

STOCK PRICE PREDICTION

PROBLEM DEFINITION:

Predicting stock prices is a complex and challenging problem in applied data science and finance. The goal is to develop models that can accurately forecast the future prices of stocks or other financial assets. The task is to develop predictive models to forecast future stock prices for a given set of financial assets based on historical stock price data, contextual information, and relevant features from the years 1986 to 2020.

The Kaggle dataset context provides insights on accurate stock price predictions that are essential for investors, traders, and financial professionals to make informed decisions about buying or selling financial assets. This problem leverages data science techniques to create predictive solutions that assist in portfolio optimization, risk management, and profit maximization within the historical time frame of 1986-2020.

Data: The primary data sources include historical stock price data, trading volumes, open price, close price, economic indicators and dates, and news sentiment scores covering the period from 1986 to 2020.

Task: Creating machine learning or predictive algorithms that provide reliable estimates of future stock prices within the specified 1986–2020-time frame, with the objective of minimizing prediction errors.

Evaluation Metrics: The performance of stock price prediction models can be assessed using various metrics. This has been referred as the methods that would predict it with certain accuracy.

Mean Absolute Error (MAE): This metric measures the average absolute difference between the predicted and actual stock prices.

Mean Squared Error (MSE): This metric quantifies the average squared difference between predicted and actual prices, giving more weight to larger errors.

Root Mean Squared Error (RMSE): RMSE is the square root of the MSE, providing a measure of the average magnitude of prediction errors in the same unit as the target variable.

R-squared (R^2) Score: R-squared measures the proportion of the variance in the stock price that is explained by the model. A higher R^2 score indicates a better fit.

Feature engineering can help transform data to be more stationary, making it easier for models to capture trends and patterns because raw stock price data alone may not be sufficient to make accurate predictions.

Challenges specific to this time frame include dealing with long-term historical data, changes in market over time, and the need to account for economic events that occurred within the 1986-2020 period.

Stock price prediction is a complex problem in applied data science, with numerous challenges and approaches. It requires a combination of domain knowledge, data analysis, feature engineering, and machine learning techniques to build accurate and effective predictive models.

DESIGN THINKING:

Applied Data science plays a pivotal role in stock price prediction by providing the tools and techniques to analyse, model, and make decisions based on historical data and relevant information. Here, we have designed an approach to make this prediction on stock price accurately using the Kaggle dataset covering the years 1986 to 2020.

<https://www.kaggle.com/datasets/prasoonkottarathil/microsoft-lifetime-stocks-dataset>

We have provided a step-by-step guide based on our knowledge that we obtained from this course so far and the provided dataset.

Step 1: Data Acquisition (provided dataset)

The dataset obtained from Kaggle, includes historical stock price data, dates, trading volumes, open price, close price and economic indicators and make sure the dataset covers the period from 1986 to 2020.

Step 2: Data Exploration and Understanding

To Load the dataset into a data analysis tool (e.g., Python with Pandas, Google Colab or visual studio code) and explore its structure and contents. We will use google colab and visual studio code based on the dependency of our project needs and preferences.

Google Colab is an excellent choice for data science and machine learning tasks that require GPU acceleration and collaborative work in Jupyter notebooks, especially when working on cloud-based usages.

Visual Studio Code, on the other hand, is a versatile code editor suitable for a wide range of programming tasks, offering extensive customization and offline development capabilities.

So, our choice varies based on our preferences.

Check for **missing values, outliers, and data types**.

Depending on the nature of the data and the analysis, we will perform data imputation to replace missing values with meaningful estimates, ensuring that valuable information is not lost. Some of them include:

- **Mean/Median Imputation**
- **Interpolation Methods (Time Series Data)**
- **K-Nearest Neighbours (KNN) Imputation**
- **Forward Fill and Backward Fill (Time Series Data)**

The choice of imputation method for handling missing values depends on several factors, like the nature of the dataset and the problem definition of our project.

There can be exceptionally high or low values that deviate from the general pattern or distribution of the data and hence identifying outliers is important. Outliers can distort data visualizations, making it challenging to interpret plots and graphs accurately, so we will use visualization techniques like box plots, scatter plots, and histograms to visually identify potential outliers and then Transform Data and then convert date columns to datetime objects for time series analysis.

Step 4: Splitting the Data

We will divide the dataset into training and testing sets. For time series data, we'll use a time-based split ensuring that the training set contains data from earlier years and the testing set contains data from later years within the dataset.

Step 5: Feature Engineering

To improve the performance of machine learning models (LSTM) by Normalizing or scaling numerical features. Using One-hot encoding converts categorical variables into binary columns to represent each category.

Step 6: Model Selection and Building

We will gradually explore complex models like time series models (e.g., ARIMA for the historical dataset) and machine learning models (LSTM) Long Short-Term Memory networks.

Step 7: Model Evaluation

To Evaluate model performance on the testing data using metrics like Mean Absolute Error (MAE), Mean Squared Error (MSE), Root Mean Squared Error (RMSE), and R-squared (R²) Score and create visualizations to compare predicted prices with actual prices to see accuracy.

Step 8: Continuous Learning and Monitoring

To implement mechanisms for continuous learning, updating models with new data, and monitoring model performance over time.

Step 8: Hyperparameter Tuning

As we are using machine learning models, performing hyperparameter tuning techniques like grid search will be used. To generate evaluation reports, visualizations, and confusion matrices is used to provide insights into the model's strengths and weaknesses.

This step-by-step detail provides a comprehensive approach to stock price prediction using the Kaggle dataset spanning from 1986 to 2020 based on our understanding to solving the problem statement.

Stock price prediction is a complex task that involves analyzing historical data and making forecasts about future price movements. Two common approaches for stock price prediction are Linear Regression and Long Short-Term Memory (LSTM) neural networks. A brief introduction to both methods and how they can be used for stock price prediction is described below:

Linear Regression:

Linear regression is a simple and interpretable machine learning technique used for predicting a continuous variable, such as stock prices. It assumes a linear relationship between the input features and the target variable (stock price in this case).

Data Preparation: Use the historical stock price data and relevant features provided (e.g., trading volume, technical indicators, economic factors).

Feature Engineering: Extract and preprocess features, including lagged values of stock prices, moving averages, and other relevant data.

Model Training: Fitting a linear regression model to the training data, where the input features are used to predict the future stock price.

Evaluation: Using metrics like Mean Squared Error (MSE) or Root Mean Squared Error (RMSE) to evaluate the model's performance on a validation dataset.

Prediction: To Make predictions on unseen data to forecast future stock prices.

Linear regression has limitations in capturing complex patterns in stock price data, which is where LSTM comes into play.

Long Short-Term Memory (LSTM):

LSTM is a type of recurrent neural network (RNN) designed to handle sequential data and capture long-term dependencies. It is particularly well-suited for time series forecasting tasks like stock price prediction.

Data Preparation: Similar to Linear Regression, we use the historical stock price data and relevant features.

Data Preprocessing: Normalize or scale the data to ensure that it falls within a specific range.

Sequence Creation: Converting the time series data into sequences of fixed length. Each sequence contains historical stock prices and corresponding features.

Model Architecture: Build an LSTM neural network architecture with input layers, LSTM layers, and output layers. The LSTM layers can capture temporal dependencies in the data.

Model Training: Training the LSTM model using the dataset, optimizing it to make accurate predictions.

Evaluation: Use appropriate evaluation metrics (e.g., Mean Absolute Error, Mean Absolute Percentage Error) to assess the model's performance on a validation dataset.

Use the trained LSTM model to make predictions about future stock prices based on input sequences.

LSTM models are generally better at capturing complex patterns and long-term dependencies in stock price data compared to linear regression. However, they are also more complex to train and require more data.

Both linear regression and LSTM models can serve as useful tools, but they may have certain limitations and risks associated with stock trading decisions.

BASIC PYTHON CODE FOR LINEAR REGRESSION

The first step in implementing linear regression with a dataset involves data preparation, which includes the following key steps:

Collecting the Data:

Obtain the dataset that contains the relevant information for the linear regression analysis. In the context of stock price prediction, this might include historical stock prices and relevant features.

Exploratory Data Analysis (EDA):

Performing an initial exploration of the dataset to understand its structure, features, and any missing values. We use tools like Python's Pandas library or Google colab to load and explore the data. EDA helps you gain insights into the dataset and decide which features to use for regression model.

Data Preprocessing:

Data preprocessing is crucial for preparing the dataset for linear regression. Common preprocessing steps include:

Handling missing data: Decide how to handle missing values (e.g., imputing with the mean, median, or remove rows with missing data).

Feature selection: Choosing relevant features (independent variables) for regression model.

