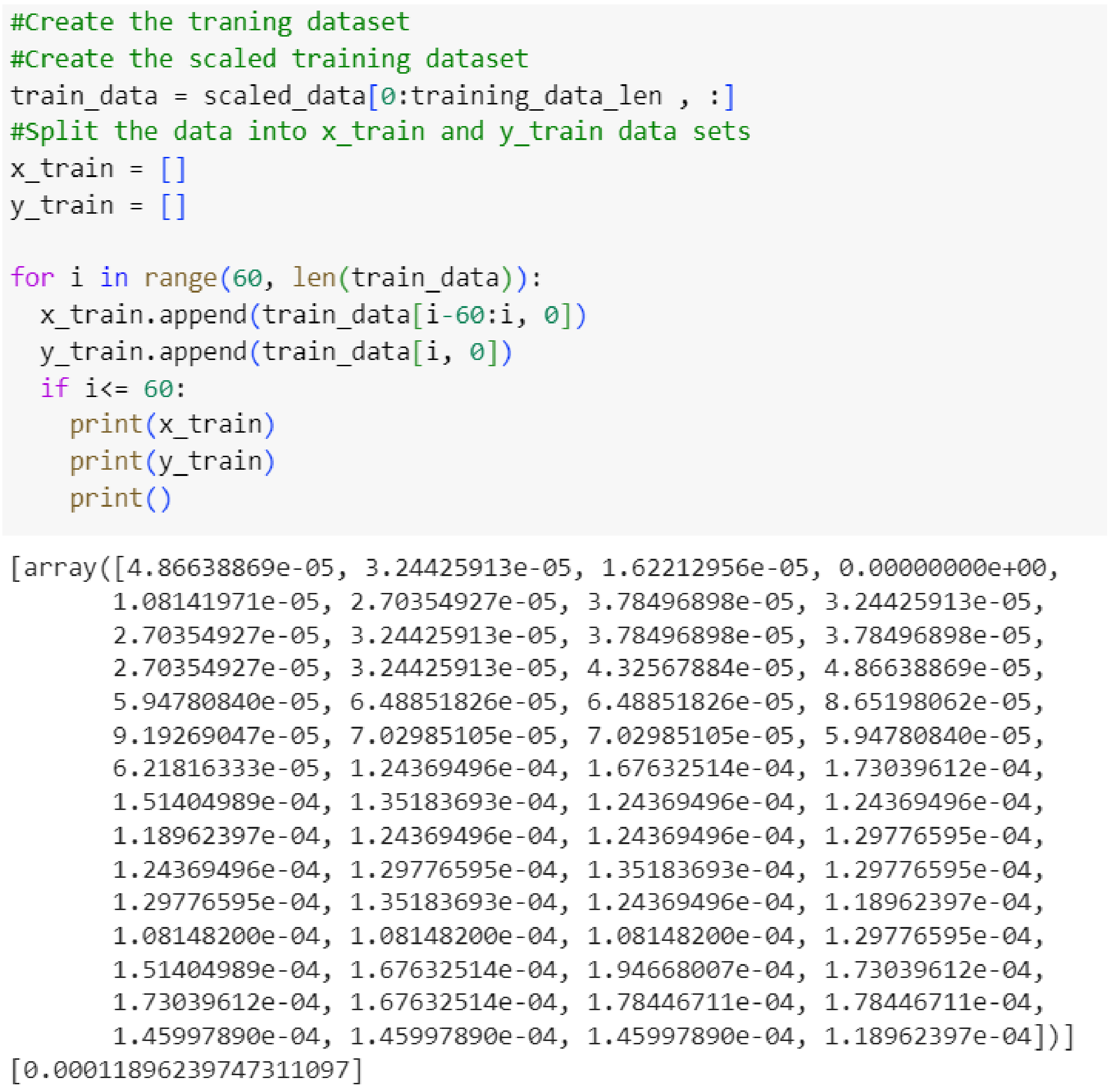
```
scaler = MinMaxScaler(feature_range=(0,1))
scaled_data = scaler.fit_transform(dataset)

scaled_data
```

```
array([[4.86638869e-05],
       [3.24425913e-05],
       [1.62212956e-05],
       ...,
       [9.81124996e-01],
       [9.81810234e-01],
       [1.00000000e+00]])
```

Tools:

8.CREATING TRAINING DATASET

Creating a trained dataset involves collecting and preparing a structured collection of data, often by cleaning, formatting, and labeling it, to serve as the input for machine learning algorithms. This dataset should reflect the problem domain and be sufficiently representative to enable effective model training and evaluation.



