

STOCK PRICE PREDICTION

DATA LOADING AND PREPROCESSING

INTRODUCTION:

-In this section, we will embark on the initial steps of our stock price prediction project by collecting and preprocessing historical stock market data.

-Accurate and well-organized data is the foundation of any successful predictive modeling project. We will utilize Python and relevant libraries to accomplish this task.

DATA COLLECTION:

DATA SOURCE:

-To begin our project, we need historical stock market data. You can obtain such data from various sources, including financial data providers, APIs, or public datasets.

-For the purpose of this project, we assume that we have already obtained a dataset in a suitable format.

- This dataset typically contains information about a stock's daily performance, including open price, close price, high price, low price, and trading volume.

LOADING THE DATA:

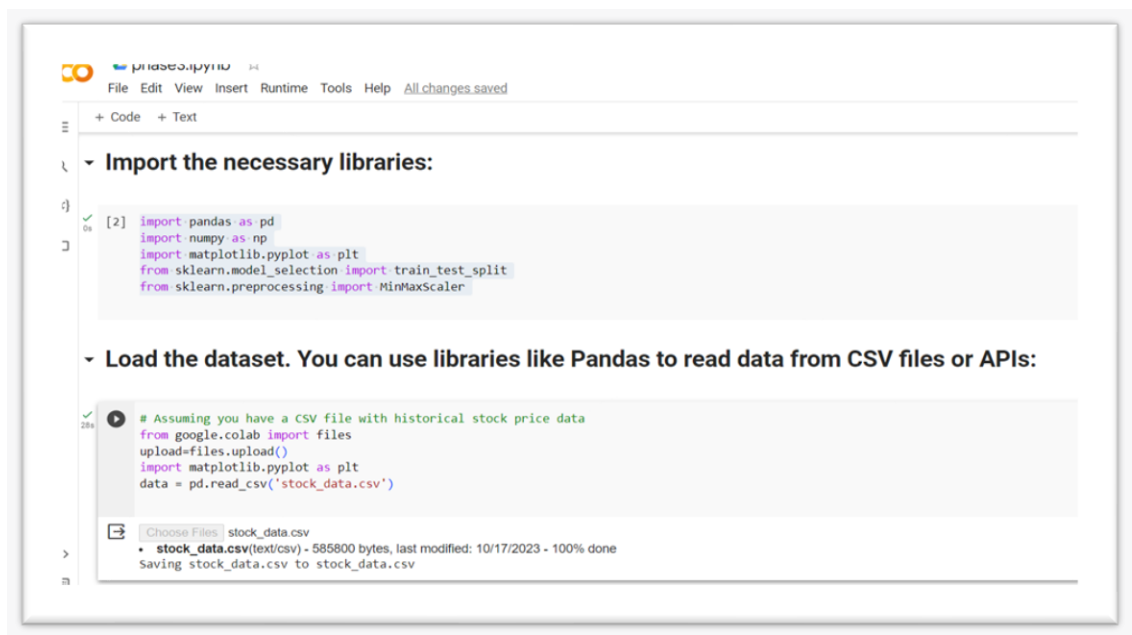
We will employ the Pandas library to load our dataset. Pandas is a powerful data manipulation library in Python that provides an efficient and flexible way to work with structured data.

INPUT:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import
train_test_split
from sklearn.preprocessing import
MinMaxScaler

# Assuming you have a CSV file with
historical stock price data
from google.colab import files
upload=files.upload()
import matplotlib.pyplot as plt
data = pd.read_csv('stock_data.csv')
```

OUTPUT:



The screenshot shows a Jupyter Notebook with two code cells. The first cell, titled 'Import the necessary libraries:', contains the following code:

```
[2] import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler
```

The second cell, titled 'Load the dataset. You can use libraries like Pandas to read data from CSV files or APIs:', contains the following code:

```
# Assuming you have a CSV file with historical stock price data
from google.colab import files
upload=files.upload()
import matplotlib.pyplot as plt
data = pd.read_csv('stock_data.csv')
```

Below the code, a file upload interface is visible, showing a file named 'stock_data.csv' (585800 bytes, last modified: 10/17/2023) being saved to the notebook.

Once we've loaded our data, we can inspect its structure and quality.

DATA PROCESSING:

DATA EXPLORATION:

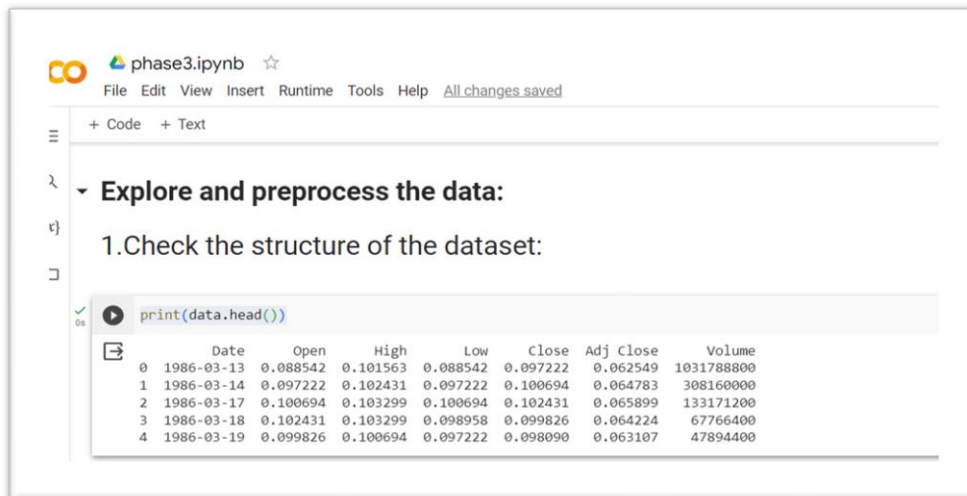
Before we dive into preprocessing, it's crucial to explore the dataset. Some key steps in this phase include:

1. Check the structure of the dataset:

INPUT:

```
print(data.head())
```

OUTPUT:



The screenshot shows a Jupyter Notebook window titled "phase3.ipynb". The menu bar includes File, Edit, View, Insert, Runtime, Tools, Help, and a link for "All changes saved". The left sidebar shows a file explorer with a folder icon and a list of files. The main area displays a code cell with the following content:

```
print(data.head())
```

The output of the code cell is a table with 8 columns: Date, Open, High, Low, Close, Adj Close, and Volume. The table contains 5 rows of data, indexed 0 to 4.

	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

2.Handle missing data, if any:

INPUT:

```
a=data.dropna(inplace=True) # Remove rows with  
missing values  
print(a)
```

OUTPUT:



The screenshot shows a Jupyter Notebook window titled "phase3.ipynb". The menu bar includes File, Edit, View, Insert, Runtime, Tools, Help, and a link for "All changes saved". The left sidebar shows a file explorer with a folder icon and a list of files. The main area displays a code cell with the following content:

```
a=data.dropna(inplace=True) # Remove rows with missing values  
print(a)
```

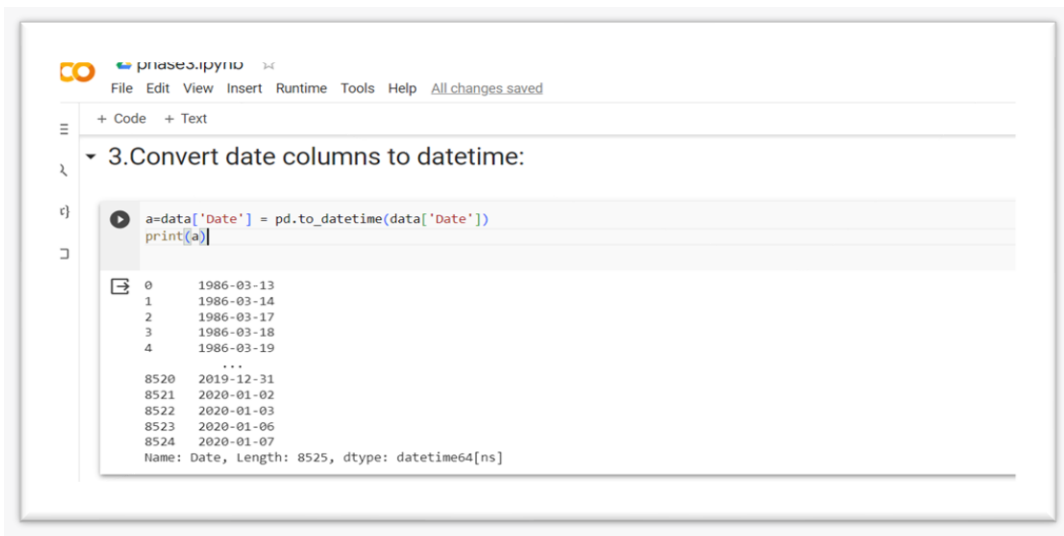
The output of the code cell is a single line: "None".

3.Convert date columns to datetime:

INPUT:

```
a=data['Date'] = pd.to_datetime(data['Date'])  
print(a)
```

OUTPUT:



The screenshot shows a Jupyter Notebook interface with a menu bar (File, Edit, View, Insert, Runtime, Tools, Help) and a status bar (All changes saved). The notebook has two tabs: '+ Code' and '+ Text'. The active tab is '+ Code', which contains the following code:

```
3. Convert date columns to datetime:  
  
a=data['Date'] = pd.to_datetime(data['Date'])  
print(a)
```

Below the code, the output is displayed as a table with two columns: an index and a date. The index ranges from 0 to 8524, with some rows omitted (indicated by dots). The dates are in YYYY-MM-DD format. At the bottom, the output is summarized as: Name: Date, Length: 8525, dtype: datetime64[ns].

0	1986-03-13
1	1986-03-14
2	1986-03-17
3	1986-03-18
4	1986-03-19
...	...
8520	2019-12-31
8521	2020-01-02
8522	2020-01-03
8523	2020-01-06
8524	2020-01-07

Name: Date, Length: 8525, dtype: datetime64[ns]

4.Sort the data by date:

INPUT:

```
a=data.sort_values(by='Date',  
inplace=True)  
print(a)
```

OUTPUT:

▼ 4.Sort the data by date:

```
[22] a=data.sort_values(by='Date', inplace=True)
      print(a)
None
```

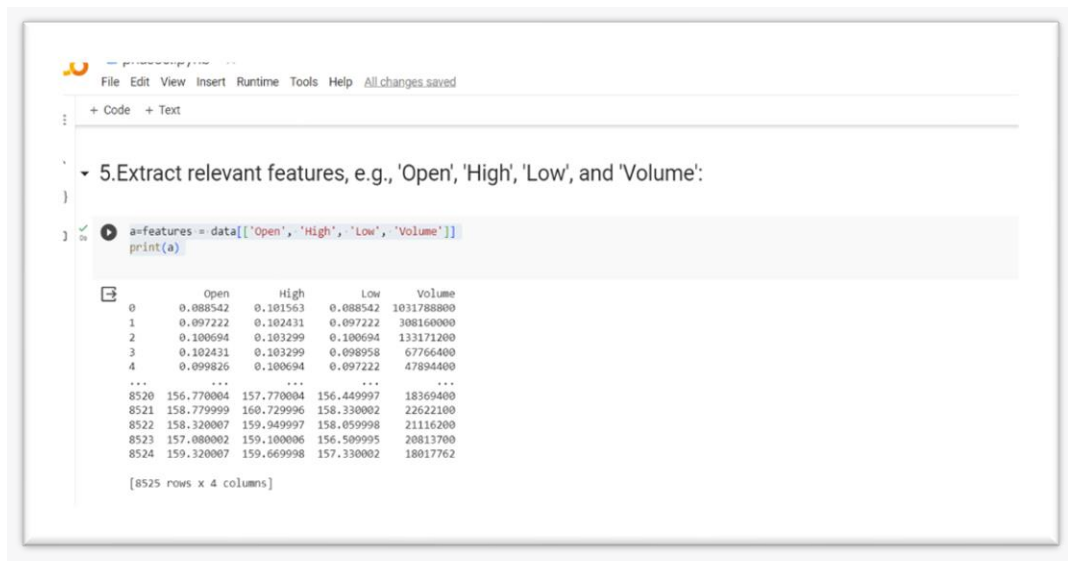
5.Extract relevant **features**, e.g., 'Open', 'High', 'Low', and 'Volume':

-In our stock price prediction model, we need to decide which features to include. Common features for stock prediction models include open price, high price, low price, and trading volume. We extract these features from the dataset:

INPUT:

```
a=features = data[['Open', 'High', 'Low',
'Volume']]
print(a)
```

OUTPUT:



The screenshot shows a Jupyter Notebook interface. The code cell contains the following Python code:

```
5.Extract relevant features, e.g., 'Open', 'High', 'Low', and 'Volume':  
  
a=features = data[['Open', 'High', 'Low', 'Volume']]  
print(a)
```

The output of the code is a DataFrame with 8525 rows and 4 columns: Open, High, Low, and Volume. The first few rows are displayed, followed by an ellipsis, and then the last few rows.

	Open	High	Low	Volume
0	0.088542	0.101563	0.088542	1031788800
1	0.097222	0.102431	0.097222	308160000
2	0.100694	0.103299	0.100694	133171200
3	0.102431	0.103299	0.098958	67766400
4	0.099826	0.100694	0.097222	47894400
...
8520	156.770004	157.770004	156.449997	18369400
8521	158.779999	160.729996	158.330002	22622100
8522	158.320007	159.949997	158.059998	21116200
8523	157.080002	159.100006	156.509995	20813700
8524	159.320007	159.669998	157.330002	18017762

[8525 rows x 4 columns]

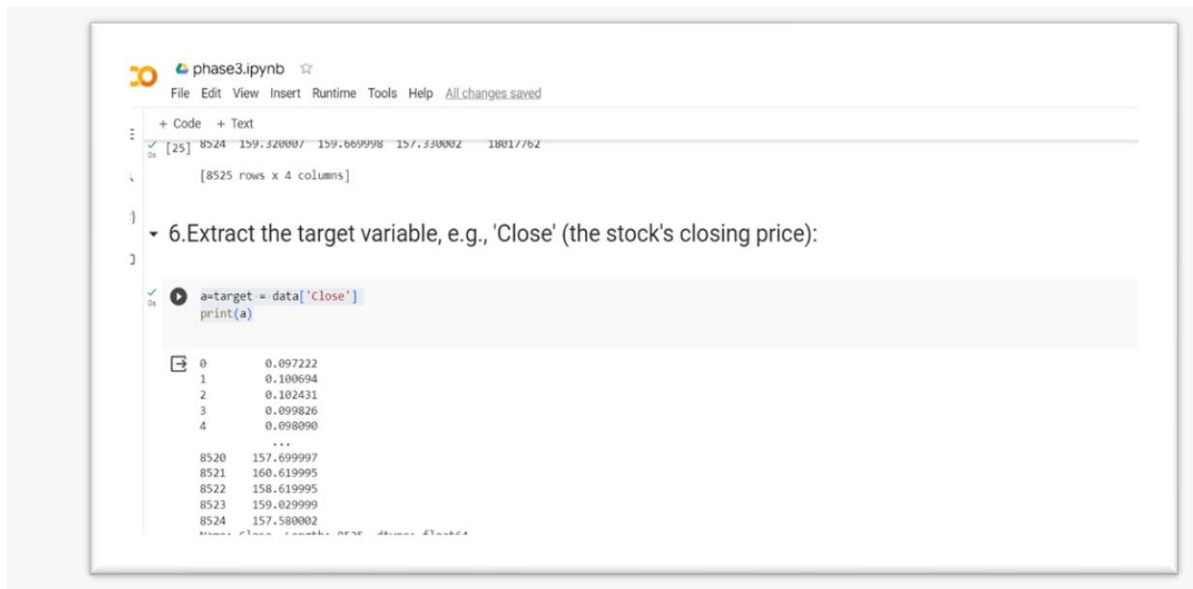
6.Extract the **target variable**, e.g., 'Close' (the stock's closing price):

The target variable, which we aim to predict, is typically the closing price of the stock. We extract this target variable from the dataset:

INPUT:

```
a=target = data['Close']  
print(a)
```

OUTPUT:



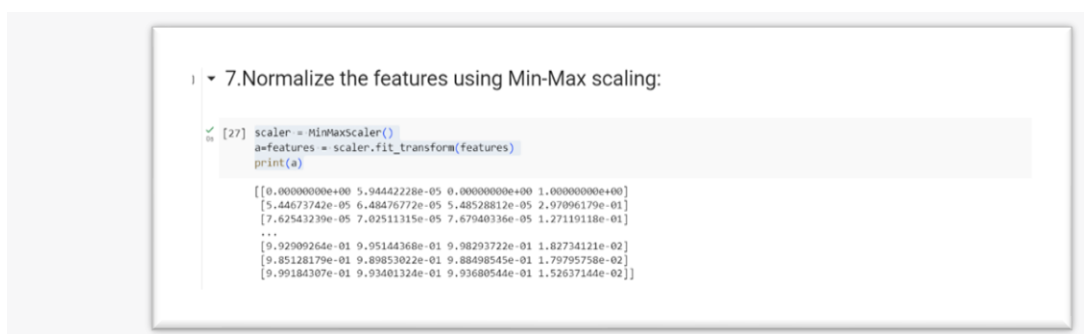
A screenshot of a Jupyter Notebook interface. The top bar shows the file name 'phase3.ipynb' and a star icon. Below the menu bar, there are tabs for '+ Code' and '+ Text'. The code cell shows a line of code: `a=target = data['Close']` followed by `print(a)`. The output of the code is displayed below the code cell, showing a list of values: `0 0.097222`, `1 0.100694`, `2 0.102431`, `3 0.099826`, `4 0.098090`, followed by an ellipsis `...`, and then `8520 157.699997`, `8521 160.619995`, `8522 158.619995`, `8523 159.029999`, and `8524 157.580002`. The output is truncated with `... 4 rows x 1 column`.

7.Normalize the features using Min-Max scaling:

INPUT:

```
scaler = MinMaxScaler()
a=features = scaler.fit_transform(features)
print(a)
```

OUTPUT:



A screenshot of a Jupyter Notebook interface. The top bar shows the file name 'phase3.ipynb' and a star icon. Below the menu bar, there are tabs for '+ Code' and '+ Text'. The code cell shows a line of code: `scaler = MinMaxScaler()` followed by `a=features = scaler.fit_transform(features)` and `print(a)`. The output of the code is displayed below the code cell, showing a list of values: `[0.00000000e+00 5.94442228e-05 0.00000000e+00 1.00000000e+00]`, `[5.44673742e-05 6.48476772e-05 5.48528812e-05 2.97096179e-01]`, `[7.62543239e-05 7.02511315e-05 7.67940336e-05 1.27119118e-01]`, followed by an ellipsis `...`, and then `[9.92909264e-01 9.95144368e-01 9.98293722e-01 1.82734121e-02]`, `[9.85128179e-01 9.89853022e-01 9.88498545e-01 1.79795758e-02]`, and `[9.99184307e-01 9.93401324e-01 9.93680544e-01 1.52637144e-02]`. The output is truncated with `... 4 rows x 1 column`.

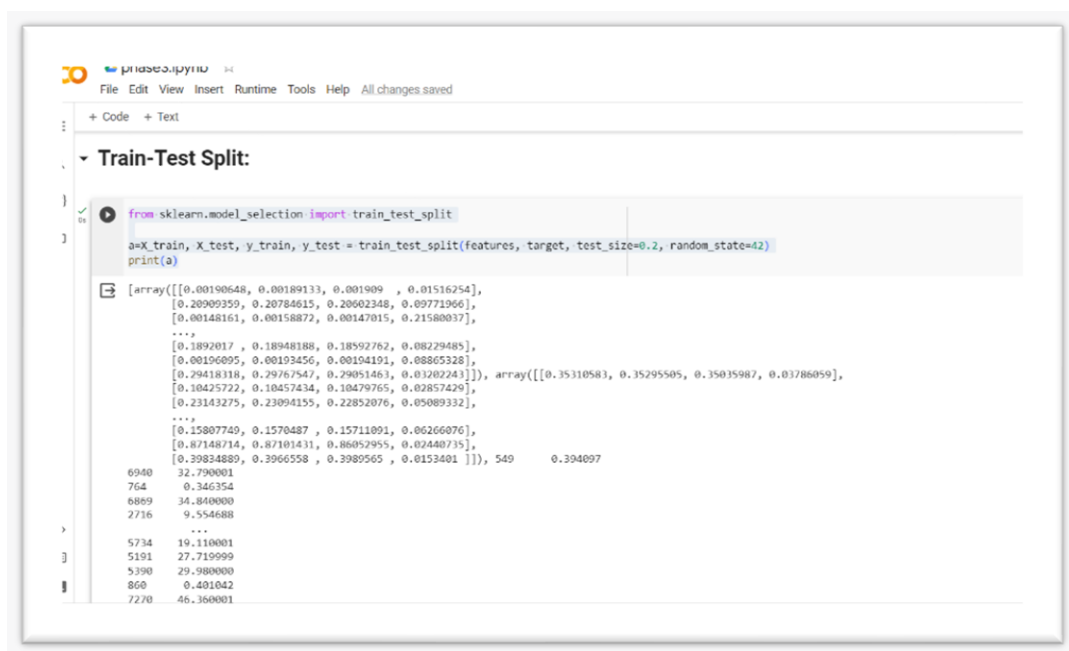
Train-Test Split:

INPUT:

```
from sklearn.model_selection import
train_test_split

a=X_train, X_test, y_train, y_test =
train_test_split(features, target, test_size=0.2,
random_state=42)
print(a)
```

OUTPUT:



```
from sklearn.model_selection import train_test_split

a=X_train, X_test, y_train, y_test = train_test_split(features, target, test_size=0.2, random_state=42)
print(a)
```

```
[array([[0.00190648, 0.00189133, 0.001909 , 0.01516254],
        [0.20909359, 0.20784615, 0.20602348, 0.09771966],
        [0.00148161, 0.00158872, 0.00147015, 0.21580037],
        ...,
        [0.1892017 , 0.18948188, 0.18592762, 0.08229485],
        [0.00196095, 0.00193456, 0.00194191, 0.08865328],
        [0.29418118, 0.29767547, 0.29051461, 0.03202243]]), array([[0.35110581, 0.35295505, 0.35035987, 0.03786059],
        [0.10425722, 0.10457434, 0.10479765, 0.02857429],
        [0.23143275, 0.23094155, 0.22852076, 0.05089332],
        ...,
        [0.15807749, 0.1570487 , 0.15711091, 0.06266076],
        [0.87148714, 0.87101431, 0.86052955, 0.02440735],
        [0.39834889, 0.3966558 , 0.3980565 , 0.0153401 ]]), 549      0.394097
```

LINEAR REGRESSION MODEL AND VISUALIZING THE RESULTS:

INPUT:

```
# Import the linear regression model
from sklearn.linear_model import
LinearRegression
from sklearn.metrics import mean_squared_error