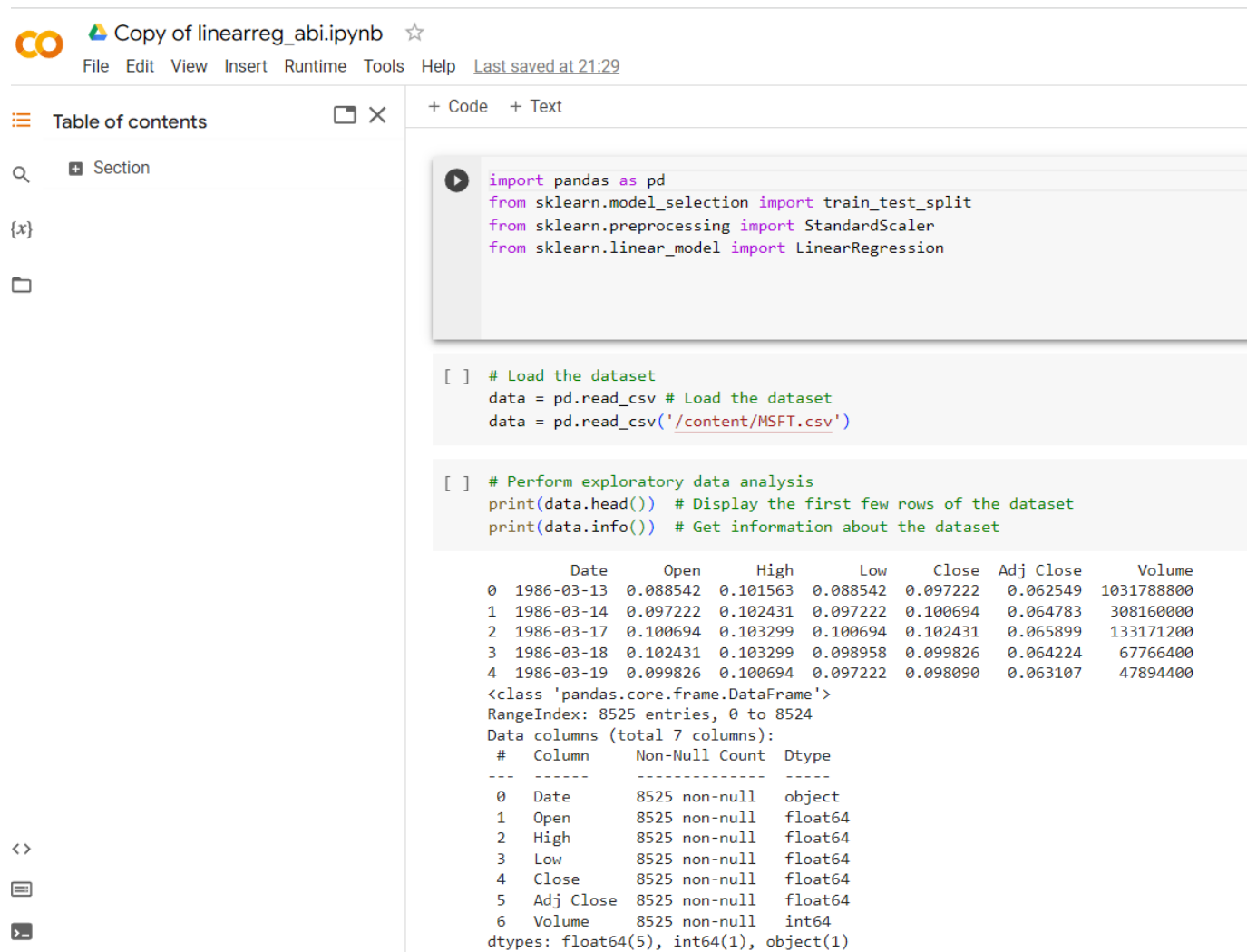
Data scaling: Normalize or standardize numerical features to ensure they have a similar scale (e.g., using Min-Max scaling).

Categorical variables: In case of categorical variables, we need to encode them (e.g., one-hot encoding) to make them suitable for regression.

Splitting the Data:

Split the dataset into training and testing sets. The training set is used to train the linear regression model, while the testing set is used to evaluate its performance. A common split ratio is 80% for training and 20% for testing.

Here is a Python code demonstrating the steps using the popular libraries Pandas and Scikit-Learn using our dataset:



The screenshot shows a Jupyter Notebook titled "Copy of linearreg_abi.ipynb". The interface includes a "Table of contents" sidebar on the left and a main code area on the right. The code area contains two code cells. The first cell imports pandas as pd and uses sklearn for train_test_split, StandardScaler, and LinearRegression. The second cell loads the dataset from "/content/MSFT.csv" and performs exploratory data analysis by printing the first few rows and the dataset's information. The output of the second cell shows a preview of the first five rows of the dataset and a summary of the DataFrame's structure.

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LinearRegression

[ ] # Load the dataset
data = pd.read_csv # Load the dataset
data = pd.read_csv('/content/MSFT.csv')

[ ] # Perform exploratory data analysis
print(data.head()) # Display the first few rows of the dataset
print(data.info()) # Get information about the dataset
```

	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

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 8525 entries, 0 to 8524
Data columns (total 7 columns):
#   Column      Non-Null Count  Dtype
---  -
0   Date        8525 non-null   object
1   Open        8525 non-null   float64
2   High        8525 non-null   float64
3   Low         8525 non-null   float64
4   Close       8525 non-null   float64
5   Adj Close   8525 non-null   float64
6   Volume      8525 non-null   int64
dtypes: float64(5), int64(1), object(1)
```

Step 1: Import Libraries and Load dataset

numpy: A library for numerical operations in Python.

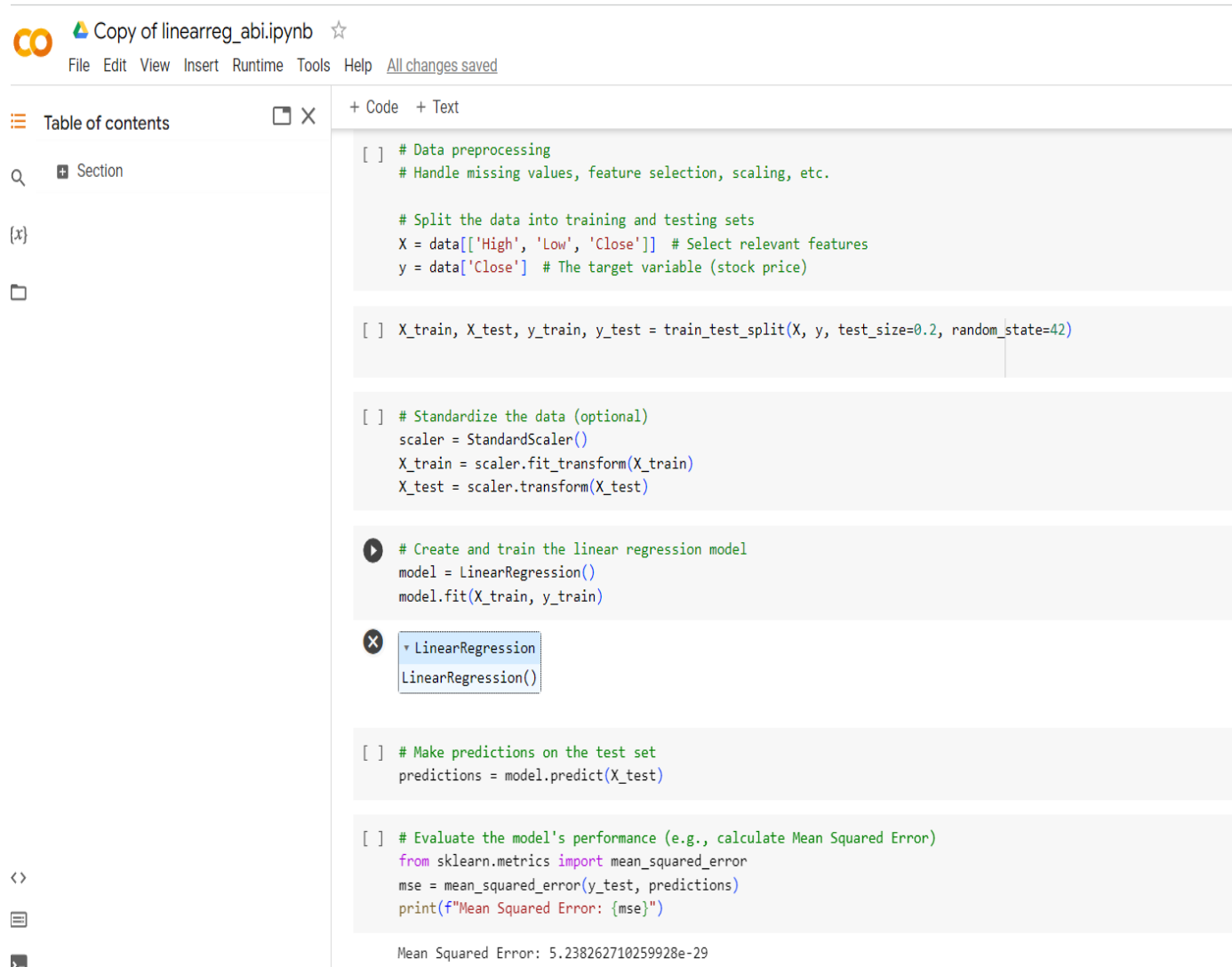
matplotlib, pyplot: Used for data visualization.

