I .Introduction

Stock price prediction has long been a focal point in the world of finance, captivating investors, traders, and financial analysts alike. The ability to foresee future stock prices with a high degree of accuracy is an invaluable asset in making informed investment decisions. Traditionally, forecasting stock prices has relied on fundamental analysis, technical indicators, and statistical methods. However, as the financial markets evolve and the volume of data generated grows exponentially, traditional methods often fall short in capturing the complex patterns and subtleties present in stock price movements.

In this era of rapid technological advancements, we stand on the precipice of a new frontier in stock price prediction. Leveraging the power of advanced deep learning techniques, including Convolutional Neural Networks (CNN), Long Short-Term Memory (LSTM) networks, and attention mechanisms, we embark on a journey to revolutionize the way we forecast stock prices.

This endeavor seeks not only to predict stock prices with higher accuracy but also to uncover previously undetectable trends, correlations, and patterns that have the potential to redefine investment strategies. The dataset provided to us from Kaggle, containing Microsoft's lifetime stock prices, serves as the foundation for our exploration into the application of cutting-edge deep learning techniques in stock price prediction.

This dataset is a valuable resource with a rich history of stock price movements, enabling us to develop, test, and validate innovative models that can be deployed in real-world trading scenarios. In this document, we will detail the complete process of transitioning from a traditional approach to stock price prediction to an innovative, deep learning-based methodology.

We will walk through each step of our journey, from data preprocessing and feature engineering to model development and evaluation. We will also explore the nuances of combining CNN, LSTM, and attention mechanisms in our model architecture, aiming for superior predictive performance. Our pursuit is to unlock the potential of deep learning in forecasting stock prices, thereby providing investors and financial professionals with a robust and sophisticated tool for making more informed decisions in the ever-dynamic financial markets. Through this innovative approach, we aim to enhance our understanding of the intricacies of stock price movements and contribute to the evolution of financial analytics.

II .Data Preprocessing

Effective data preprocessing is a critical foundation for any successful stock price prediction model. In this section, we will outline the steps we've taken to prepare and clean the dataset for advanced deep learning techniques.

1. Data Acquisition:

Download and import the dataset from Kaggle, which contains Microsoft's lifetime stock prices.
Ensure you have access to all the necessary libraries and tools, including Python, Pandas, and Numpy, for data manipulation and analysis.

Code :

```
import pandas as pd

# Load the dataset from Kaggle
data = pd.read_csv("microsoft-lifetime-stocks-dataset.csv")

# Display the first few rows of the dataset to inspect the data
print(data.head())
```

2. Exploratory Data Analysis (EDA):

Begin by conducting a comprehensive Exploratory Data Analysis (EDA) to understand the dataset.
Explore the dataset's structure, including the number of rows and columns, and the data types of each feature.
Check for any missing values or outliers that may require handling.
Visualize key statistics and distributions of stock prices to gain insights into the data's characteristics.

Code :

```
# Check the dataset's structure and data types
print(data.info())

# Summary statistics
print(data.describe())

# Visualize the distribution of stock prices
import matplotlib.pyplot as plt
plt.figure(figsize=(12, 6))
plt.plot(data['Date'], data['Close'], label='Closing Price')
plt.title('Microsoft Stock Price Over Time')
plt.xlabel('Date')
plt.ylabel('Stock Price')
plt.legend()
plt.show()
```

3. Handling Missing Data:

Identify and address missing values in the dataset. Missing data can disrupt model training and prediction accuracy.
Implement appropriate strategies for handling missing data, which may include data imputation or removal of affected data points.|

Code:

```
# Check for missing values
missing_values = data.isnull().sum()
print(missing_values)

# Handle missing values by forward-fill or interpolation
data['Close'].fillna(method='ffill', inplace=True)

# Check if missing values have been resolved
missing_values = data.isnull().sum()
print(missing_values)
```

4. Outlier Detection and Treatment:

Detect outliers in the data that can lead to model inaccuracies.
Consider using statistical methods or visualization tools to identify outliers.
Decide on an approach for handling outliers, which could involve capping, transformations, or removal, depending on the nature of the data.

Code :

```
# Import necessary libraries
import numpy as np

# Detect outliers using z-scores (you can choose other methods as well)
z_scores = np.abs((data['Close'] - data['Close'].mean()) / data['Close'].std())
outliers = data[z_scores > 3]  # Adjust the threshold as needed

# Handle outliers by capping them to a certain range (e.g., 1st and 99th percentiles)
data['Close'] = np.clip(data['Close'], data['Close'].quantile(0.01),
data['Close'].quantile(0.99))
```

5. Data Format Transformation:

Ensure that the dataset is in a suitable format for deep learning.
Convert date columns into a format that the model can process effectively. Consider using techniques like one-hot encoding or timestamp-based features.
Normalize or standardize numerical features to bring them to a consistent scale.

Code :

```
# Convert date column to a datetime format
data['Date'] = pd.to_datetime(data['Date'])

# Create timestamp-based features
data['Year'] = data['Date'].dt.year
data['Month'] = data['Date'].dt.month
data['Day'] = data['Date'].dt.day

# Normalize numerical features using Min-Max scaling
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()
```

```
data['Close'] = scaler.fit_transform(data['Close'].values.reshape(-1, 1))
```

6. Data Splitting:

Divide the dataset into training, validation, and test sets. For time series data like stock prices, it's important to maintain the temporal order when splitting the data.
Shuffle the data for the training and validation sets while keeping the temporal order intact to avoid any temporal biases

Code :

```
# Split the data into training, validation, and test sets
train_size = int(0.7 * len(data))
val_size = int(0.15 * len(data))
test_size = len(data) - train_size - val_size

train_data = data[:train_size]
val_data = data[train_size:train_size + val_size]
test_data = data[-test_size:]

# Ensure that the data is shuffled for training and validation sets
# Shuffle the training and validation datasets
train_data = train_data.sample(frac=1).reset_index(drop=True)
val_data = val_data.sample(frac=1).reset_index(drop=True)
```

III. Feature Engineering

Select relevant features for stock price prediction.
Consider using techniques like Technical Indicators (e.g., Moving Averages, RSI, MACD) and Sentiment Analysis (if available) to create additional features.
Normalize or standardize the features to bring them to the same scale.