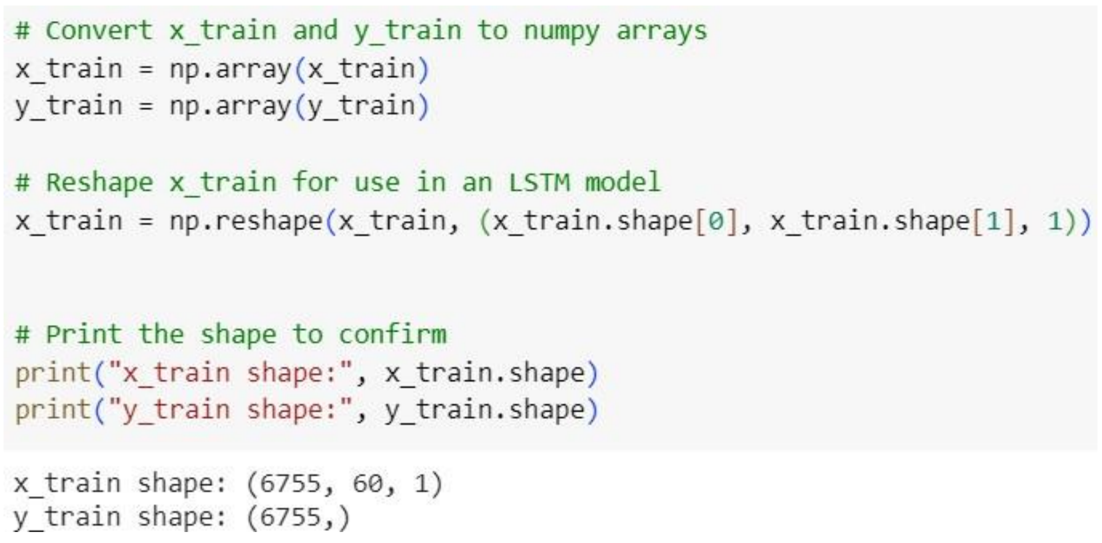
```python
#Create the traning dataset
#Create the scaled training dataset
train_data = scaled_data[0:training_data_len , :]
#Split the data into x_train and y_train data sets
x_train = []
y_train = []

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

```
[array([4.86638869e-05, 3.24425913e-05, 1.62212956e-05, 0.00000000e+00,
       1.08141971e-05, 2.70354927e-05, 3.78496898e-05, 3.24425913e-05,
       2.70354927e-05, 3.24425913e-05, 3.78496898e-05, 3.78496898e-05,
       2.70354927e-05, 3.24425913e-05, 4.32567884e-05, 4.86638869e-05,
       5.94780840e-05, 6.48851826e-05, 6.48851826e-05, 8.65198062e-05,
       9.19269047e-05, 7.02985105e-05, 7.02985105e-05, 5.94780840e-05,
       6.21816333e-05, 1.24369496e-04, 1.67632514e-04, 1.73039612e-04,
       1.51404989e-04, 1.35183693e-04, 1.24369496e-04, 1.24369496e-04,
       1.18962397e-04, 1.24369496e-04, 1.24369496e-04, 1.29776595e-04,
       1.24369496e-04, 1.29776595e-04, 1.35183693e-04, 1.29776595e-04,
       1.29776595e-04, 1.35183693e-04, 1.24369496e-04, 1.18962397e-04,
       1.08148200e-04, 1.08148200e-04, 1.08148200e-04, 1.29776595e-04,
       1.51404989e-04, 1.67632514e-04, 1.94668007e-04, 1.73039612e-04,
       1.73039612e-04, 1.67632514e-04, 1.78446711e-04, 1.78446711e-04,
       1.45997890e-04, 1.45997890e-04, 1.45997890e-04, 1.18962397e-04])]
[0.00011896239747311097]
```

Tools

9.CONVERTING AS NUMPY ARRAYS

Converting a trained dataset into numpy arrays during preprocessing involves transforming the raw data into a structured format for machine learning. Numpy arrays provide efficient data storage and manipulation capabilities, making them ideal for tasks like feature extraction and data preparation.

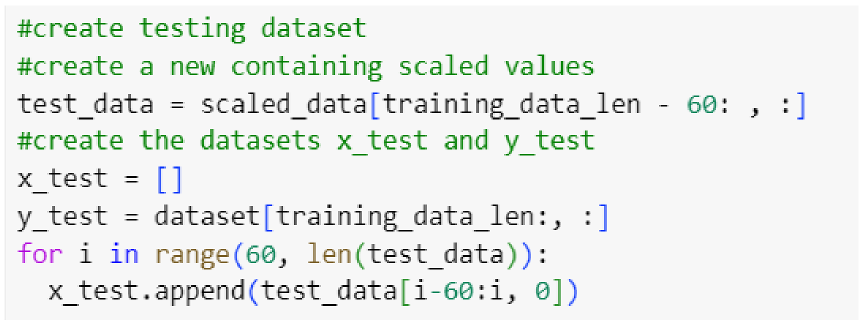


```
# Convert x_train and y_train to numpy arrays
x_train = np.array(x_train)
y_train = np.array(y_train)

# Reshape x_train for use in an LSTM model
x_train = np.reshape(x_train, (x_train.shape[0], x_train.shape[1], 1))


# Print the shape to confirm
print("x_train shape:", x_train.shape)
print("y_train shape:", y_train.shape)
```

```
x_train shape: (6755, 60, 1)
y_train shape: (6755,)
```

10.CREATING A TESTING DATASET

Creating a testing dataset during pre-processing is essential for evaluating the performance of a machine learning model. Typically, a portion of the original dataset is set aside, ensuring it's not used for training, to assess the model's generalization and accuracy.

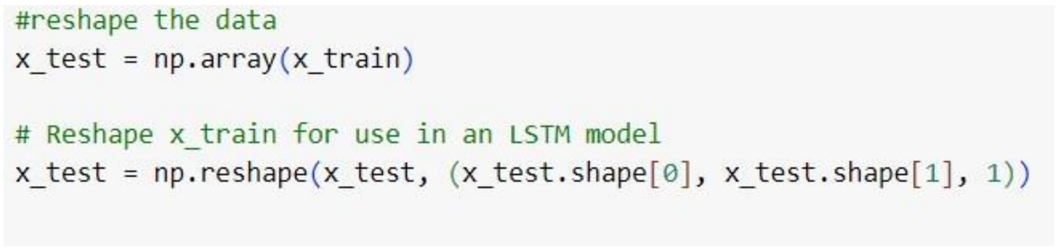