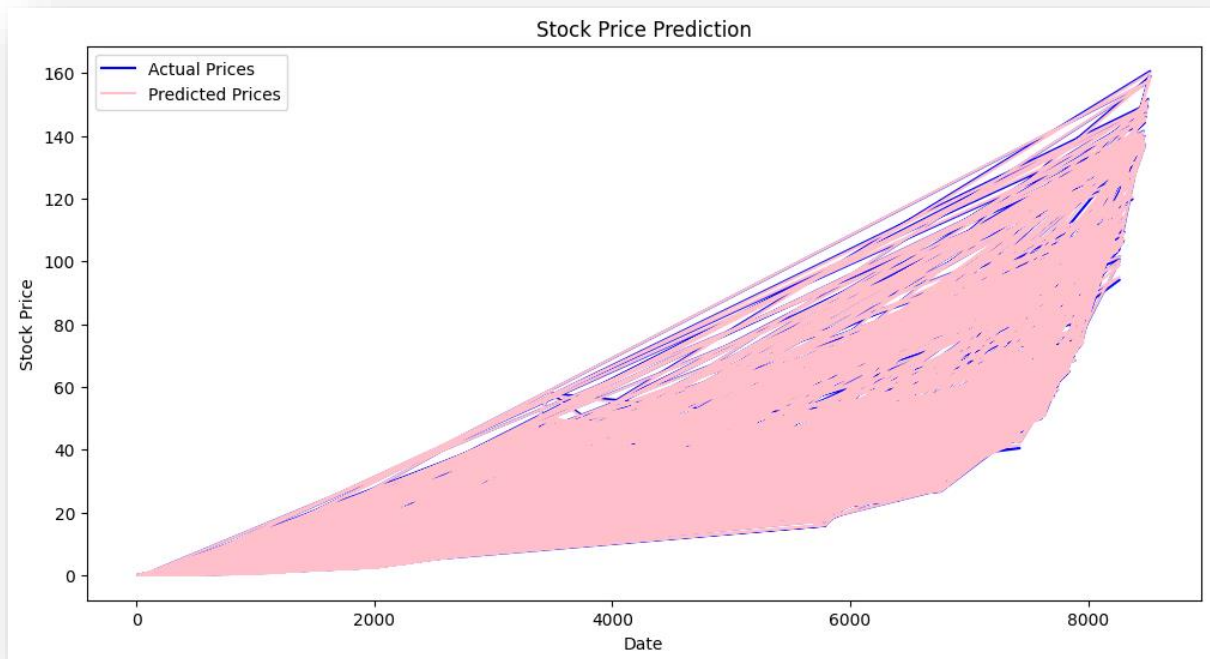
# Initialize and train the model
model = LinearRegression()
model.fit(X_train, y_train)

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

# Calculate the Mean Squared Error (MSE) to
evaluate the model
mse = mean_squared_error(y_test, y_pred)
print(f"Mean Squared Error: {mse}")

# Visualize the predicted vs. actual stock
prices
plt.figure(figsize=(12, 6))
plt.plot(y_test.index, y_test.values,
label='Actual Prices', color='blue')
plt.plot(y_test.index, y_pred, label='Predicted
Prices', color='pink')
plt.title('Stock Price Prediction')
plt.xlabel('Date')
plt.ylabel('Stock Price')
plt.legend()
plt.show()
```

OUTPUT:



CREATE A BAR CHART TO VISUALIZE THE DAILY TRADING VOLUME OVER TIME:

INPUT:

```
import pandas as pd
import matplotlib.pyplot as plt

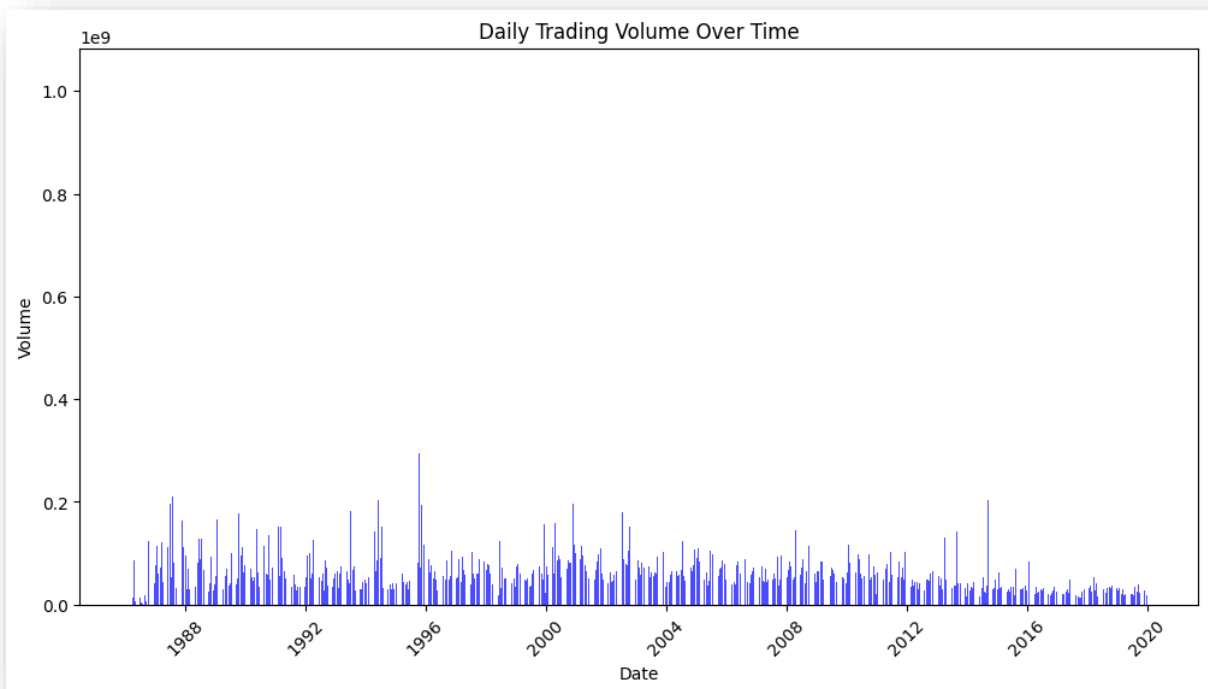
# Load the dataset (assuming you've already loaded it)
# Example data:
# data = pd.read_csv('stock_data.csv')