Linear Regression from sklearn.linear_model: This class provides the linear regression model that we'll use.

Import pandas and use one of its functions like **read_csv()** to load data from a CSV file

Step 2: Prepare Data

We have a dataset with two arrays: one representing the independent variable (usually denoted as 'X') and another representing the dependent variable (usually denoted as 'y').



The screenshot shows a Jupyter Notebook titled "Copy of linearreg_abi.ipynb". The interface includes a top menu bar with "File", "Edit", "View", "Insert", "Runtime", "Tools", and "Help". On the left, there is a "Table of contents" sidebar with a search bar and a list of sections. The main area displays a series of code cells. The first cell contains comments about data preprocessing and splitting the data into training and testing sets. The second cell shows the use of `train_test_split` to create `X_train`, `X_test`, `y_train`, and `y_test`. The third cell shows the standardization of the data using `StandardScaler`. The fourth cell, which is currently selected, shows the creation and training of a `LinearRegression` model. A tooltip for `LinearRegression()` is visible. The fifth cell shows the prediction of values on the test set using `model.predict(X_test)`. The sixth cell shows the evaluation of the model's performance using `mean_squared_error` from `sklearn.metrics`. The output of the final cell is "Mean Squared Error: 5.238262710259928e-29".

```
[ ] # Data preprocessing
# Handle missing values, feature selection, scaling, etc.

# Split the data into training and testing sets
X = data[['High', 'Low', 'Close']] # Select relevant features
y = data['Close'] # The target variable (stock price)

[ ] X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

[ ] # Standardize the data (optional)
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Create and train the linear regression model
model = LinearRegression()
model.fit(X_train, y_train)

# LinearRegression
LinearRegression()

[ ] # Make predictions on the test set
predictions = model.predict(X_test)

[ ] # Evaluate the model's performance (e.g., calculate Mean Squared Error)
from sklearn.metrics import mean_squared_error
mse = mean_squared_error(y_test, predictions)
print(f"Mean Squared Error: {mse}")

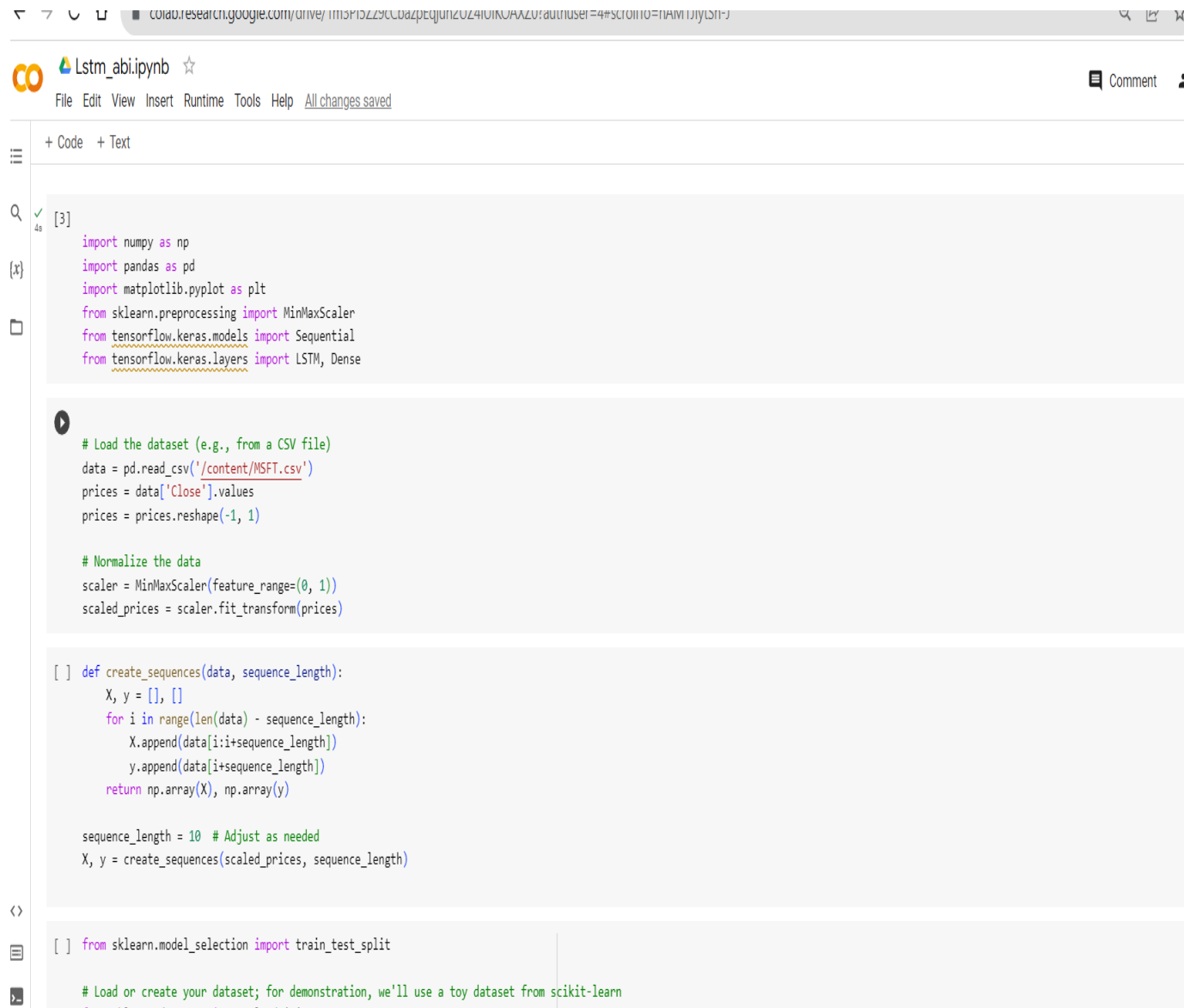
Mean Squared Error: 5.238262710259928e-29
```

Step 3: Create and Train the Linear Regression Model

Now, let's create an instance of the `LinearRegression` model and fit it to our data.

Step 4: Make Predictions

Now that the model is trained, you can use it to make predictions. This method predicts the values of the dependent variable ('y') based on the independent variable ('X').



```
[3]
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.preprocessing import MinMaxScaler
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense

# Load the dataset (e.g., from a CSV file)
data = pd.read_csv('/content/MSFT.csv')
prices = data['Close'].values
prices = prices.reshape(-1, 1)

# Normalize the data
scaler = MinMaxScaler(feature_range=(0, 1))
scaled_prices = scaler.fit_transform(prices)

[ ] def create_sequences(data, sequence_length):
    X, y = [], []
    for i in range(len(data) - sequence_length):
        X.append(data[i:i+sequence_length])
        y.append(data[i+sequence_length])
    return np.array(X), np.array(y)

sequence_length = 10 # Adjust as needed
X, y = create_sequences(scaled_prices, sequence_length)

[ ] from sklearn.model_selection import train_test_split

# Load or create your dataset; for demonstration, we'll use a toy dataset from scikit-learn
from sklearn.datasets import load_boston
boston = load_boston()
X, y = boston.data, boston.target
```