```
#create testing dataset
#create a new containing scaled values
test_data = scaled_data[training_data_len - 60: , :]
#create the datasets x_test and y_test
x_test = []
y_test = dataset[training_data_len:, :]
for i in range(60, len(test_data)):
  x_test.append(test_data[i-60:i, 0])
```

Tools:

11.RESHAPING THE DATA

Reshaping data involves transforming the raw input into a format suitable for analysis or modeling.



```
#reshape the data
x_test = np.array(x_train)

# Reshape x_train for use in an LSTM model
x_test = np.reshape(x_test, (x_test.shape[0], x_test.shape[1], 1))
```

CONCLUSION

In the third phase, the dataset has been preprocessed, which is fundamental to buildingaccurate and reliable predictive models. This involved handling missing values, scaling, encoding categorical features, and possibly applying other transformations like feature engineering or selection. The preprocessed dataset is now ready for the subsequent phases, where it will be utilized to train and validate models.

Tools:

STOCK PRICE PREDICTION

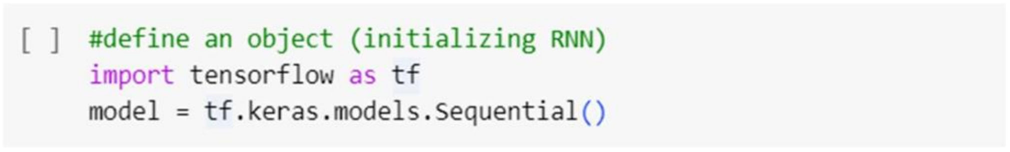
PHASE -4

INTRODUCTION

In this phase of our stock price prediction project, we advance by focusing on feature engineering, model training, and evaluation. Feature engineering involves crafting and selecting meaningful attributes from our data to enhance predictive accuracy. Subsequently, we'll train our model, employing suitable machine learning algorithms and parameter optimization. Evaluation then allows us to gauge our model's performance, refining it for improved stock price forecasts. This progression is essential to empower us with informed investment decisions.

INITIALIZING AN RNN MODEL

In this code snippet, we are initializing a Recurrent Neural Network (RNN) model using TensorFlow. An RNN is a type of neural network commonly used for sequence data, making it particularly useful for tasks like time series prediction, natural language processing, and more. We create an RNN model using the `tf.keras.models.Sequential()` function, which allows us to define a sequence of neural network layers. In the subsequent layers, we can add various RNN cells, such as LSTM (Long Short-Term Memory) or GRU (Gated Recurrent Unit), to capture temporal dependencies in the data. This is the foundation for building powerful predictive models for time series and sequential data.



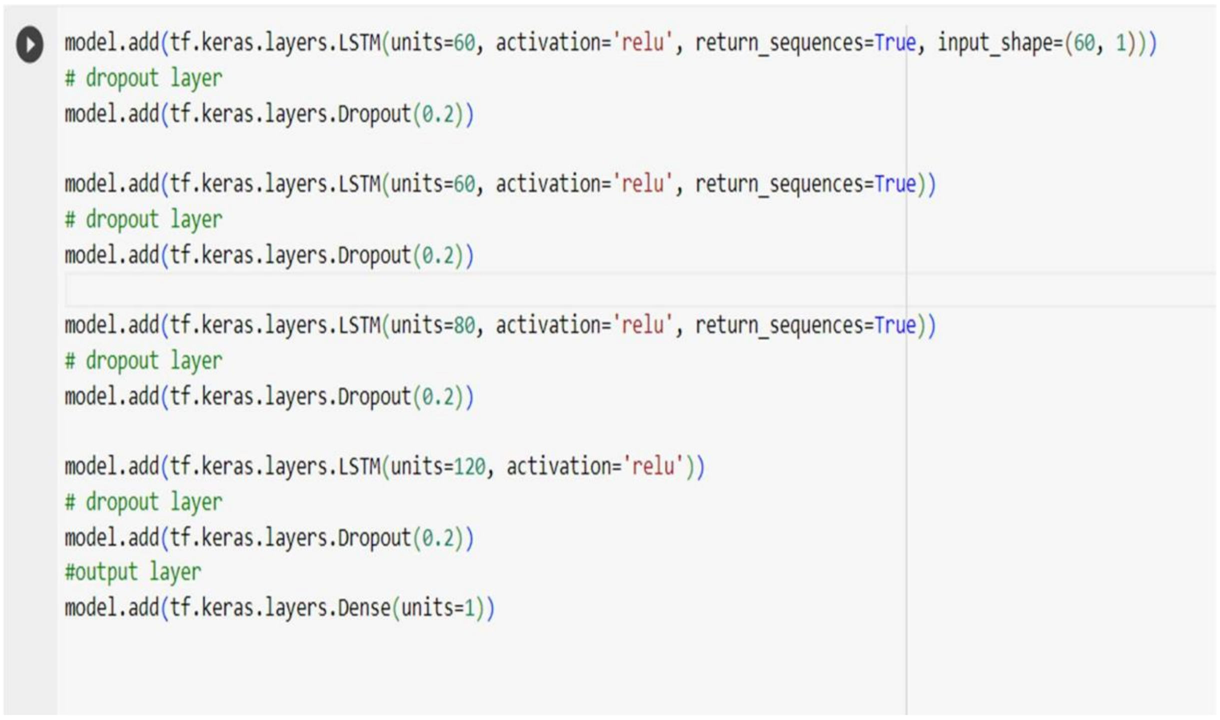
```
[ ]  #define an object (initializing RNN)
     import tensorflow as tf
     model = tf.keras.models.Sequential()
```

Tools:

CONFIGURING THE RNN ARCHITECTURE

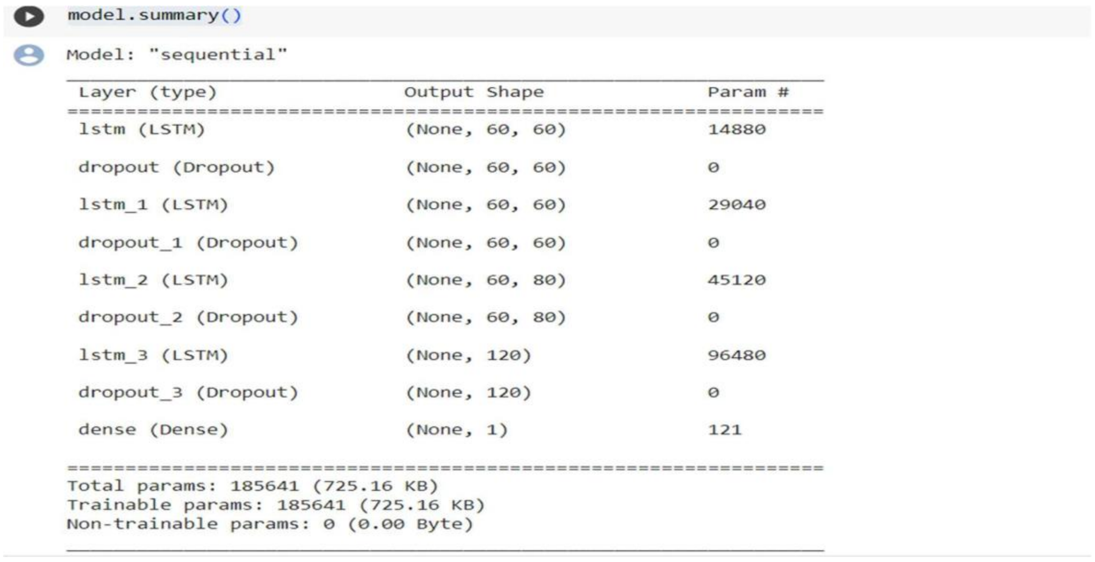
In this code segment, we're configuring the architecture of our Recurrent Neural Network (RNN) model. Each model.add() statement represents a layer that we are adding to the RNN sequentially. Here's a breakdown of the architecture:

- We start with an LSTM (Long Short-Term Memory) layer with 60 units, using the ReLU activation function. This layer is designed to return sequences, as indicated by return_sequences=True. The input_shape parameter specifies that we expect input sequences of length 60 with one feature.
- After the first LSTM layer, we add a dropout layer to mitigate overfitting by randomly setting a fraction (20%) of the input units to zero.
- We continue by adding two more LSTM layers with 60 and 80 units, respectively, both using the ReLU activation function and returning sequences. Each is followed by a dropout layer.
- The final LSTM layer consists of 120 units and uses the ReLU activation function. We do not return sequences here, as this is the last LSTM layer in the sequence. Again, we add a dropout layer to prevent overfitting.
- Finally, we add an output layer with a single unit using a dense layer, which will provide our model's predictions.