# Assuming 'Date' and 'Volume' columns exist in your dataset
dates = data['Date']
volume = data['Volume']
```

```
# Create a bar chart to visualize daily trading volume
plt.figure(figsize=(12, 6))
plt.bar(dates, volume, color='blue', alpha=0.7)
plt.xlabel('Date')
plt.ylabel('Volume')
plt.title('Daily Trading Volume Over Time')
plt.xticks(rotation=45) # Rotate x-axis labels for readability

plt.show()
```

OUTPUT:



CREATE A SCATTER PLOT TO VISUALIZE THE RELATIONSHIP BETWEEN THE STOCK'S CLOSING PRICE AND ITS TRADING VOLUME OVER TIME:

INPUT:

```
import pandas as pd
import matplotlib.pyplot as plt

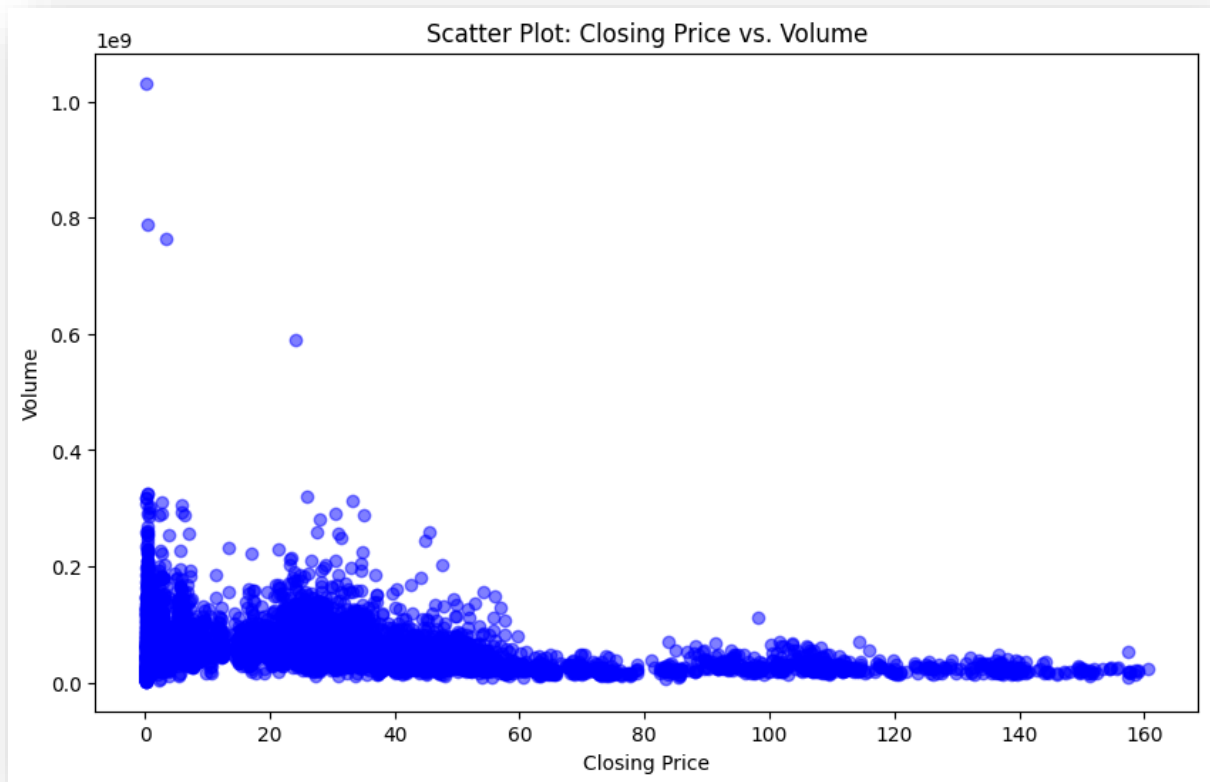
# Load the dataset (assuming you've already
loaded it)
# Example data:
# data = pd.read_csv('stock_data.csv')

# Assuming 'Close' and 'Volume' columns exist in
your dataset
close_price = data['Close']
volume = data['Volume']

# Create a scatter plot to visualize the
relationship between closing price and volume
plt.figure(figsize=(10, 6))
plt.scatter(close_price, volume, alpha=0.5,
color='blue')
plt.xlabel('Closing Price')
plt.ylabel('Volume')
plt.title('Scatter Plot: Closing Price vs.
Volume')

plt.show()
```

OUTPUT:



RESULT VISUALIZATION:

INPUT:

```
import pandas as pd
import matplotlib.pyplot as plt

# Load your stock price data into a
DataFrame (data assumed to be loaded)

# Convert the 'Date' column to a datetime
object
data['Date'] = pd.to_datetime(data['Date'])

# Set the 'Date' column as the index
data.set_index('Date', inplace=True)
```

```
# Create subplots for multiple
visualizations
fig, axes = plt.subplots(nrows=2, ncols=1,
figsize=(12, 8))

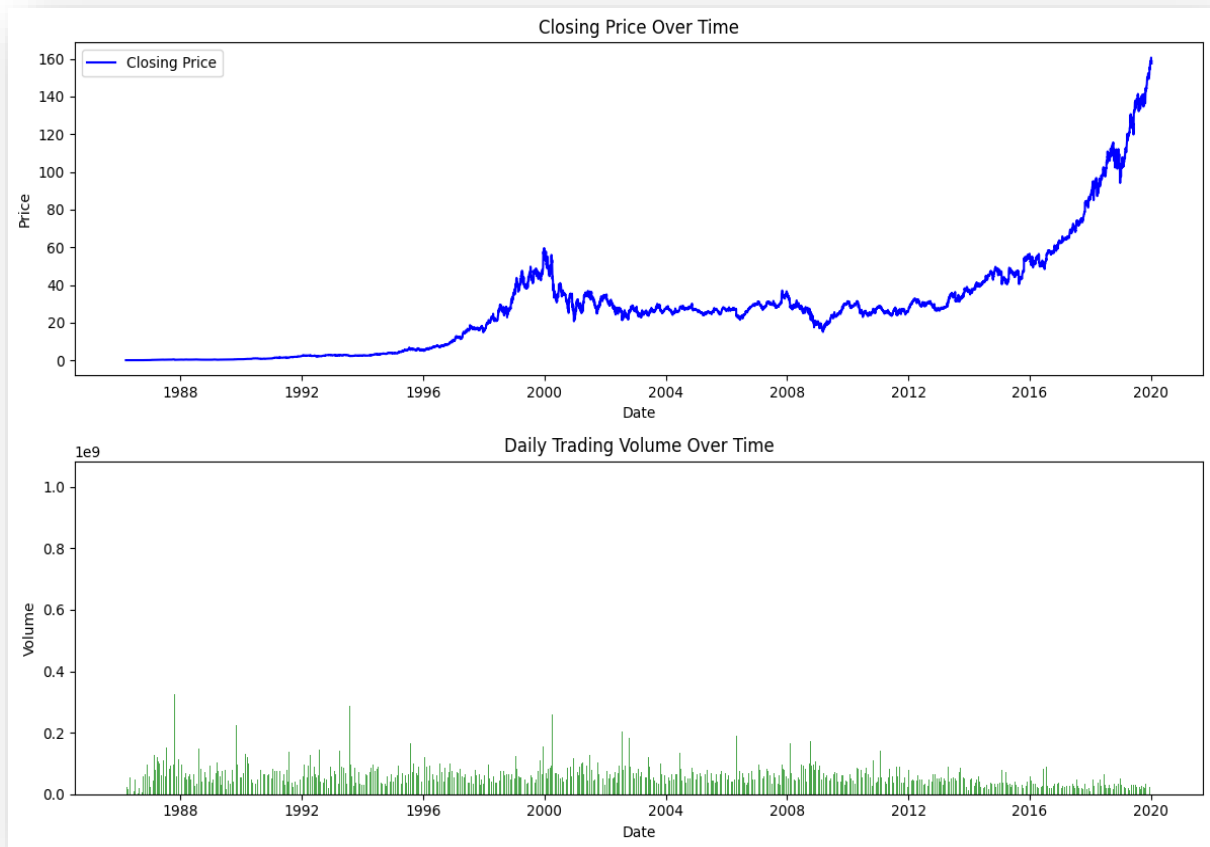
# Plot 1: Closing price over time
axes[0].plot(data.index, data['Close'],
label='Closing Price', color='blue')
axes[0].set_title('Closing Price Over Time')
axes[0].set_xlabel('Date')
axes[0].set_ylabel('Price')
axes[0].legend()

# Plot 2: Daily trading volume over time
axes[1].bar(data.index, data['Volume'],
color='green', alpha=0.7)
axes[1].set_title('Daily Trading Volume Over
Time')
axes[1].set_xlabel('Date')
axes[1].set_ylabel('Volume')

# Ensure the plots don't overlap
plt.tight_layout()

# Show the plots
plt.show()
```

OUTPUT:



CONCLUSION:

-In this section, we have successfully loaded and preprocessed the historical stock market data, making it ready for model development. In the next steps, you can proceed with building and evaluating your stock price prediction model using various machine learning or time-series analysis techniques.

-These initial data preprocessing steps are crucial for ensuring the quality and suitability of the dataset for your prediction model. By following these steps, you have set a solid foundation for your stock price prediction project.