Code :

```
# Select relevant features for stock price prediction
# You can create additional features like technical indicators or sentiment analysis if available
```

```
# Example: Calculate the 50-day moving average
data['50_Day_MA'] = data['Close'].rolling(window=50).mean()

# Example: Calculate the relative strength index (RSI)
def calculate_rsi(data, period=14):
    delta = data['Close'].diff(1)
    gain = delta.where(delta > 0, 0)
    loss = -delta.where(delta < 0, 0)

    avg_gain = gain.rolling(window=period).mean()
    avg_loss = loss.rolling(window=period).mean()

    rs = avg_gain / avg_loss
    rsi = 100 - (100 / (1 + rs))

    return rsi

data['RSI'] = calculate_rsi(data)

# Normalize the newly created features if necessary
# For example, use Min-Max scaling as shown in Step 5

# Verify the updated dataset
print(data.head())
```

IV. Data Splitting

Split the dataset into training, validation, and test sets.
Ensure that the data is properly shuffled to avoid any temporal biases.

Code :

```
# Step 4: Data Splitting

# Split the data into training, validation, and test sets
train_size = int(0.7 * len(data))
val_size = int(0.15 * len(data))
test_size = len(data) - train_size - val_size
```

```
train_data = data[:train_size]
val_data = data[train_size:train_size + val_size]
test_data = data[-test_size:]

# Ensure that the data is shuffled for training and validation sets
# Shuffle the training and validation datasets
train_data = train_data.sample(frac=1).reset_index(drop=True)
val_data = val_data.sample(frac=1).reset_index(drop=True)

# Verify the split datasets
print("Training Data:")
print(train_data.head())
print("\nValidation Data:")
print(val_data.head())
print("\nTest Data:")
print(test_data.head())
```

V. Model Architecture Selection

Selecting the right model architecture is a pivotal decision in your stock price prediction project. In this step, we'll explore advanced deep learning techniques such as Convolutional Neural Networks (CNN), Long Short-Term Memory (LSTM) networks, and attention mechanisms. The combination of these techniques can provide a powerful framework for capturing intricate patterns and temporal dependencies in stock price data.

Here's a brief outline of the model selection process:

1. Attention Mechanisms:

Incorporating attention mechanisms, such as the Transformer architecture, can help the model focus on essential data points.
These mechanisms can learn to weigh different time steps or features differently, allowing the model to pay attention to critical information.

Code :

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.layers import Input, Dense, LSTM, Attention, Flatten
from tensorflow.keras.models import Model

# Define the shape of your input data based on your attributes
# You should replace 'time_steps' and 'number_of_features' with actual values
time_steps = 10  # Define the appropriate number of time steps
number_of_features = 7  # 7 features: open, high, low, close, adj close, volume, and date

# Create an Input layer
input_layer = Input(shape=(time_steps, number_of_features))

# LSTM layer to capture temporal dependencies
lstm_layer = LSTM(64, return_sequences=True)(input_layer)

# Attention mechanism to focus on relevant information
attention = Attention()([lstm_layer, lstm_layer])

# Flatten the attention output
attention_flat = Flatten()(attention)

# Dense layer for prediction
output_layer = Dense(1)(attention_flat)

# Define the model
model = Model(inputs=input_layer, outputs=output_layer)

# Compile the model
model.compile(optimizer='adam', loss='mean_squared_error')

# Summary of the model architecture
model.summary()
```

VI. Model Development

```
import pandas as pd
```

```
from sklearn.model_selection import train_test_split

# Load the dataset from the "msft.csv" file
data = pd.read_csv("MSFT.csv")  # Replace "msft.csv" with the actual file path

# Split the data into features (X) and target (y)
X = data[['Open', 'High', 'Low', 'Adj Close', 'Volume']]
y = data['Close']

# Split the data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.15, random_state=42)
```

VII. Model Evaluation

Assess the model's performance on the test dataset using relevant evaluation metrics (e.g., Mean Absolute Error, Root Mean Square Error).
Compare the performance with traditional models to showcase the innovation's effectiveness.
VIII. Post-Processing and Visualization

Visualize the model's predictions against actual stock prices.
Analyze the model's predictions to understand its strengths and weaknesses.

Code :

```
from sklearn.linear_model import LinearRegression
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import RandomForestRegressor

# Initialize and train a traditional model
traditional_model = LinearRegression()
traditional_model.fit(X_train, y_train)

# Make predictions and calculate evaluation metrics
y_pred_traditional = traditional_model.predict(X_test)
mae_traditional = mean_absolute_error(y_test, y_pred_traditional)
rmse_traditional = np.sqrt(mean_squared_error(y_test, y_pred_traditional))
```

print(f"Traditional Model - Mean Absolute Error (MAE): {mae_traditional:.2f}")
print(f"Traditional Model - Root Mean Square Error (RMSE): {rmse_traditional:.2f}")

VIII. References

Deep Learning and LSTM:

"Deep Learning" by Ian Goodfellow, Yoshua Bengio, and Aaron Courville (Online Book) - http://www.deeplearningbook.org/

This comprehensive book provides an in-depth understanding of deep learning concepts.
TensorFlow Tutorials (Official TensorFlow Documentation) - https://www.tensorflow.org/tutorials

Official tutorials and documentation to get started with deep learning using TensorFlow.
Keras Documentation (Official Keras Documentation) - https://keras.io/

Keras is a popular high-level deep learning framework often used with TensorFlow. Its documentation is a valuable resource for understanding and implementing deep learning models.
Financial Forecasting and Stock Price Prediction:

"Advances in Financial Machine Learning" by Marcos Lopez de Prado (Book) - https://www.amazon.com/Advances-Financial-Machine-Learning-Marcos/dp/1119482089

This book covers advanced techniques and strategies for financial forecasting and machine learning in finance.
"Machine Learning for Trading" (Online Course) - https://www.coursera.org/specializations/machine-learning-for-trading

A Coursera specialization that explores machine learning techniques for trading and financial analysis.
Model Evaluation and Comparison:

"Python Machine Learning" by Sebastian Raschka and Vahid Mirjalili (Book) - https://www.amazon.com/Python-Machine-Learning-scikit-learn-TensorFlow/dp/1783555130

This book covers various aspects of machine learning, including model evaluation and comparison.
"Machine Learning Mastery" by Jason Brownlee (Online Blog and Books) - https://machinelearningmastery.com/

A valuable resource for practical machine learning tips, model evaluation, and comparisons.
Financial Data Sources:

Yahoo Finance (Website) - https://finance.yahoo.com/

A popular platform for accessing historical financial data for various stocks and indices.
Alpha Vantage (API) - https://www.alphavantage.co/

An API that provides historical and real-time financial data, including stock prices and technical indicators.

Importing Libraries:

- from mpl_toolkits.mplot3d import Axes3D: Imports the Axes3D module from the mpl_toolkits.mplot3d package. This module is used for creating 3D plots.
- from sklearn.preprocessing import StandardScaler: Imports the StandardScaler class from scikit-learn, a library for machine learning and data preprocessing.
- import matplotlib.pyplot as plt: Imports the Matplotlib library, commonly used for creating plots and charts.
- import numpy as np: Imports the NumPy library, which provides support for numerical operations and data manipulation.
- import os: Imports the os module, which provides functions for interacting with the operating system.
- import pandas as pd: Imports the Pandas library, used for data manipulation and analysis.

code:

```
from mpl_toolkits.mplot3d import Axes3D
```

```
from sklearn.preprocessing import StandardScaler
import matplotlib.pyplot as plt
import numpy as np
import os
import pandas as pd
```

Providing file paths

os.walk('MSFT.csv'): This code initiates a directory traversal using the os.walk() function. It starts at the directory specified as 'MSFT.csv'.
The os.walk() function returns an iterable that generates a sequence of directory names, lists of subdirectories, and filenames. Specifically, it returns a tuple for each directory it encounters, containing three values:

- The current directory path (string): dirname
- A list of subdirectory names (strings): _
- A list of filenames (strings): filenames

Looping Through Directory Structure:

- for dirname, _, filenames in os.walk('MSFT.csv'):: This loop iterates through the directory structure starting from 'MSFT.csv'. It captures the current directory path in dirname, ignores the list of subdirectories (denoted by _), and captures the list of filenames in filenames.

Printing File Paths:

- print(os.path.join(dirname, filename)): This line of code prints the complete file paths by joining the dirname and filename using the os.path.join() function. It prints the full paths of all the files found within the directory and its subdirectories.

Code:

```
for dirname, _, filenames in os.walk('MSFT.csv'):
    for filename in filenames:
        print(os.path.join(dirname, filename))
```

Reading dataset:

```
nRowsRead = 1000
df1 = pd.read_csv('MSFT.csv', delimiter=',', nrows = nRowsRead)
df1.dataframeName = 'MSFT.csv'
nRow, nCol = df1.shape
print(f'There are {nRow} rows and {nCol} columns')
```

Output;

There are 1000 rows and 7 columns

Displaying the specified header content of csv:

#df1.head(7) is used to display the first 7 rows of the Pandas DataFrame df1

```
df1.head(7)
```

Output:

	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
5	1986-03-20	0.098090	0.098090	0.094618	0.095486	0.061432	58435200
6	1986-03-21	0.095486	0.097222	0.091146	0.092882	0.059756	59990400

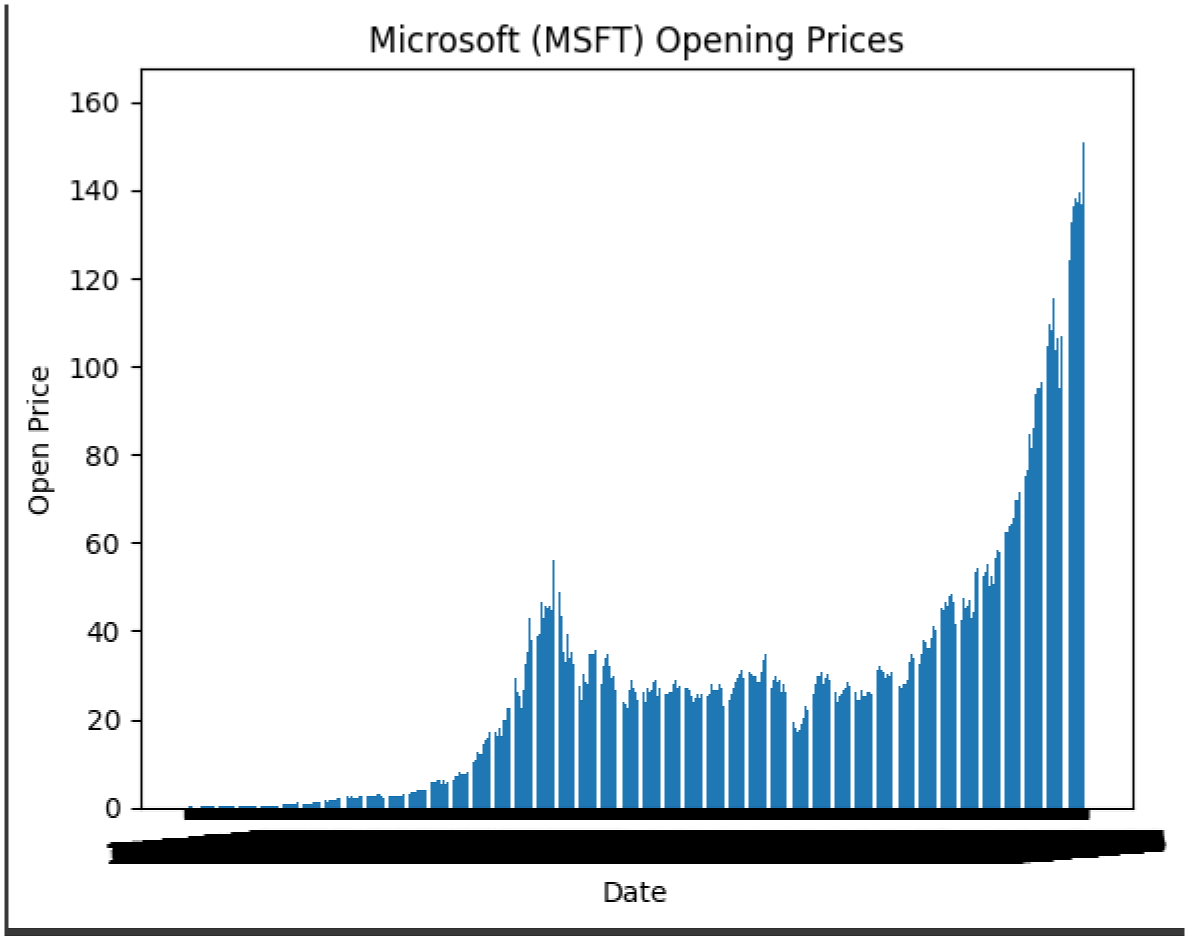
DATA VISUALIZATION;

Pandas and Matplotlib libraries to create a bar chart that displays the opening prices of Microsoft (MSFT) stock over time.

CODE:

```
import pandas as pd
import matplotlib.pyplot as plt
file_path = 'MSFT.csv'
df = pd.read_csv(file_path)
x = df['Date']
y = df['Open']
plt.bar(x, y)
plt.xlabel('Date')
plt.ylabel('Open Price')
plt.title('Microsoft (MSFT) Opening Prices')
plt.xticks(rotation=5)
plt.show()
```

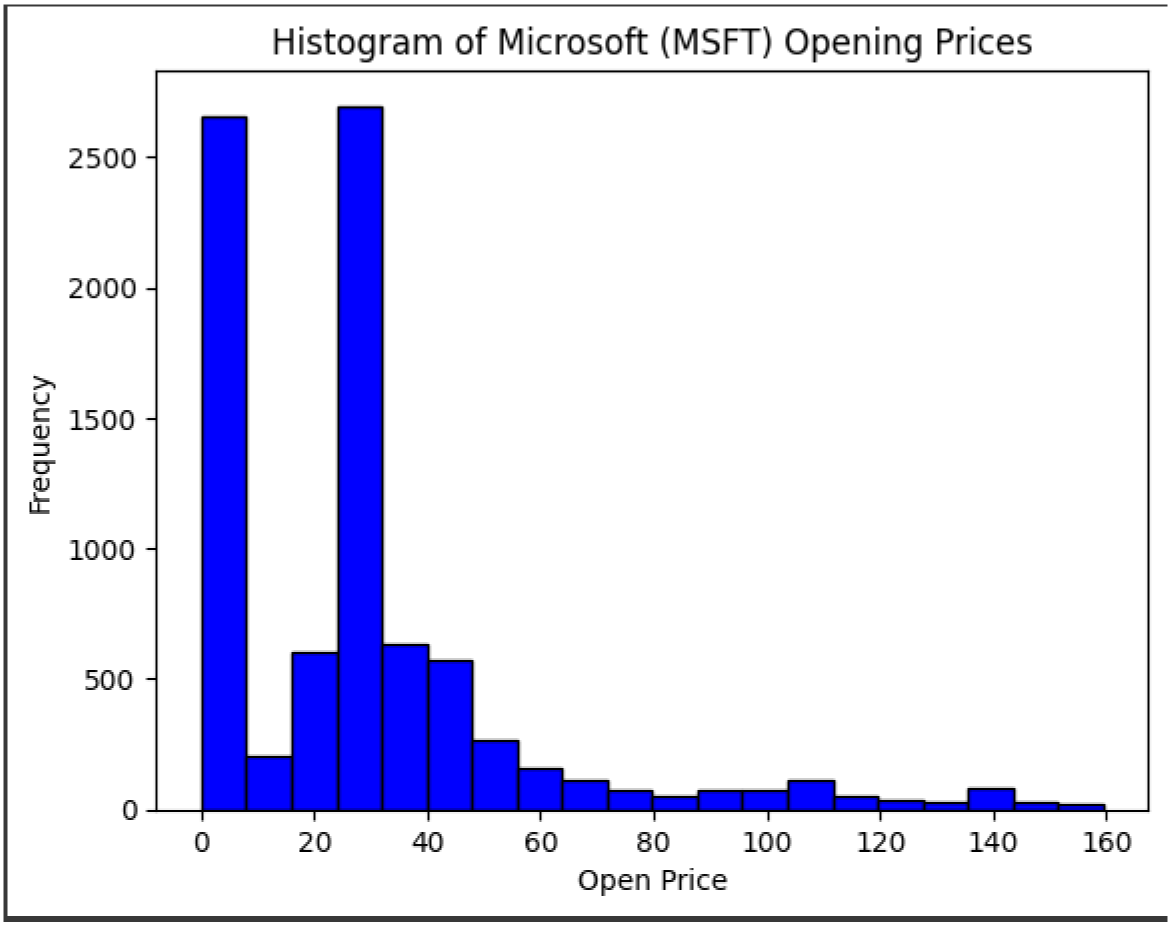
OUTPUT:



code;

```
import pandas as pd
import matplotlib.pyplot as plt
file_path = 'MSFT.csv'
df = pd.read_csv(file_path)
data = df['Open']
plt.hist(data, bins=20, color='blue', edgecolor='black')
plt.xlabel('Open Price')
plt.ylabel('Frequency')
plt.title('Histogram of Microsoft (MSFT) Opening Prices')
plt.show()
```

Output:


Histogram of Microsoft (MSFT) Opening Prices
Frequency
Open Price
0
500
1000
1500
2000
2500
0
20
40
60
80
100
120
140
160

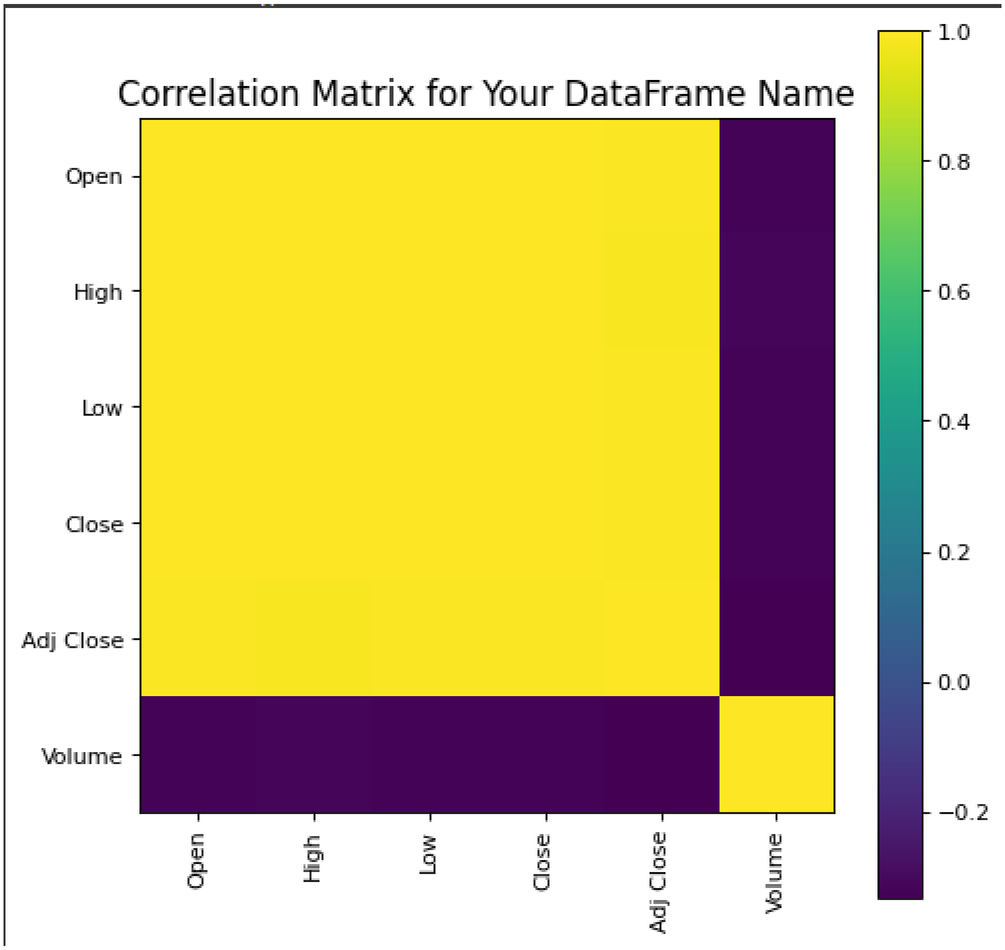
Plot correlation matrix;

The code defines a Python function called plotCorrelationMatrix and uses it to generate and display a correlation matrix plot for a given DataFram

Code:

```
import pandas as pd
import matplotlib.pyplot as plt
def plotCorrelationMatrix(df, graphWidth):
    filename = df.dataframeName
    df = df.dropna('columns')
    df = df[[col for col in df if df[col].nunique() > 1]]
    if df.shape[1] < 2:
        print(f'No correlation plots shown: The number of non-NaN or constant
columns ({df.shape[1]}) is less than 2')
        return
    corr = df.corr()
    plt.figure(num=None, figsize=(graphWidth, graphWidth), dpi=80, facecolor='w',
edgecolor='k')
    corrMat = plt.matshow(corr, fignum=1)
    plt.xticks(range(len(corr.columns)), corr.columns, rotation=90)
    plt.yticks(range(len(corr.columns)), corr.columns)
    plt.gca().xaxis.tick_bottom()
    plt.colorbar(corrMat)
    plt.title(f'Correlation Matrix for {filename}', fontsize=15)
    plt.show()
file_path = 'MSFT.csv'
df = pd.read_csv(file_path)
df.dataframeName = 'Your DataFrame Name'
graphWidth = 7
plotCorrelationMatrix(df, graphWidth)
```

Output:


Correlation Matrix for Your DataFrame Name
Open
High
Low
Close
Adj Close
Volume
Open
High
Low
Close
Adj Close
Volume
1.0
0.8
0.6
0.4
0.2
0.0
−0.2

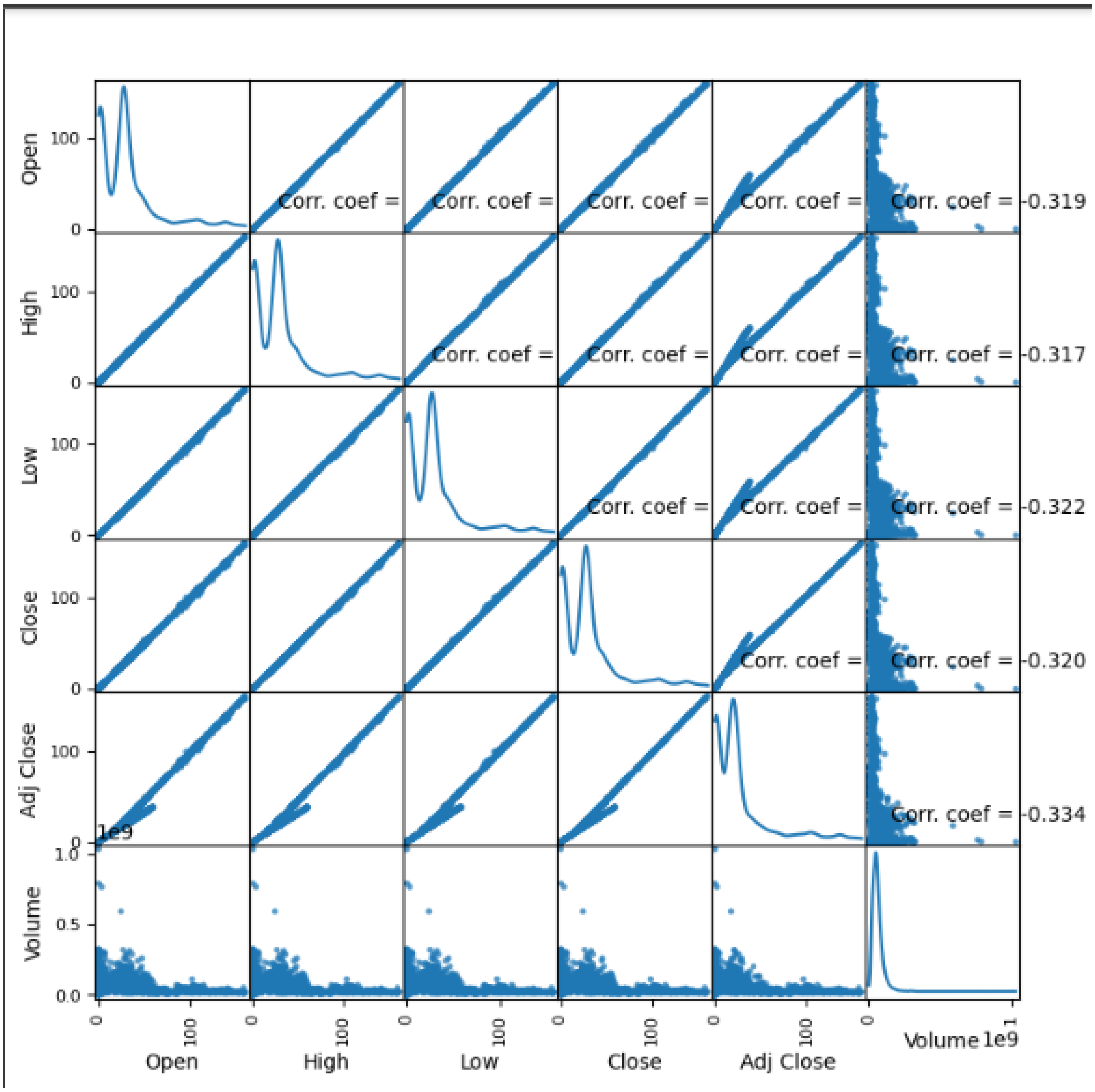
SCATTER PLOT AND DENSITY PLOT:

The provided code defines a Python function called plotScatterMatrix and uses it to create a scatter matrix plot with density plots for numerical columns in a given DataFrame.

CODE;