Step 1: Import Libraries

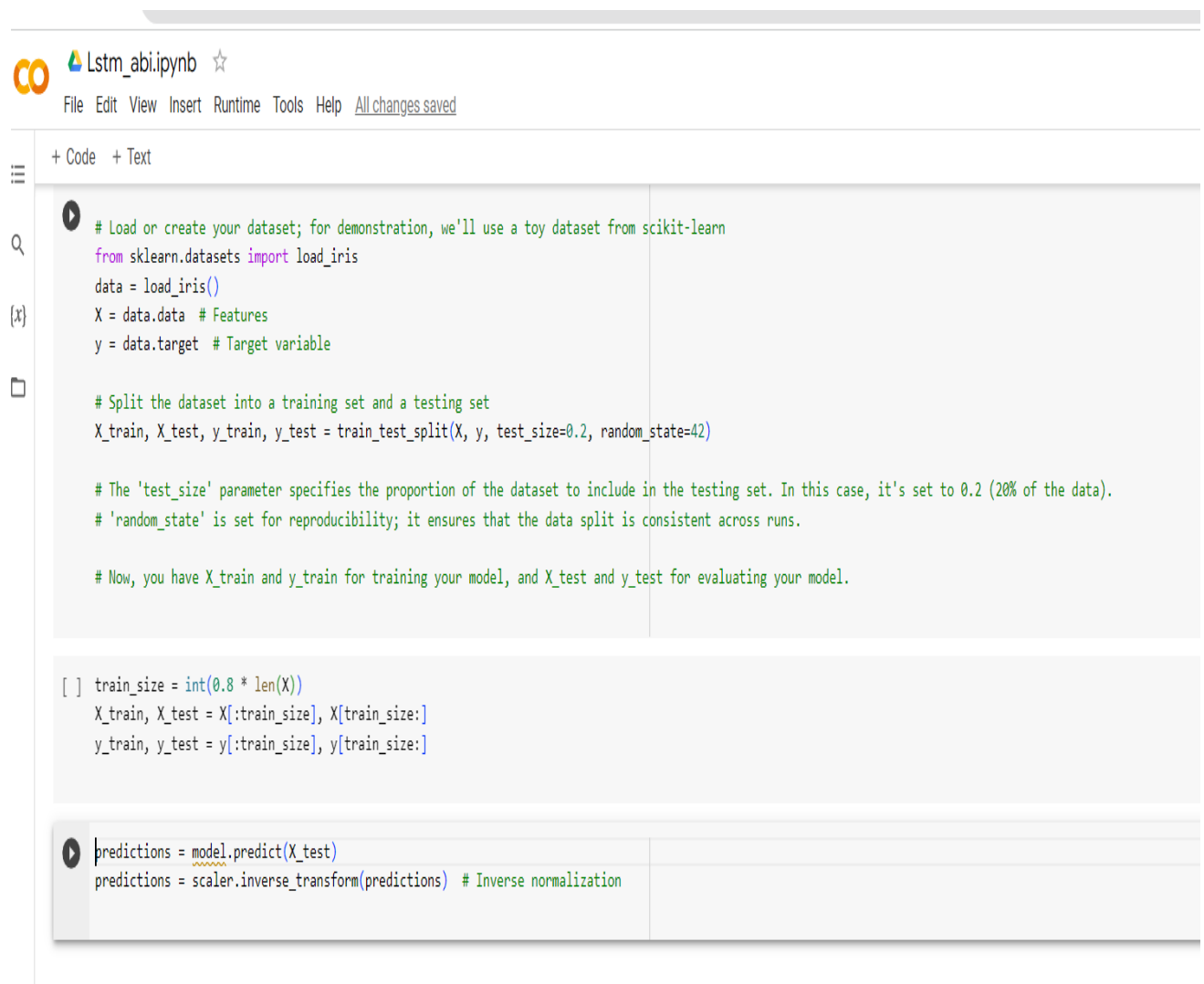
We'll need to import the necessary libraries, including TensorFlow/Keras and other data processing tools.

Step 2: Load and Preprocess Data

Load historical stock price data and preprocess it. We should have a dataset with at least two columns: one for the date and one for the stock prices.

Step 3: Prepare Sequences

To train the LSTM model, create sequences of dataset, where each input sequence contains a window of past stock prices and the corresponding output sequence contains the predicted stock price. This is referred to as sequence-to-sequence forecasting.



The screenshot shows a Jupyter Notebook titled 'Lstm_abi.ipynb'. The interface includes a menu bar with 'File', 'Edit', 'View', 'Insert', 'Runtime', 'Tools', and 'Help', along with a status bar indicating 'All changes saved'. On the left, there are icons for a table of contents, search, and a variable inspector. The main area contains two code cells. The first cell is a comment block explaining the data loading and splitting process. The second cell contains Python code for splitting the data into training and testing sets.

```
# Load or create your dataset; for demonstration, we'll use a toy dataset from scikit-learn
from sklearn.datasets import load_iris
data = load_iris()
X = data.data # Features
y = data.target # Target variable

# Split the dataset into a training set and a testing set
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# The 'test_size' parameter specifies the proportion of the dataset to include in the testing set. In this case, it's set to 0.2 (20% of the data).
# 'random_state' is set for reproducibility; it ensures that the data split is consistent across runs.

# Now, you have X_train and y_train for training your model, and X_test and y_test for evaluating your model.

[ ] train_size = int(0.8 * len(X))
X_train, X_test = X[:train_size], X[train_size:]
y_train, y_test = y[:train_size], y[train_size:]

predictions = model.predict(X_test)
predictions = scaler.inverse_transform(predictions) # Inverse normalization
```

Step 4: Split Data into Training and Testing Sets

Split the data into training and testing sets to evaluate the model's performance.

Step 5: Train the Model

Train the LSTM model using the training data.

Step 7: Make Predictions

Use the trained model to make predictions on the test data.

We should fine-tune hyperparameters, optimize the model architecture, and consider other factors like feature engineering and using more sophisticated techniques for better stock price predictions.

Also, be aware that predicting stock prices is challenging and uncertain due to various external factors affecting financial markets.

The predicted stock prices are typically obtained by feeding a sequence of historical stock prices as input to your trained Long Short-Term Memory (LSTM) model.

The sequence used as input, often referred to as a "window" or "lookback window," is used to make predictions about the next stock price.

We can use visualization libraries like **matplotlib** to plot the actual and predicted stock prices to assess the model's performance.

Loading and pre-processing the provided stock price dataset typically involves the following steps:

Data Collection: We need to obtain historical stock price data as provided from Kaggle website. The link for the dataset on which tasks are performed is given below:

<https://www.kaggle.com/datasets/prasoonkottarathil/microsoft-lifetime-stocks-dataset>

Data Format: First we should ensure that the data is in a format that you can work with. Typically, this data will be in CSV (Comma-Separated Values) format or some other structured format.

Import Libraries: We will need to import relevant libraries for data manipulation and analysis. Common libraries for this task include *pandas*, *numpy*, and *matplotlib* for visualization. We may also need libraries to fetch data from the web.



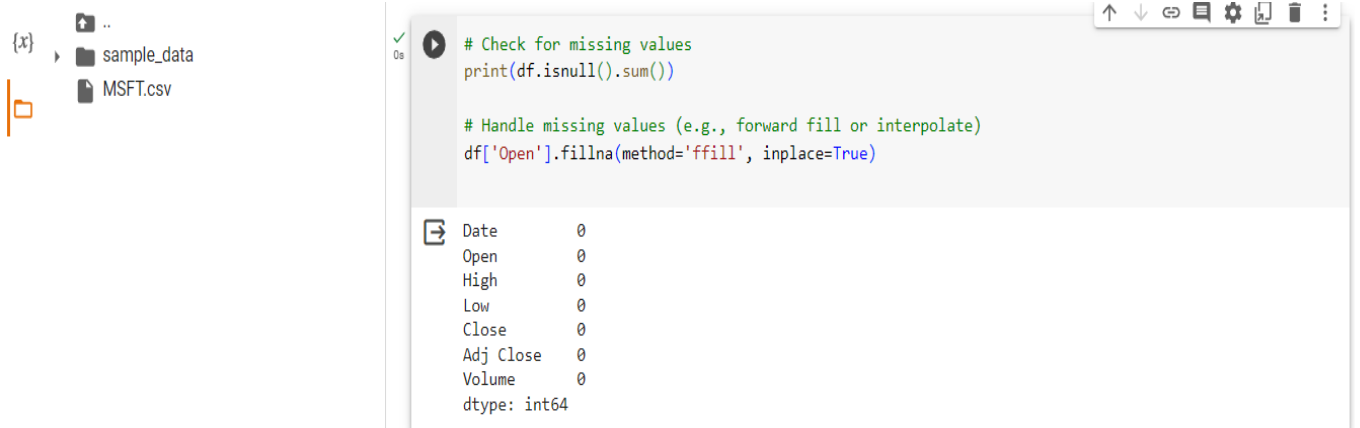
```
[1] import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

Load the Data: Use pandas to read the dataset into a DataFrame. Here we load a CSV file:



```
[2] # dataset is saved as CSV in the working directory
df = pd.read_csv('/content/MSFT.csv')
```

Data Cleaning: Check for missing values, duplicate rows, and outliers in the dataset. We need to handle missing data through imputation, drop duplicates, and handle outliers based on our analysis requirements.



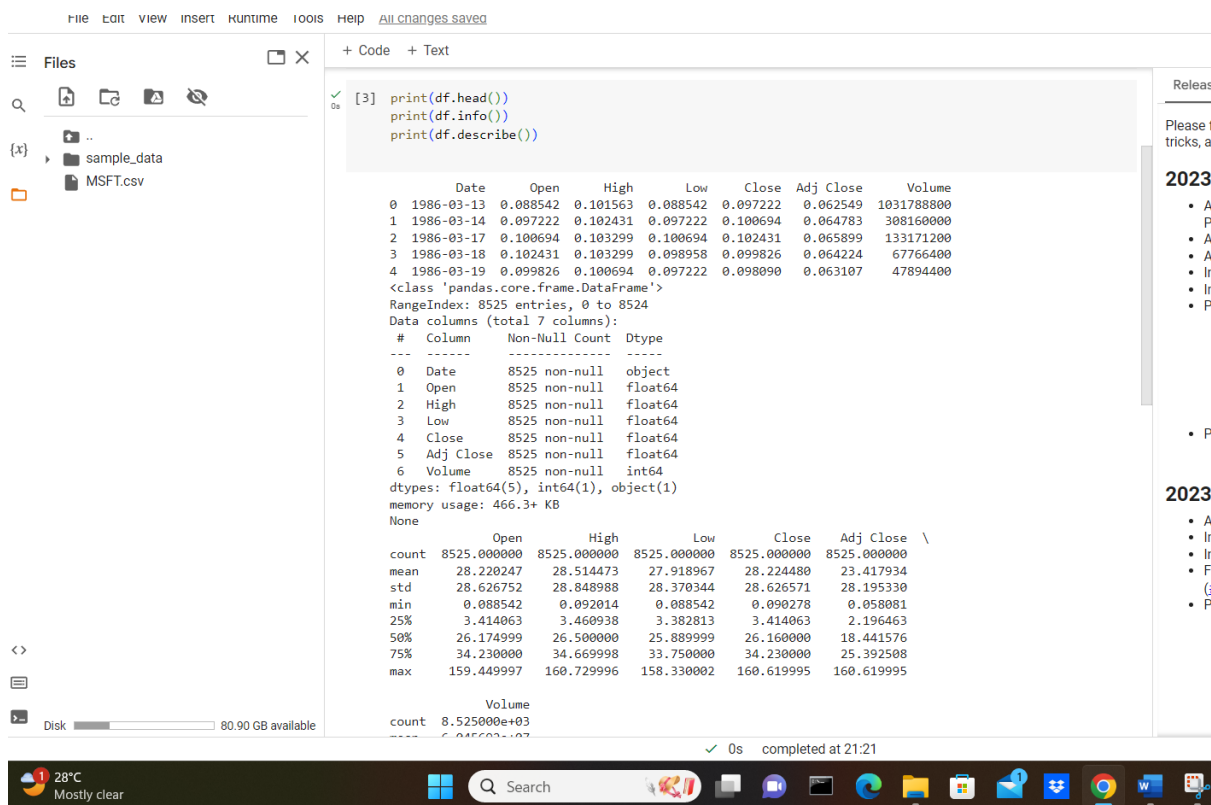
```
{x} ..
  sample_data
  MSFT.csv

# Check for missing values
print(df.isnull().sum())

# Handle missing values (e.g., forward fill or interpolate)
df['Open'].fillna(method='ffill', inplace=True)

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

Data Exploration: Explore the dataset to gain insights. We use methods like `describe()`, `info()`, and visualizations to understand the data better.



```
File Edit View Insert Runtime Tools Help All changes saved

Files
{x} ..
  sample_data
  MSFT.csv

[3] print(df.head())
    print(df.info())
    print(df.describe())

      Date      Open      High      Low      Close  Adj Close  Volume
0  1986-03-13  0.088542  0.101563  0.088542  0.097222  0.062549  1031788000
1  1986-03-14  0.097222  0.102431  0.097222  0.100694  0.064783  3081600000
2  1986-03-17  0.100694  0.103299  0.100694  0.102431  0.065899  1331712000
3  1986-03-18  0.102431  0.103299  0.098958  0.099826  0.064224  677664000
4  1986-03-19  0.099826  0.100694  0.097222  0.098090  0.063107  478944000
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 8525 entries, 0 to 8524
Data columns (total 7 columns):
#   Column      Non-Null Count  Dtype
---  ---
0   Date        8525 non-null   object
1   Open        8525 non-null   float64
2   High        8525 non-null   float64
3   Low         8525 non-null   float64
4   Close       8525 non-null   float64
5   Adj Close   8525 non-null   float64
6   Volume      8525 non-null   int64
dtypes: float64(5), int64(1), object(1)
memory usage: 466.3+ KB
None
      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    6.045000e+02
std     1.045000e+02
min      0.000000e+00
25%     0.000000e+00
50%     0.000000e+00
75%     0.000000e+00
max     0.000000e+00

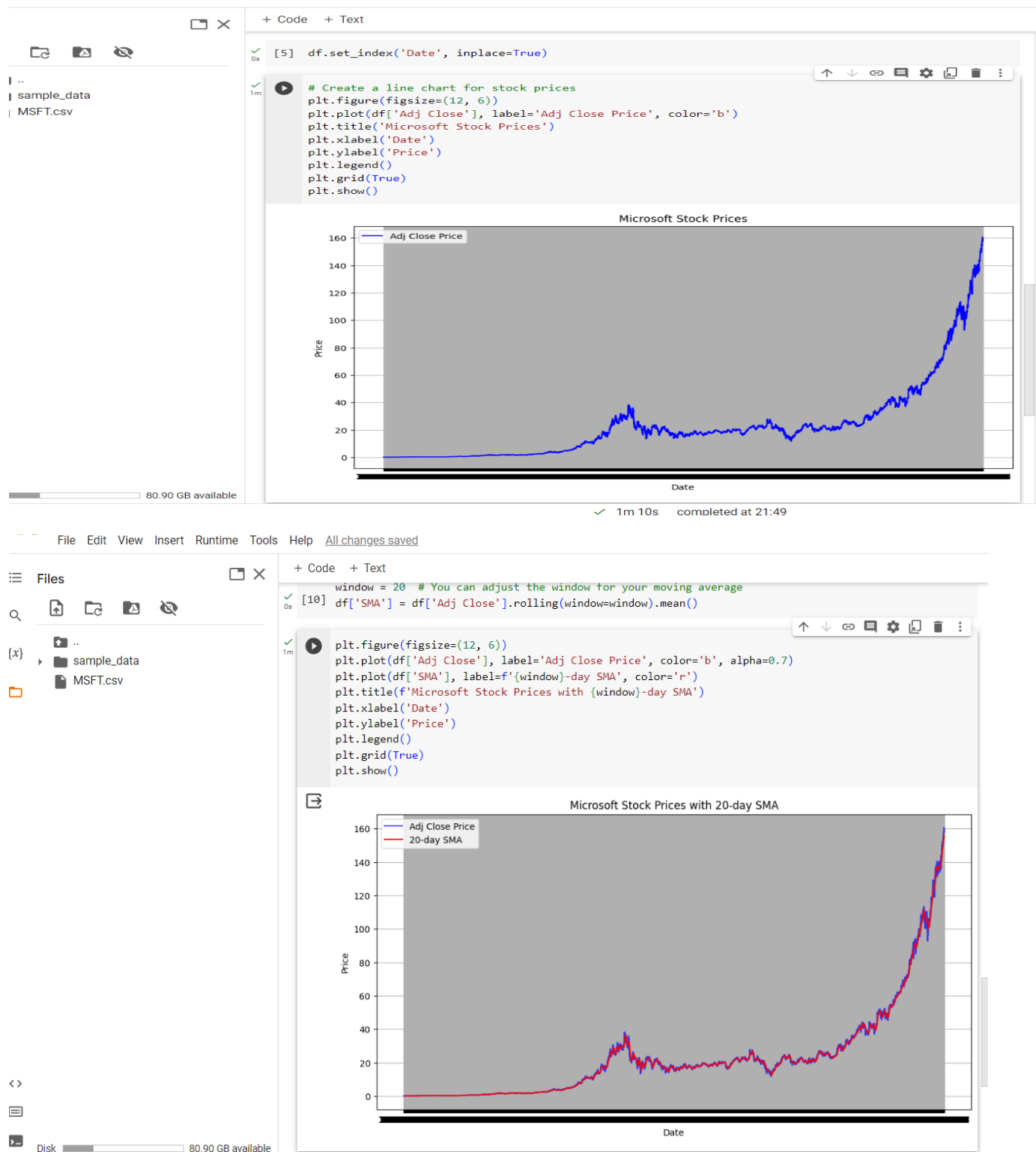
2023
• A
• P
• A
• A
• Ir
• Ir
• P

2023
• A
• Ir
• Ir
• F
• F
• P

Disk 80.90 GB available
completed at 21:21
```

After loading and pre-processing stock price dataset, it's a good practice to create visualizations to gain insights and better understand the data. Common visualizations for financial time series data like stock prices include line charts, candlestick charts, volume plots, and

moving averages. Here we can create these visualizations using Python.