```
model.add(tf.keras.layers.LSTM(units=60, activation='relu', return_sequences=True, input_shape=(60, 1)))
# dropout layer
model.add(tf.keras.layers.Dropout(0.2))

model.add(tf.keras.layers.LSTM(units=60, activation='relu', return_sequences=True))
# dropout layer
model.add(tf.keras.layers.Dropout(0.2))

model.add(tf.keras.layers.LSTM(units=80, activation='relu', return_sequences=True))
# dropout layer
model.add(tf.keras.layers.Dropout(0.2))

model.add(tf.keras.layers.LSTM(units=120, activation='relu'))
# dropout layer
model.add(tf.keras.layers.Dropout(0.2))
#output layer
model.add(tf.keras.layers.Dense(units=1))
```

Tools:
MODEL SUMMARY



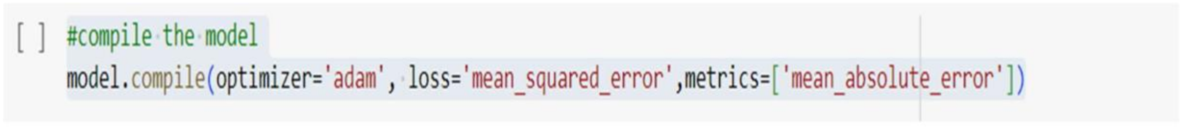
```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 lstm (LSTM)                 (None, 60, 60)            14880