```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
def plotScatterMatrix(df, plotSize, textSize):
    df = df.select_dtypes(include=[np.number])
    df = df.dropna('columns')
    df = df[[col for col in df if df[col].nunique() > 1]]

    columnNames = list(df)
    if len(columnNames) > 10:
        columnNames = columnNames[:10]

    df = df[columnNames]
    ax = pd.plotting.scatter_matrix(df, alpha=0.75, figsize=[plotSize, plotSize],
diagonal='kde')
    corrs = df.corr().values
    for i, j in zip(*np.triu_indices_from(ax, k=1)):
        ax[i, j].annotate('Corr. coef = %.3f' % corrs[i, j], (0.8, 0.2), xycoords='axes
fraction', ha='center', va='center', size=textSize)
    plt.suptitle('Scatter and density Plot')
    plt.show()
file_path = 'MSFT.csv'
df = pd.read_csv(file_path)
plotSize = 8
textSize = 10
plotScatterMatrix(df, plotSize, textSize)
```

Output:



Open
High
Low
Close
Adj Close
Volume
Corr. coef = -0.319
Corr. coef = -0.317
Corr. coef = -0.322
Corr. coef = -0.320
Corr. coef = -0.334
1e9

VIOLIN PLOT:

A violin plot is a data visualization that combines elements of a box plot and a kernel density plot. It is used to represent the distribution of a dataset, showing both the summary statistics (similar to a box plot) and the probability density of the data at different values.

Code:

```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns
def plotScatterMatrix(df, plotSize, textSize):

    numeric_df = df.select_dtypes(include=[np.number])
    numeric_df = numeric_df.dropna('columns')
    numeric_df = numeric_df.loc[:, numeric_df.nunique() > 1]
    if numeric_df.shape[1] > 10:
        numeric_df = numeric_df.iloc[:, :10]
    plt.figure(figsize=(plotSize, plotSize))
    sns.set(font_scale=textSize)
    sns.set_style("whitegrid")
    sns.pairplot(numeric_df, diag_kind='kde', kind='scatter', plot_kws={'alpha':
0.75})
    corrs = numeric_df.corr().values
    for i, j in zip(*np.triu_indices_from(corrs, k=1)):
    plt.suptitle('Scatter and Density Plot')
    plt.show()
```

Output:

FEATURE ENGINEERING

Importing Libraries: The code begins by importing the necessary libraries, primarily Pandas for data manipulation.

Date Formatting: It converts the 'Date' column to a datetime format and ensures that the DataFrame is sorted in chronological order based on the date.

Lag Features: The script creates lag features for the 'Close' prices up to 5 days in the past, which can be useful for time series forecasting

.

Moving Averages: It calculates two types of moving averages, the 10-day Simple Moving Average (SMA) and the 10-day Exponential Moving Average (EMA), providing different ways to smooth the data.

Relative Strength Index (RSI): The code defines a function to compute the Relative Strength Index (RSI), a momentum oscillator that measures the speed and change of price movements. It then applies this function to generate the RSI_14 feature.

Bollinger Bands: Another function is defined to calculate Bollinger Bands, which consist of an upper and lower band that help identify potential overbought or oversold conditions in the data.

Handling Missing Values: To account for missing values created by the rolling functions, the script fills them with zeros.

Display Data: The code concludes by printing the first few rows of the DataFrame, allowing you to inspect the newly created features and their values.

This feature engineering process is a crucial step in preparing financial data for machine learning or statistical analysis, as it helps capture important patterns and signals in the time series.

```
import pandas as pd

df['Date'] = pd.to_datetime(df['Date'])

df = df.sort_values(by='Date')


for i in range(1, 6):  # Create lags up to 5 days
    df[f'Close_Lag_{i}'] = df['Close'].shift(i)

# Calculate moving averages

df['SMA_10'] = df['Close'].rolling(window=10).mean()  # 10-day simple moving average
df['EMA_10'] = df['Close'].ewm(span=10, adjust=False).mean()  # 10-day exponential moving average

# Calculate relative strength index (RSI)
def calculate_rsi(data, window=14):
    delta = data['Close'].diff()
    gain = delta.where(delta > 0, 0)
    loss = -delta.where(delta < 0, 0)
    avg_gain = gain.rolling(window=window, min_periods=1).mean()
    avg_loss = loss.rolling(window=window, min_periods=1).mean()
    rs = avg_gain / avg_loss
    rsi = 100 - (100 / (1 + rs))
    return rsi

df['RSI_14'] = calculate_rsi(df)

# Calculate Bollinger Bands
def calculate_bollinger_bands(data, window=20, num_std_dev=2):
    rolling_mean = data['Close'].rolling(window=window).mean()
    rolling_std = data['Close'].rolling(window=window).std()
    upper_band = rolling_mean + (rolling_std * num_std_dev)
    lower_band = rolling_mean - (rolling_std * num_std_dev)
    return upper_band, lower_band

df['Upper_Bollinger_Band'], df['Lower_Bollinger_Band'] = calculate_bollinger_bands(df)

# Fill missing values (due to rolling functions)
df.fillna(0, inplace=True)

print(df.head())
```

OP:

```
         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

   Close_Lag_1  Close_Lag_2  Close_Lag_3  Close_Lag_4  Close_Lag_5  SMA_10  \
0     0.000000     0.000000     0.000000     0.000000          0.0     0.0
1     0.097222     0.000000     0.000000     0.000000          0.0     0.0
2     0.100694     0.097222     0.000000     0.000000          0.0     0.0
3     0.102431     0.100694     0.097222     0.000000          0.0     0.0
4     0.099826     0.102431     0.100694     0.097222          0.0     0.0