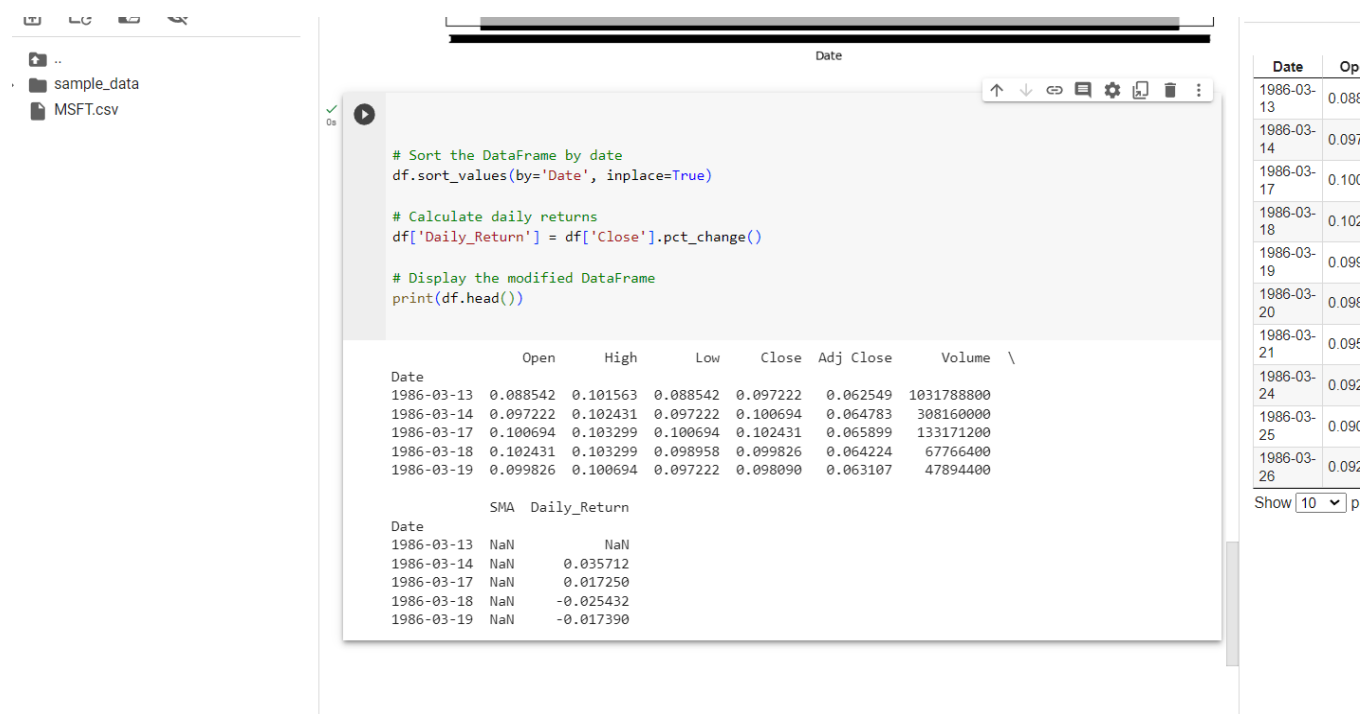
Data Pre-processing:

Date Conversion: Convert the date column to a datetime object if it's not already in that format.

Sorting: Sort the Data Frame by date if it's not already in chronological order.

Feature Engineering: Create additional features if needed, such as moving averages, daily returns, or technical indicators.

Normalization/Scaling: If you plan to use machine learning models, you might want to normalize or scale the data. Common techniques include Min-Max scaling or standardization.



The screenshot shows a Jupyter Notebook interface. On the left, a file explorer displays a folder named 'sample_data' containing a file 'MSFT.csv'. The main notebook area contains the following Python code:

```
# Sort the DataFrame by date
df.sort_values(by='Date', inplace=True)

# Calculate daily returns
df['Daily_Return'] = df['Close'].pct_change()

# Display the modified DataFrame
print(df.head())
```

The output of the code is displayed below the code cell. It consists of two tables. The first table shows the first five rows of the sorted DataFrame with columns: Date, Open, High, Low, Close, Adj Close, and Volume. The second table shows the first five rows of the DataFrame with columns: Date, SMA, and Daily_Return.

Date	Open	High	Low	Close	Adj Close	Volume
1986-03-13	0.088542	0.101563	0.088542	0.097222	0.062549	1031788800
1986-03-14	0.097222	0.102431	0.097222	0.100694	0.064783	308160000
1986-03-17	0.100694	0.103299	0.100694	0.102431	0.065899	133171200
1986-03-18	0.102431	0.103299	0.098958	0.099826	0.064224	67766400
1986-03-19	0.099826	0.100694	0.097222	0.098090	0.063107	47894400

Date	SMA	Daily_Return
1986-03-13	NaN	NaN
1986-03-14	NaN	0.035712
1986-03-17	NaN	0.017250
1986-03-18	NaN	-0.025432
1986-03-19	NaN	-0.017390

On the right side of the notebook, a preview of the DataFrame is shown with columns 'Date' and 'Op' (likely representing 'Open'). It displays the first 10 rows of the data.

Data Splitting: If you plan to build a predictive model, split the dataset into training and testing sets. Make sure to maintain the chronological order.

```
[10] from sklearn.model_selection import train_test_split

# Split the data into training and testing sets
train_size = 0.8 # 80% for training, 20% for testing
train_data, test_data = train_test_split(df, train_size=train_size, shuffle=False)
```

To visualize the training and testing datasets for stock price dataset, we create separate line charts for each dataset to compare them. This helps in understanding how our model's predictions (on the test dataset) compare to the actual stock price movements (from the training dataset).



We load the pre-processed data, containing the 'Adj Close' prices.

We have separate Data Frames for training and testing data, 'train_data' and 'test_data'.

We create line charts for both datasets, with different colours to distinguish between training and testing data.

The x-axis represents the date, and the y-axis represents the adjusted closing price.

This visualization helps compare the stock price movements in the training and testing datasets. It's useful for assessing how well our model generalizes to unseen data. In an ideal scenario, the testing data

would closely follow the trends seen in the training data, indicating that our model has learned and generalizes well.

In addition to the basic preprocessing steps mentioned earlier, there are several other advanced preprocessing steps that can be performed on a stock price dataset. These steps are often used to enhance the quality and relevance of the data for analysis and modelling.

Feature engineering is a critical step in building a stock price prediction model. The goal is to create informative input features that help our model capture meaningful patterns in the data.

Lagged Prices and Returns:

Create lag features by shifting historical prices and returns. For example, we can include features like the closing price of the previous day, week, or month.

Calculate daily or intraday returns, which can provide information about short-term price movements.

Moving Averages:

Moving averages, such as the simple moving average (SMA) and exponential moving average (EMA), can help smooth out noise and identify trends.

Volatility Indicators:

Features like historical price volatility, Bollinger Bands, or Average True Range (ATR) can capture the stock's price volatility.

Volume-Related Features:

Incorporate trading volume data, which can provide insights into market interest and liquidity.

Create moving averages or other statistics related to trading volume.