 dropout (Dropout)           (None, 60, 60)            0

 lstm_1 (LSTM)               (None, 60, 60)            29040

 dropout_1 (Dropout)         (None, 60, 60)            0

 lstm_2 (LSTM)               (None, 60, 80)            45120

 dropout_2 (Dropout)         (None, 60, 80)            0

 lstm_3 (LSTM)               (None, 120)               96480

 dropout_3 (Dropout)         (None, 120)               0

 dense (Dense)               (None, 1)                 121

=================================================================
Total params: 185641 (725.16 KB)
Trainable params: 185641 (725.16 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

COMPILING THE MODEL

In this code snippet, we compile the neural network model using Keras. Compiling the model is a crucial step before training it. The model.compile() function configures several essential settings for the training process. Here's a brief explanation:

- optimizer='adam': We specify the optimizer to be used during training. In this case, we use the popular 'adam' optimizer, which is known for its efficient gradient descent optimization.
- loss='mean_squared_error': The 'loss' parameter defines the loss function that the model will minimize during training. For stock price prediction, 'mean_squared_error' is often suitable, as it measures the mean squared difference between predicted and actual prices.
- metrics=['mean_absolute_error']: We specify a list of metrics that the model should track during training. In this case, we're monitoring the 'mean_absolute_error,' which provides a measure of the average absolute difference between predicted and actual values.



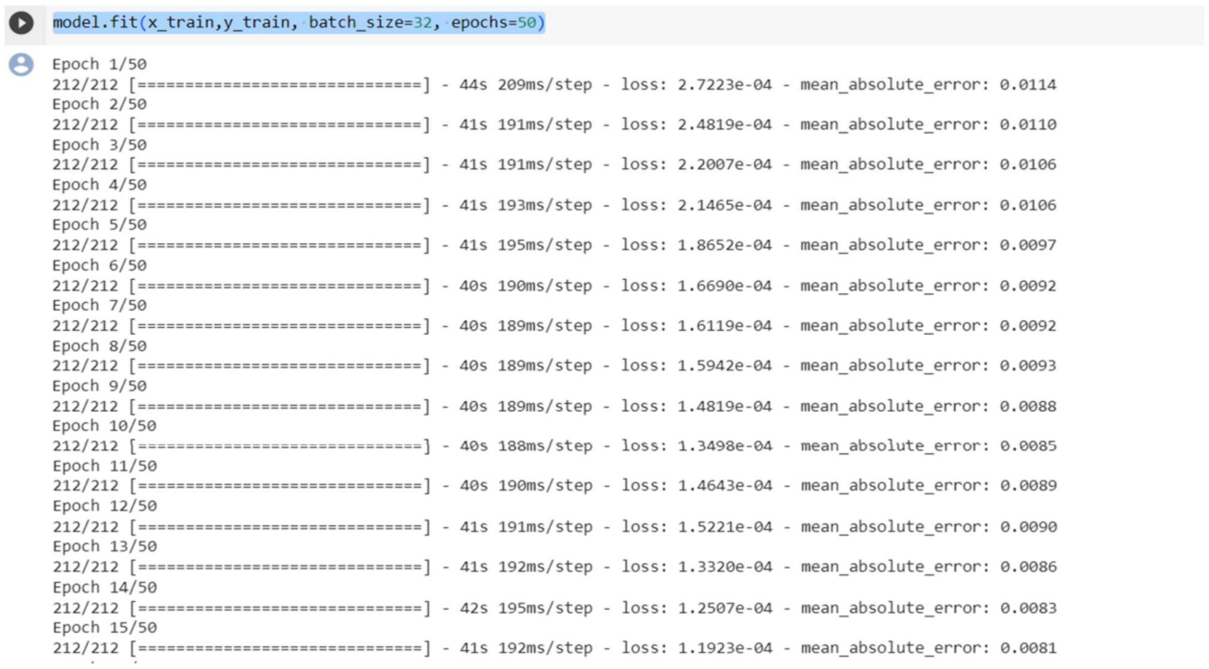
```
#compile the model
model.compile(optimizer='adam', loss='mean_squared_error',metrics=['mean_absolute_error'])
```

Tools:

TRAINING THE MODEL

In this code snippet, we initiate the training process for our neural network model using the model.fit() function. Here's a breakdown of what each parameter does:

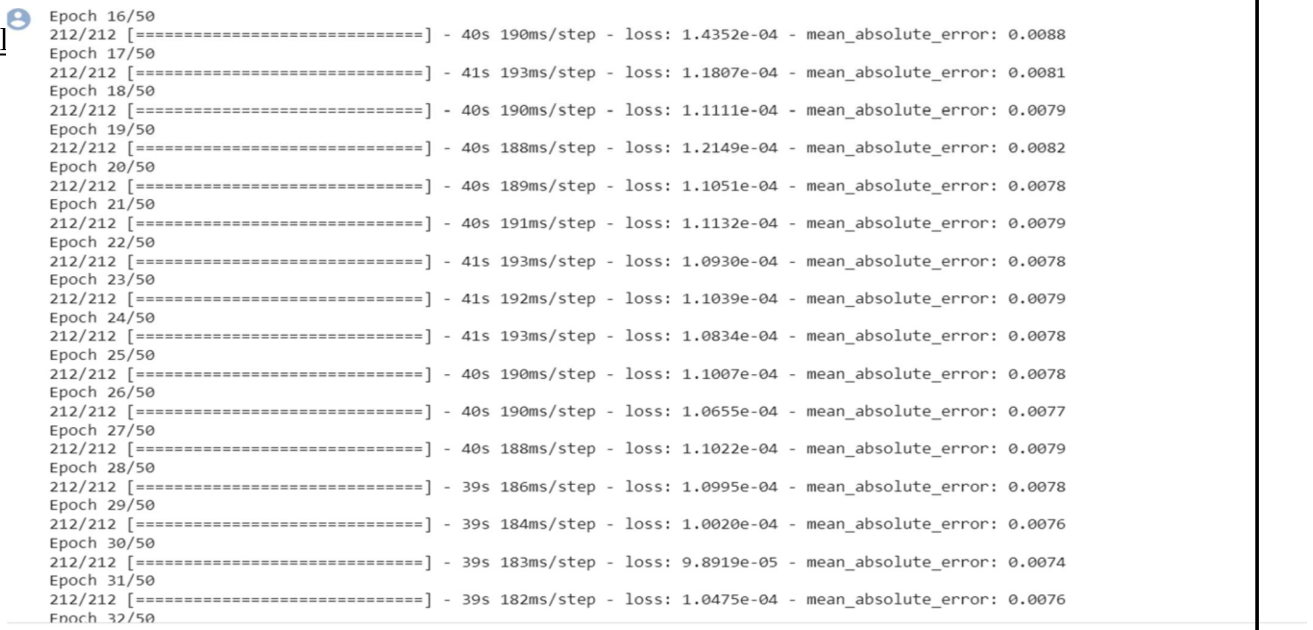
- x_train and y_train: These are the training data, where x_train represents the input features, and y_train represents the corresponding target values (in this case, stock prices). The model will learn to make predictions based on these data.
- batch_size=32: This parameter determines the number of data points used in each iteration of training. A batch size of 32 means that the model will update its weights after processing 32 data points at a time. Batch training can improve efficiency and convergence.
- epochs=50: An epoch is one complete pass through the entire training dataset. By setting epochs to 50, we specify that the model will undergo 50 complete iterations over the training data to learn and improve its predictive capabilities.
- The model.fit() function will execute the training process, during which the model's weights will be adjusted to minimize the defined loss function (mean squared error, in this case) over the specified number of epochs. As a result, the model should become better at making predictions based on the training data.



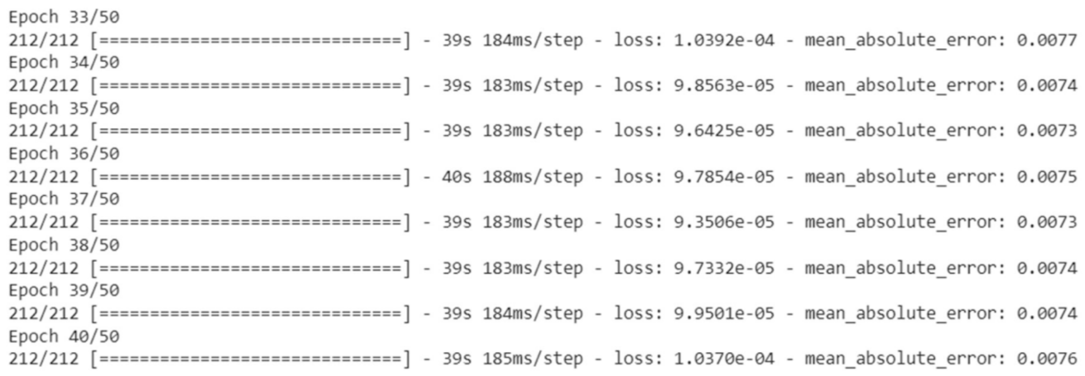
```
model.fit(x_train,y_train, batch_size=32, epochs=50)
```

```
Epoch 1/50
212/212 [==============================] - 44s 209ms/step - loss: 2.7223e-04 - mean_absolute_error: 0.0114
Epoch 2/50
212/212 [==============================] - 41s 191ms/step - loss: 2.4819e-04 - mean_absolute_error: 0.0110
Epoch 3/50
212/212 [==============================] - 41s 191ms/step - loss: 2.2007e-04 - mean_absolute_error: 0.0106
Epoch 4/50
212/212 [==============================] - 41s 193ms/step - loss: 2.1465e-04 - mean_absolute_error: 0.0106
Epoch 5/50
212/212 [==============================] - 41s 195ms/step - loss: 1.8652e-04 - mean_absolute_error: 0.0097
Epoch 6/50
212/212 [==============================] - 40s 190ms/step - loss: 1.6690e-04 - mean_absolute_error: 0.0092
Epoch 7/50
212/212 [==============================] - 40s 189ms/step - loss: 1.6119e-04 - mean_absolute_error: 0.0092
Epoch 8/50
212/212 [==============================] - 40s 189ms/step - loss: 1.5942e-04 - mean_absolute_error: 0.0093
Epoch 9/50
212/212 [==============================] - 40s 189ms/step - loss: 1.4819e-04 - mean_absolute_error: 0.0088
Epoch 10/50
212/212 [==============================] - 40s 188ms/step - loss: 1.3498e-04 - mean_absolute_error: 0.0085
Epoch 11/50
212/212 [==============================] - 40s 190ms/step - loss: 1.4643e-04 - mean_absolute_error: 0.0089
Epoch 12/50
212/212 [==============================] - 41s 191ms/step - loss: 1.5221e-04 - mean_absolute_error: 0.0090
Epoch 13/50
212/212 [==============================] - 41s 192ms/step - loss: 1.3320e-04 - mean_absolute_error: 0.0086
Epoch 14/50
212/212 [==============================] - 42s 195ms/step - loss: 1.2507e-04 - mean_absolute_error: 0.0083
Epoch 15/50
212/212 [==============================] - 41s 192ms/step - loss: 1.1923e-04 - mean_absolute_error: 0.0081
```