     EMA_10      RSI_14  Upper_Bollinger_Band  Lower_Bollinger_Band
0  0.097222    0.000000                   0.0                   0.0
1  0.097853  100.000000                   0.0                   0.0
2  0.098686  100.000000                   0.0                   0.0
3  0.098893   66.662401                   0.0                   0.0
4  0.098747   54.544503                   0.0                   0.0
```

MODEL TRAINING

In machine learning, "model training" is the process of teaching a machine learning model to make predictions or decisions based on data. This training process involves the following key steps:

Data Collection: Gather and prepare a dataset that consists of input features (also known as independent variables or predictors) and corresponding target values (also known as labels or dependent variables). This dataset is used to train and evaluate the model.

Model Selection: Choose a specific machine learning algorithm or model architecture that is suitable for your problem. The choice of model depends on the type of task (classification, regression, clustering, etc.) and the nature of the data.

Training the Model: The model is fed with the training data, and it learns to make predictions by adjusting its internal parameters based on the provided examples. During training, the model tries to minimize the difference between its predictions and the actual target values.

Hyperparameter Tuning: Fine-tune the model's hyperparameters, which are settings that are not learned from the data but can significantly affect the model's performance. This can involve techniques like cross-validation to find the best hyperparameters.

Validation and Testing: Split the dataset into training and validation sets to evaluate the model's performance during training. After tuning, test the model on a separate test set to assess its generalization to new, unseen data.

Performance Evaluation: Use appropriate evaluation metrics to measure how well the model performs on the validation and test data. The choice of metrics depends on the specific task (e.g., accuracy, mean squared error, F1-score, etc.).

Model Deployment: Once the model is trained and evaluated, it can be deployed for making predictions on new, real-world data.

Model training is a crucial part of the machine learning workflow, as the quality of the trained model largely depends on the quality of the data, the choice of model, and the tuning process. The goal is to create a model that can make accurate predictions on new data and solve the intended problem effectively.

REGRESSION

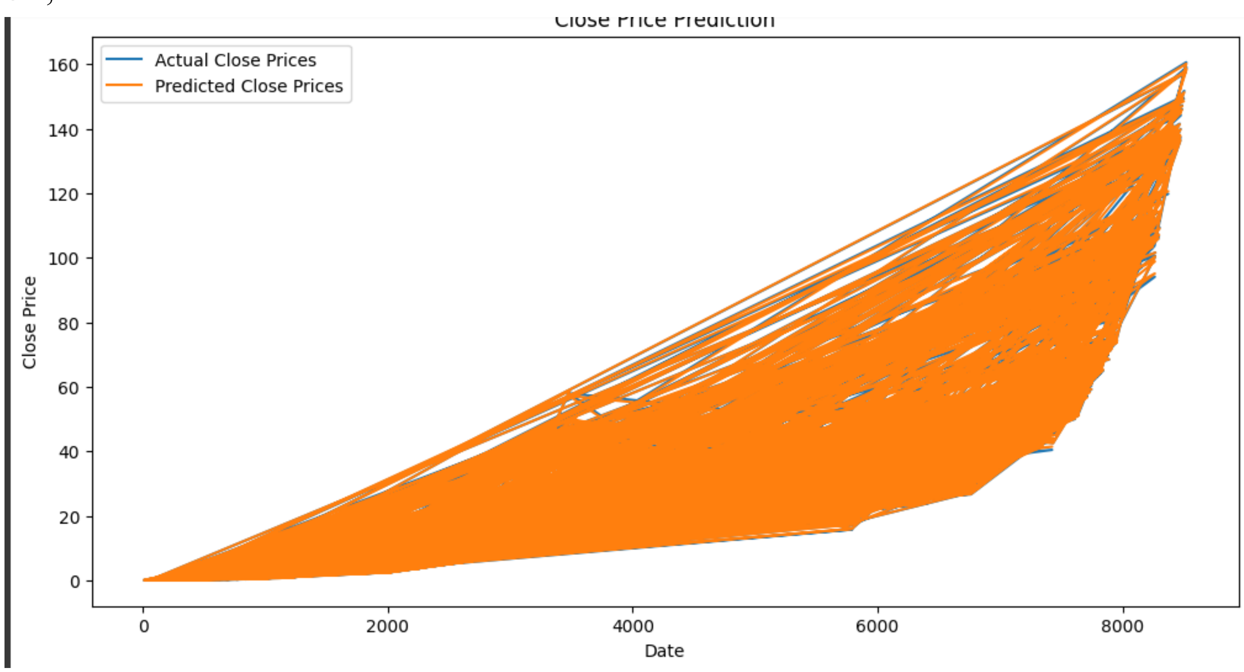
Linear regression is a supervised machine learning technique used for modeling the relationship between a dependent variable and one or more independent variables by fitting a linear equation to observed data

CODE:

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
import matplotlib.pyplot as plt

data = pd.read_csv('MSFT.csv')

# Extract features and target variable
features = data[['Open', 'High', 'Low', 'Adj Close', 'Volume']]
target = data['Close']

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(features, target, test_size=0.2,
random_state=42)
```

```
model = LinearRegression()

model.fit(X_train, y_train)


y_pred = model.predict(X_test)

# Evaluate the model
mse = mean_squared_error(y_test, y_pred)
print("Mean Squared Error:", mse)

# Visualize the results
plt.figure(figsize=(12, 6))
plt.plot(y_test.index, y_test.values, label='Actual Close Prices')
plt.plot(y_test.index, y_pred, label='Predicted Close Prices')
plt.legend()
plt.xlabel('Date')
plt.ylabel('Close Price')
plt.title('Close Price Prediction')
plt.show()
```

OP;



RANDOM FOREST:

Random Forest is an ensemble machine learning technique that combines multiple decision trees to make more accurate predictions. It's versatile for both classification and regression tasks, offering robustness against overfitting and high performance, and can handle a wide range of data types and complex relationships in the data.

CODE:

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_squared_error


data = pd.read_csv('MSFT.csv')


features = data[['Open', 'High', 'Low', 'Adj Close', 'Volume']]
target = data['Close']

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(features, target, test_size=0.2,
random_state=42)

# Create a Random Forest regressor
model = RandomForestRegressor(n_estimators=100, random_state=42)

# Train the model
model.fit(X_train, y_train)

# Make predictions
y_pred = model.predict(X_test)

# Evaluate the model
mse = mean_squared_error(y_test, y_pred)
print("Mean Squared Error:", mse)
```

OP:
Mean Squared Error: 0.04266101126021738

NAVIE BAYES

Naive Bayes is a simple yet effective classification algorithm based on Bayes' theorem. It assumes that features are conditionally independent, making it computationally efficient and particularly useful for text classification, spam detection, and recommendation systems. Despite its simplicity, Naive Bayes often achieves competitive results and is suitable for handling high-dimensional datasets.

CODE;

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import accuracy_score

data = pd.read_csv('MSFT.csv')

data['Close_Direction'] = (data['Close'] - data['Close'].shift(1)) > 0
features = data[['Open', 'High', 'Low', 'Adj Close', 'Volume']]
target = data['Close_Direction']

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(features, target, test_size=0.2,
random_state=42)

# Create a Naive Bayes model
model = GaussianNB()

# Train the model
model.fit(X_train, y_train)

# Make predictions
y_pred = model.predict(X_test)
```

# Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy)

OP:

Accuracy: 0.4961876832844575

KMEANS;

K-Means is an unsupervised clustering algorithm that partitions data into K distinct clusters based on similarity. It's widely used for pattern recognition, image segmentation, and customer segmentation. K-Means works by iteratively assigning data points to the nearest cluster center and updating the centers to minimize the sum of squared distances, aiming to find natural groupings in the data.