Feature Scaling and Transformation:

Standardize or normalize features to ensure they are on a similar scale, which can improve the performance of some machine learning algorithms.

```
+ Code + Text

[2] import pandas as pd
import numpy as np

!curl -L http://prdownloads.sourceforge.net/ta-lib/ta-lib-0.4.0-src.tar.gz -O && tar xzvf ta-lib-0.4.0-src.tar.gz
!cd ta-lib && ./configure --prefix=/usr && make && make install && cd - && pip install ta-lib

+ Code + Text

[5] import talib # Technical Analysis Library for technical indicators

[7] # Load your stock price data into a DataFrame
# Replace 'your_data.csv' with the actual file or data source
df = pd.read_csv('/content/MSFT.csv')
df['Date'] = pd.to_datetime(df['Date'])
df.set_index('Date', inplace=True)

[8] # Create lagged returns
df['Lagged_Return'] = df['Close'].pct_change()

[9] # Create moving averages
df['SMA_50'] = df['Close'].rolling(window=50).mean()
df['SMA_200'] = df['Close'].rolling(window=200).mean()

[10] # Calculate RSI (Relative Strength Index)
df['RSI'] = talib.RSI(df['Close'])

[11] # Calculate MACD (Moving Average Convergence Divergence)
macd, signal, _ = talib.MACD(df['Close'])
df['MACD'] = macd
df['MACD_Signal'] = signal

[12] # Calculate Bollinger Bands
upper, middle, lower = talib.BBANDS(df['Close'])
df['Bollinger_Upper'] = upper
df['Bollinger_Middle'] = middle
df['Bollinger_Lower'] = lower
```

```
[13] # Display the first few rows of the DataFrame
print(df.head())
```

Date	Open	High	Low	Close	Adj Close	Volume
1986-03-13	0.088542	0.101563	0.088542	0.097222	0.062549	1031788800
1986-03-14	0.097222	0.102431	0.097222	0.100694	0.064783	308160000
1986-03-17	0.100694	0.103299	0.100694	0.102431	0.065899	133171200
1986-03-18	0.102431	0.103299	0.098958	0.099826	0.064224	67766400
1986-03-19	0.099826	0.100694	0.097222	0.098090	0.063107	47894400

Date	Lagged_Return	SMA_50	SMA_200	RSI	MACD	MACD_Signal
1986-03-13	NaN	NaN	NaN	NaN	NaN	NaN
1986-03-14	0.035712	NaN	NaN	NaN	NaN	NaN
1986-03-17	0.017250	NaN	NaN	NaN	NaN	NaN
1986-03-18	-0.025432	NaN	NaN	NaN	NaN	NaN
1986-03-19	-0.017390	NaN	NaN	NaN	NaN	NaN

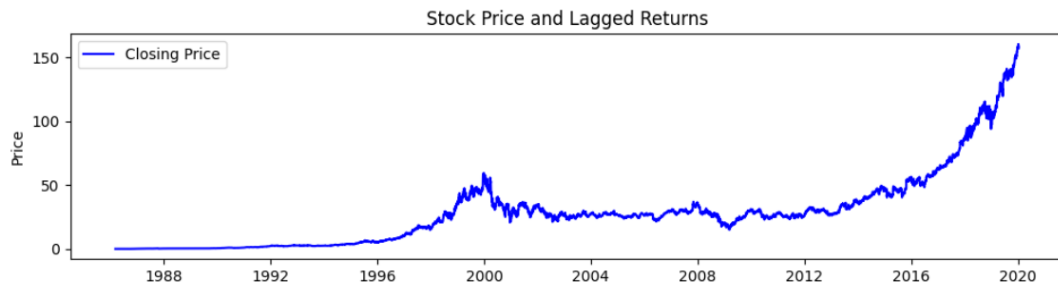
Date	Bollinger_Upper	Bollinger_Middle	Bollinger_Lower
1986-03-13	NaN	NaN	NaN
1986-03-14	NaN	NaN	NaN
1986-03-17	NaN	NaN	NaN
1986-03-18	NaN	NaN	NaN
1986-03-19	0.10336	0.099653	0.095945

```
[14] import matplotlib.pyplot as plt

# Plot the stock's closing price and lagged returns
plt.figure(figsize=(12, 6))
plt.subplot(2, 1, 1)
plt.plot(df.index, df['Close'], label='Closing Price', color='blue')
plt.title('Stock Price and Lagged Returns')
plt.ylabel('Price')
plt.legend(loc='best')
```

```
[14] # Plot the stock's closing price and lagged returns
plt.figure(figsize=(12, 6))
plt.subplot(2, 1, 1)
plt.plot(df.index, df['Close'], label='Closing Price', color='blue')
plt.title('Stock Price and Lagged Returns')
plt.ylabel('Price')
plt.legend(loc='best')
```

<matplotlib.legend.Legend at 0x7b0d58182e30>



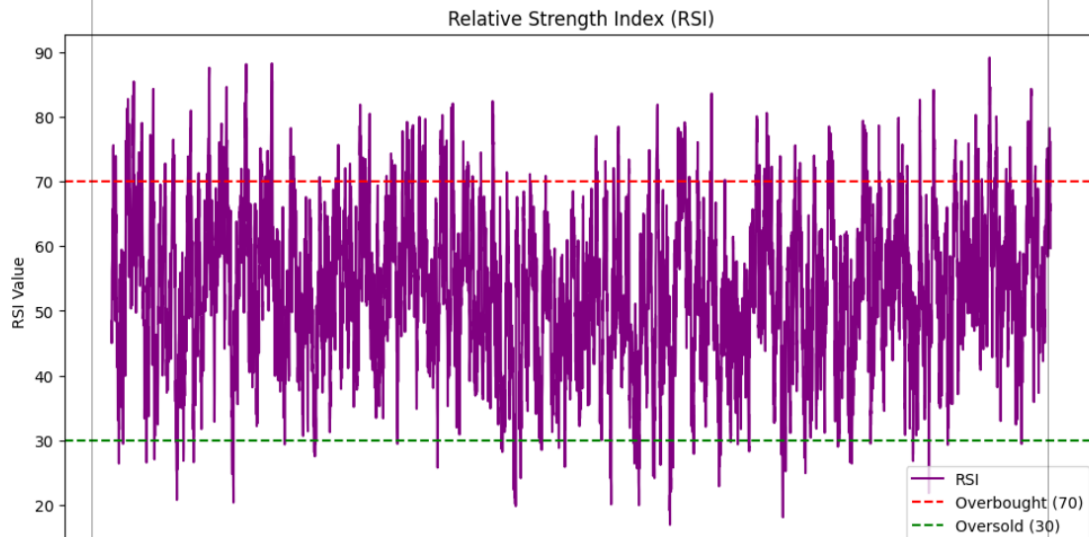
```
# Plot RSI
plt.figure(figsize=(12, 6))
plt.plot(df.index, df['RSI'], label='RSI', color='purple')
plt.axhline(y=70, color='r', linestyle='--', label='Overbought (70)')
plt.axhline(y=30, color='g', linestyle='--', label='Oversold (30)')
plt.title('Relative Strength Index (RSI)')
plt.xlabel('Date')
plt.ylabel('RSI Value')
plt.legend(loc='best')
```

```

# Plot RSI
plt.figure(figsize=(12, 6))
plt.plot(df.index, df['RSI'], label='RSI', color='purple')
plt.axhline(y=70, color='r', linestyle='--', label='Overbought (70)')
plt.axhline(y=30, color='g', linestyle='--', label='Oversold (30)')
plt.title('Relative Strength Index (RSI)')
plt.xlabel('Date')
plt.ylabel('RSI Value')
plt.legend(loc='best')