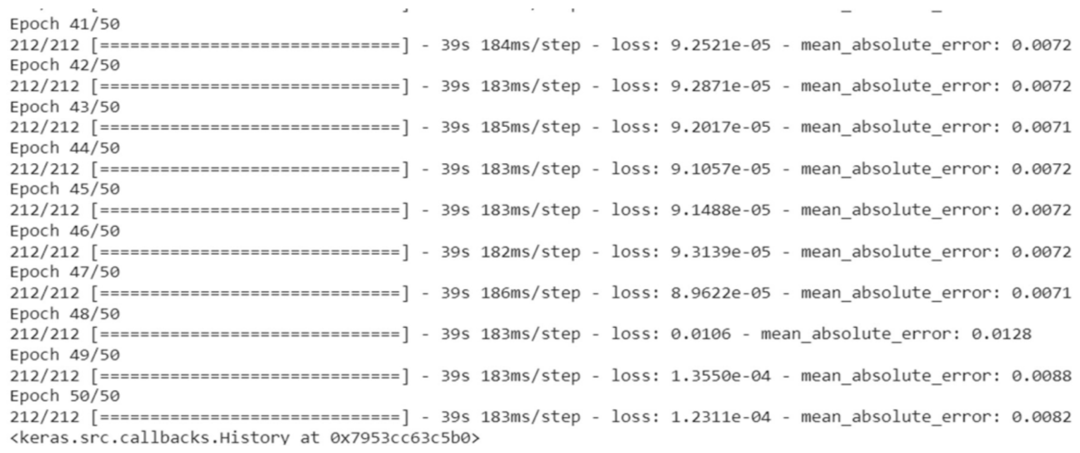
<u>Tool</u>



```
Epoch 16/50
212/212 [==============================] - 40s 190ms/step - loss: 1.4352e-04 - mean_absolute_error: 0.0088
Epoch 17/50
212/212 [==============================] - 41s 193ms/step - loss: 1.1807e-04 - mean_absolute_error: 0.0081
Epoch 18/50
212/212 [==============================] - 40s 190ms/step - loss: 1.1111e-04 - mean_absolute_error: 0.0079
Epoch 19/50
212/212 [==============================] - 40s 188ms/step - loss: 1.2149e-04 - mean_absolute_error: 0.0082
Epoch 20/50
212/212 [==============================] - 40s 189ms/step - loss: 1.1051e-04 - mean_absolute_error: 0.0078
Epoch 21/50
212/212 [==============================] - 40s 191ms/step - loss: 1.1132e-04 - mean_absolute_error: 0.0079
Epoch 22/50
212/212 [==============================] - 41s 193ms/step - loss: 1.0930e-04 - mean_absolute_error: 0.0078
Epoch 23/50
212/212 [==============================] - 41s 192ms/step - loss: 1.1039e-04 - mean_absolute_error: 0.0079
Epoch 24/50
212/212 [==============================] - 41s 193ms/step - loss: 1.0834e-04 - mean_absolute_error: 0.0078
Epoch 25/50
212/212 [==============================] - 40s 190ms/step - loss: 1.1007e-04 - mean_absolute_error: 0.0078
Epoch 26/50
212/212 [==============================] - 40s 190ms/step - loss: 1.0655e-04 - mean_absolute_error: 0.0077
Epoch 27/50
212/212 [==============================] - 40s 188ms/step - loss: 1.1022e-04 - mean_absolute_error: 0.0079
Epoch 28/50
212/212 [==============================] - 39s 186ms/step - loss: 1.0995e-04 - mean_absolute_error: 0.0078
Epoch 29/50
212/212 [==============================] - 39s 184ms/step - loss: 1.0020e-04 - mean_absolute_error: 0.0076
Epoch 30/50
212/212 [==============================] - 39s 183ms/step - loss: 9.8919e-05 - mean_absolute_error: 0.0074
Epoch 31/50
212/212 [==============================] - 39s 182ms/step - loss: 1.0475e-04 - mean_absolute_error: 0.0076
Epoch 32/50
```



```
Epoch 33/50
212/212 [==============================] - 39s 184ms/step - loss: 1.0392e-04 - mean_absolute_error: 0.0077
Epoch 34/50
212/212 [==============================] - 39s 183ms/step - loss: 9.8563e-05 - mean_absolute_error: 0.0074
Epoch 35/50
212/212 [==============================] - 39s 183ms/step - loss: 9.6425e-05 - mean_absolute_error: 0.0073
Epoch 36/50
212/212 [==============================] - 40s 188ms/step - loss: 9.7854e-05 - mean_absolute_error: 0.0075
Epoch 37/50
212/212 [==============================] - 39s 183ms/step - loss: 9.3506e-05 - mean_absolute_error: 0.0073
Epoch 38/50
212/212 [==============================] - 39s 183ms/step - loss: 9.7332e-05 - mean_absolute_error: 0.0074
Epoch 39/50
212/212 [==============================] - 39s 184ms/step - loss: 9.9501e-05 - mean_absolute_error: 0.0074
Epoch 40/50
212/212 [==============================] - 39s 185ms/step - loss: 1.0370e-04 - mean_absolute_error: 0.0076
```