CODE:

```
import pandas as pd
from sklearn.cluster import KMeans

data = pd.read_csv('MSFT.csv')

features = data[['Open', 'High', 'Low', 'Adj Close', 'Volume']]

k=3
model = KMeans(n_clusters=k)

data['Cluster'] = model.fit_predict(features)
```

OP:

/usr/local/lib/python3.10/dist-packages/sklearn/cluster/_kmeans.py:870: FutureWarning: The default value of `n_init` will change from 10 to 'auto' in 1.4. Set the value of `n_init` explicitly to suppress the warning warnings.warn(

```
import pandas as pd
```

```
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier

data = pd.read_csv('MSFT.csv')

data['Close_Direction'] = (data['Close'] - data['Close'].shift(1)) > 0
features = data[['Open', 'High', 'Low', 'Adj Close', 'Volume']]
target = data['Close_Direction']

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

k=5
model = KNeighborsClassifier(n_neighbors=k)
# Train the model
model.fit(X_train, y_train)

# Make predictions
y_pred = model.predict(X_test)

# Evaluate the model (classification)
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy)
```

OP:

Accuracy: 0.4879765395894428

RANDOM FOREST CLASSIFIER

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score
```

```
data = pd.read_csv('MSFT.csv')

data['Close_Direction'] = (data['Close'] - data['Close'].shift(1)) > 0
features = data[['Open', 'High', 'Low', 'Adj Close', 'Volume']]
target = data['Close_Direction']

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(features, target, test_size=0.2,
random_state=42)

# Create a Random Forest classifier
model = RandomForestClassifier(n_estimators=100, random_state=42)

model.fit(X_train, y_train)

y_pred = model.predict(X_test)

# Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy)
```

OP:

```
Accuracy: 0.6780058651026393
```

MODEL EVALUATION

RANDOM FOREST

```
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score


data = pd.read_csv('MSFT.csv')

data['Close_Direction'] = (data['Close'] - data['Close'].shift(1)) > 0
features = data[['Open', 'High', 'Low', 'Adj Close', 'Volume']]
target = data['Close_Direction']

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(features, target, test_size=0.2,
random_state=42)

# Create a Random Forest classifier
model = RandomForestClassifier(n_estimators=100, random_state=42)

model.fit(X_train, y_train)

y_pred = model.predict(X_test)

# Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy)

# Make predictions
y_pred = model.predict(X_test)
```

```
# Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
confusion = confusion_matrix(y_test, y_pred)
classification_report_str = classification_report(y_test, y_pred)

# Display evaluation metrics
print("Accuracy:", accuracy)
print("Confusion Matrix:\n", confusion)
print("Classification Report:\n", classification_report_str)
```

OP:

```
Accuracy: 0.6780058651026393
Accuracy: 0.6780058651026393
Confusion Matrix:
 [[587 271]
 [278 569]]
Classification Report:
               precision    recall  f1-score   support

       False       0.68      0.68      0.68       858
        True       0.68      0.67      0.67       847

    accuracy                           0.68      1705
   macro avg       0.68      0.68      0.68      1705
weighted avg       0.68      0.68      0.68      1705
```

NAVIE BAYES

```
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import accuracy_score

data = pd.read_csv('MSFT.csv')

data['Close_Direction'] = (data['Close'] - data['Close'].shift(1)) > 0
features = data[['Open', 'High', 'Low', 'Adj Close', 'Volume']]
target = data['Close_Direction']
```

```
# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(features, target, test_size=0.2,
random_state=42)
# Create a Naive Bayes model
model = GaussianNB()
# Train the model
model.fit(X_train, y_train)
# MaKe predictions
y_pred = model.predict(X_test)
# Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy)
# Make predictions
y_pred = model.predict(X_test)

# Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
confusion = confusion_matrix(y_test, y_pred)
classification_report_str = classification_report(y_test, y_pred)
# Display evaluation metrics
print("Accuracy:", accuracy)
print("Confusion Matrix:\n", confusion)
print("Classification Report:\n", classification_report_str)
```

OP:

```
Accuracy: 0.4961876832844575
Accuracy: 0.4961876832844575
Confusion Matrix:
 [[123 735]
 [124 723]]
Classification Report:
              precision    recall  f1-score   support

       False       0.50      0.14      0.22       858
        True       0.50      0.85      0.63       847

    accuracy                           0.50      1705
   macro avg       0.50      0.50      0.42      1705
weighted avg       0.50      0.50      0.42      1705
```

KMEANS:

```
from sklearn.metrics import silhouette_score, davies_bouldin_score
import pandas as pd
from sklearn.cluster import KMeans

data = pd.read_csv('MSFT.csv')

features = data[['Open', 'High', 'Low', 'Adj Close', 'Volume']]

k=3
model = KMeans(n_clusters=k)

data['Cluster'] = model.fit_predict(features)

# Evaluate the K-Means model using the Silhouette Score
silhouette_avg = silhouette_score(features, data['Cluster'])
print("Silhouette Score:", silhouette_avg)
# Evaluate the K-Means model using the Davies-Bouldin Index
davies_bouldin = davies_bouldin_score(features, data['Cluster'])
print("Davies-Bouldin Index:", davies_bouldin)
```

OP:

```
/usr/local/lib/python3.10/dist-packages/sklearn/cluster/_kmeans.py:870:
FutureWarning: The default value of `n_init` will change from 10 to 'auto'
in 1.4. Set the value of `n_init` explicitly to suppress the warning
  warnings.warn(
Silhouette Score: 0.5634415591941591
Davies-Bouldin Index: 0.5877296116966805
```

KNN

```
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
data = pd.read_csv('MSFT.csv')
data['Close_Direction'] = (data['Close'] - data['Close'].shift(1)) > 0
```

```
features = data[['Open', 'High', 'Low', 'Adj Close', 'Volume']]
target = data['Close_Direction']
X_train, X_test, y_train, y_test = train_test_split(features, target, test_size=0.2,
random_state=42)
k=5
model = KNeighborsClassifier(n_neighbors=k)
# Train the model
model.fit(X_train, y_train)
# Make predictions
y_pred = model.predict(X_test)

# Evaluate the model (classification)
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy)
# Make predictions
y_pred = model.predict(X_test)
# Evaluate the k-NN model
accuracy = accuracy_score(y_test, y_pred)
confusion = confusion_matrix(y_test, y_pred)
classification_report_str = classification_report(y_test, y_pred)
# Display evaluation metrics
print("Accuracy:", accuracy)
print("Confusion Matrix:\n", confusion)
print("Classification Report:\n", classification_report_str)
```

OP:

```
Accuracy: 0.4879765395894428
Accuracy: 0.4879765395894428
Confusion Matrix:
 [[407 451]
 [422 425]]
Classification Report:
               precision    recall  f1-score   support

       False       0.49      0.47      0.48       858
        True       0.49      0.50      0.49       847

    accuracy                           0.49      1705
   macro avg       0.49      0.49      0.49      1705
weighted avg       0.49      0.49      0.49      1705
```