```

<matplotlib.legend.Legend at 0x7b0d4e5e3d30>



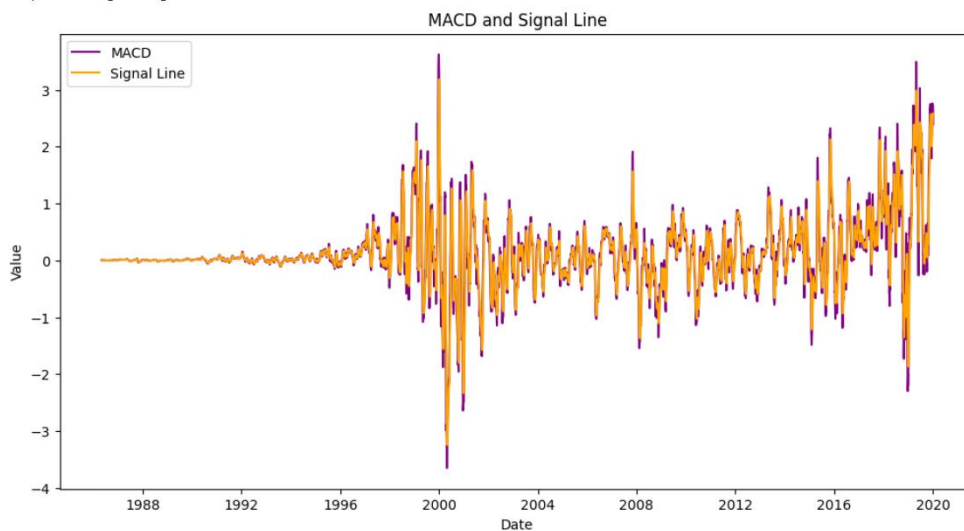
✓ 1s completed at 21:40

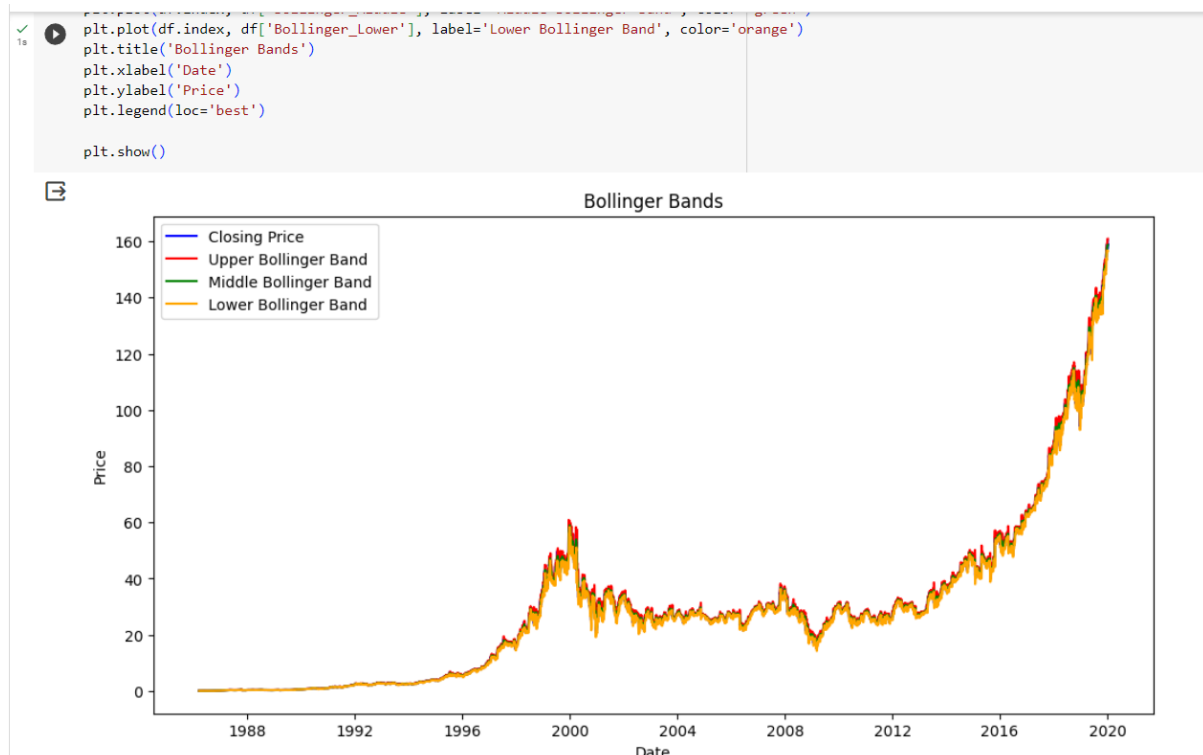
```

plt.plot(df.index, df['MACD'], label='MACD', color='purple')
plt.plot(df.index, df['MACD_Signal'], label='Signal Line', color='orange')
plt.title('MACD and Signal Line')
plt.xlabel('Date')
plt.ylabel('Value')
plt.legend(loc='best')

```

<matplotlib.legend.Legend at 0x7b0d50845ba0>





Stock Price and Lagged Returns:

The first subplot shows the stock's closing price (in blue).

The second subplot displays the lagged returns (in red), indicating the daily percentage change in the closing price.

Relative Strength Index (RSI):

The RSI plot (in purple) is a momentum oscillator that ranges from 0 to 100.

Red dashed line indicates overbought conditions ($RSI > 70$), and green dashed line indicates oversold conditions ($RSI < 30$).

MACD and Signal Line:

The plot shows the MACD (in purple) and the MACD signal line (in orange).

MACD can help identify trend changes and momentum.

Bollinger Bands:

This plot displays the closing price (in blue) along with the upper Bollinger Band (in red), middle Bollinger Band (in green), and lower Bollinger Band (in orange).

Bollinger Bands are used to measure volatility and potential price reversals.

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LinearRegression
```

```
[ ] # Load the dataset
data = pd.read_csv # Load the dataset
data = pd.read_csv('/content/MSFT.csv')
```

```
[ ] # Perform exploratory data analysis
print(data.head()) # Display the first few rows of the dataset
print(data.info()) # Get information about the dataset
```

```
   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

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 8525 entries, 0 to 8524
Data columns (total 7 columns):
#   Column      Non-Null Count  Dtype
---  ---
0   Date        8525 non-null    object
1   Open        8525 non-null    float64
2   High        8525 non-null    float64
3   Low         8525 non-null    float64
4   Close       8525 non-null    float64
5   Adj Close   8525 non-null    float64
6   Volume      8525 non-null    int64
dtypes: float64(5), int64(1), object(1)
```

Tools Help All changes saved

×

+ Code + Text

```
[ ] # Data preprocessing
# Handle missing values, feature selection, scaling, etc.

# Split the data into training and testing sets
X = data[['High', 'Low', 'Close']] # Select relevant features
y = data['Close'] # The target variable (stock price)

[ ] X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Standardize the data (optional)
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

[ ] # Create and train the linear regression model
model = LinearRegression()
model.fit(X_train, y_train)

LinearRegression
LinearRegression()

[ ] # Make predictions on the test set
predictions = model.predict(X_test)

[ ] # Evaluate the model's performance (e.g., calculate Mean Squared Error)
from sklearn.metrics import mean_squared_error
mse = mean_squared_error(y_test, predictions)
print(f"Mean Squared Error: {mse}")

Mean Squared Error: 5.238262710259928e-29
```

- Load your stock price data, rename the columns and split the data into training and test sets.
- Initialize the model and train it using the training data.
- Create a dataframe for future dates using `model.make_future_dataframe()`, where the number of future periods is set to the length of the test data.
- Generate forecasts with `model.predict(future)`.
- Extract the predicted values for the test set.
- Evaluate the model using Mean Squared Error (MSE), Root Mean Squared Error (RMSE), Mean Absolute Error (MAE), and R-squared (R2) score.

```
[ ] # Split the dataset into training (80%) and testing (20%) sets
X_train, X_test, y_train, y_test = train_test_split(dates, prices, test_size=0.2, random_state=42)
```

```
# Create a Linear Regression model
model = LinearRegression()

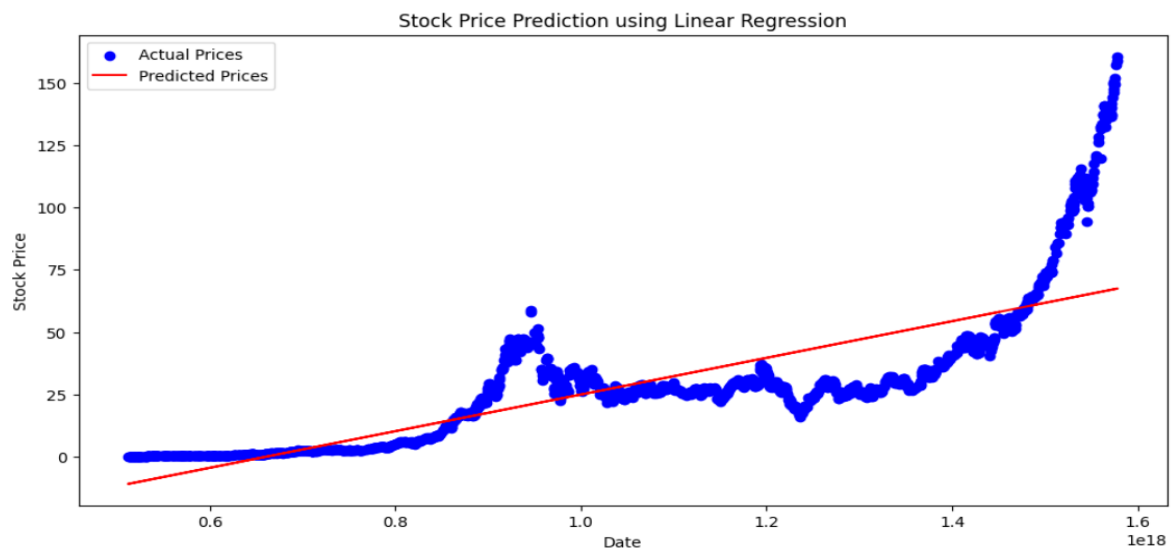
# Train the model on the training data
model.fit(X_train, y_train)
```

```
LinearRegression
LinearRegression()
```

```
[ ] # Use the model to make predictions on the test data
y_pred = model.predict(X_test)
```

Split the dataset into training and testing sets, create and train the Linear Regression model and Make predictions.

```
plt.plot(X_test, y_pred, color='r', label='Predicted Prices')
plt.xlabel('Date')
plt.ylabel('Stock Price')
plt.title('Stock Price Prediction using Linear Regression')
plt.legend()
plt.show()
```



CONCLUSION:

This code simplifies the linear regression model for stock price prediction. It downloads historical stock data, preprocesses it, splits it into training and testing sets, creates a Linear Regression model, makes predictions, and visualizes the results.