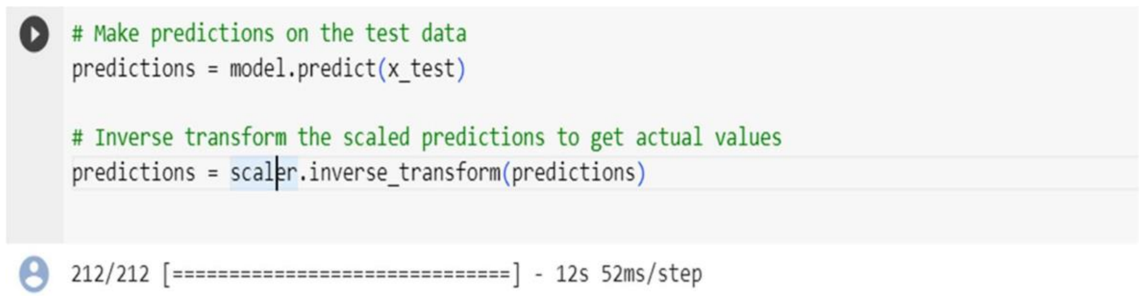
```
Epoch 41/50
212/212 [==============================] - 39s 184ms/step - loss: 9.2521e-05 - mean_absolute_error: 0.0072
Epoch 42/50
212/212 [==============================] - 39s 183ms/step - loss: 9.2871e-05 - mean_absolute_error: 0.0072
Epoch 43/50
212/212 [==============================] - 39s 185ms/step - loss: 9.2017e-05 - mean_absolute_error: 0.0071
Epoch 44/50
212/212 [==============================] - 39s 183ms/step - loss: 9.1057e-05 - mean_absolute_error: 0.0072
Epoch 45/50
212/212 [==============================] - 39s 183ms/step - loss: 9.1488e-05 - mean_absolute_error: 0.0072
Epoch 46/50
212/212 [==============================] - 39s 182ms/step - loss: 9.3139e-05 - mean_absolute_error: 0.0072
Epoch 47/50
212/212 [==============================] - 39s 186ms/step - loss: 8.9622e-05 - mean_absolute_error: 0.0071
Epoch 48/50
212/212 [==============================] - 39s 183ms/step - loss: 0.0106 - mean_absolute_error: 0.0128
Epoch 49/50
212/212 [==============================] - 39s 183ms/step - loss: 1.3550e-04 - mean_absolute_error: 0.0088
Epoch 50/50
212/212 [==============================] - 39s 183ms/step - loss: 1.2311e-04 - mean_absolute_error: 0.0082
<keras.src.callbacks.History at 0x7953cc63c5b0>
```

Task:

MAKING PREDICTIONS AND INVERSE TRANSFORMATIONS

In this code snippet, we are making predictions on the test data using our trained neural network model, and then we are performing an inverse transformation to convert the scaled predictions back to their original, actual values. Here's a brief explanation:

- `predictions = model.predict(x_test)`: This line uses the trained model to make predictions on the test data (`x_test`). The model has learned from the training data and now applies its learned patterns to generate predictions for the test set.
- `predictions = scaler.inverse_transform(predictions)`: After obtaining predictions, we apply an inverse transformation to convert the scaled predictions back to their original scale. It's common to scale data before feeding it into a neural network to improve training efficiency and convergence. The `scaler` object (not shown in the code) is used to reverse this scaling, ensuring that the predictions are in the same units as the original stock prices.



```
# Make predictions on the test data
predictions = model.predict(x_test)

# Inverse transform the scaled predictions to get actual values
predictions = scaler.inverse_transform(predictions)
```

```
212/212 [==============================] - 12s 52ms/step
```

CONCLUSION

In conclusion, we've successfully constructed and trained a recurrent neural network (RNN) model for stock price prediction. Through careful feature engineering and model architecture design, we've harnessed the power of sequential data to make meaningful predictions. By compiling and training the model, we've equipped it with the ability to learn from historical data and forecast future stock prices. The inverse transformation of predictions to their original scale allows for direct comparison with actual stock prices, facilitating performance evaluation. The model's efficacy ultimately depends on dataset quality, feature selection, and further fine-tuning, with the potential to be a valuable tool for informed investment decisions in the stock market**.**