

Stock Price Analysis and Prediction

Submitted to

Mr. Grigoriev, Andrei

Submitted by

Shivangi Modi (20093)

Vaishnavi Patil(20133)

Kashmira Chaudhari(20158)

Index:

Introduction

I. Related Work

- i. Paper 1: A Comparative Study of Supervised Machine Learning Algorithms for Stock Market Trend Prediction**
- ii. Paper 2: Predicting Stock and Stock Price Index Movement Using Trend Deterministic Data Preparation and Machine Learning Techniques**
- iii. Paper 3: Stock Market Prediction Using Machine Learning**

II. Implementation

- i. Data Preprocessing:**
- ii. Exploratory Data Analysis (EDA):**
- iii. Model Implementation**

III. Training and Testing

IV. Fit various models

- i. Linear Regression**
- ii. Decision Tree**
- iii. Random Forest**
- iv. Grid for SVM (Support Vector Machine)**

V. Conclusion

References

Introduction:

Stock price prediction is a significant and challenging task due to the complex and dynamic nature of financial markets. The ability to accurately predict stock prices has the potential to provide substantial economic benefits, not only for investors but also for financial analysts, fund managers, and policymakers. Stock markets are influenced by a wide range of factors, including economic indicators, market trends, company performance, and global events, which adds to the unpredictability of stock price movements.

In recent years, technological advancements and the availability of large datasets have led to the adoption of machine learning techniques for stock price prediction. Unlike traditional statistical methods, machine learning models can analyze large volumes of data and capture complex patterns, trends, and relationships that are often not apparent to human analysts. These models can learn from historical data and adapt to changing market conditions, improving the accuracy of predictions.

Supervised machine learning algorithms such as Support Vector Machines (SVM), Random Forest, K-Nearest Neighbors (KNN), Naïve Bayes, and Softmax classifiers are increasingly being applied to predict stock market trends. Each of these algorithms has its strengths and weaknesses, making it essential to compare their performance under different conditions, such as dataset size and the number of technical indicators used.

This report explores the application of supervised machine learning techniques for stock price prediction. It aims to identify the most effective models for predicting market trends by analyzing their performance on various datasets. The study also examines how the accuracy of these models is influenced by the reduction of technical indicators. Through this comparative analysis, the report seeks to provide insights that can assist investors and analysts in making more informed decisions, ultimately contributing to more efficient and transparent financial markets.

I. Related Work

In this section, we review significant research studies related to stock price prediction using machine learning techniques. These studies provide insights into the performance of various models, data preparation techniques, and the challenges encountered in predicting stock market trends.

Paper 1: A Comparative Study of Supervised Machine Learning Algorithms for Stock Market Trend Prediction

This study focuses on a comparative analysis of several **supervised machine learning algorithms** for predicting stock market trends. The authors employ algorithms such as **Linear Regression, Support Vector Machines (SVM), Decision Trees, and Random Forest**. The study uses historical stock market data to train and test these models, aiming to predict the direction of stock price movements (whether prices will go up or down). The models are evaluated using metrics such as **accuracy, precision, recall, and F1-score**.

Dataset: This paper likely uses historical stock market data, such as:

- **Yahoo Finance Data:** This is a popular source for stock market data, which provides historical stock prices, including opening, closing, high, low prices, and trading volume.
- **Quandl:** Another well-known source for financial data.
- **Google Finance:** This provides similar data to Yahoo Finance but can be used to compare algorithms over different stock performance metrics.

Key Highlights:

- **Linear Regression:** Simple to implement but struggles with capturing complex relationships in the data.
- **Support Vector Machines (SVM):** Effective in handling high-dimensional data and performs well with stock trend prediction tasks.
- **Decision Trees:** Easy to interpret but prone to overfitting, especially with noisy data.

- **Random Forest:** Outperforms other models due to its ensemble nature, reducing overfitting and providing robust predictions.

Key Findings and Visualizations:

1. Comparison for Small Datasets (Figure 2):

- **Naïve Bayes** performs the best when using small datasets.
- This result is illustrated with a **bar graph** in Figure 2, showing the comparative accuracy of all models.

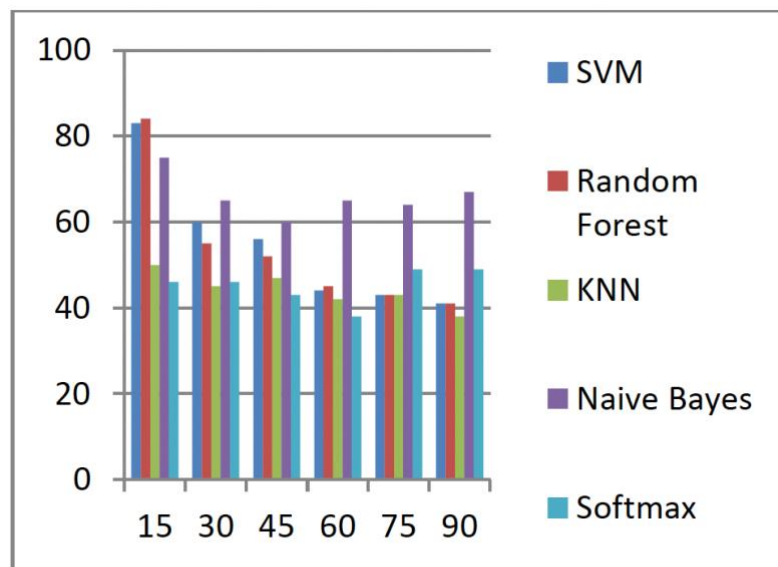


Figure 2. Comparison of algorithms for small dataset (Naive Bayes performs best).

2. Comparison for Large Datasets (Figure 3):

- **Random Forest** outperforms other algorithms when working with large datasets.
- The results are visualized in **Figure 3**, a bar graph depicting the accuracy of different algorithms on large datasets.

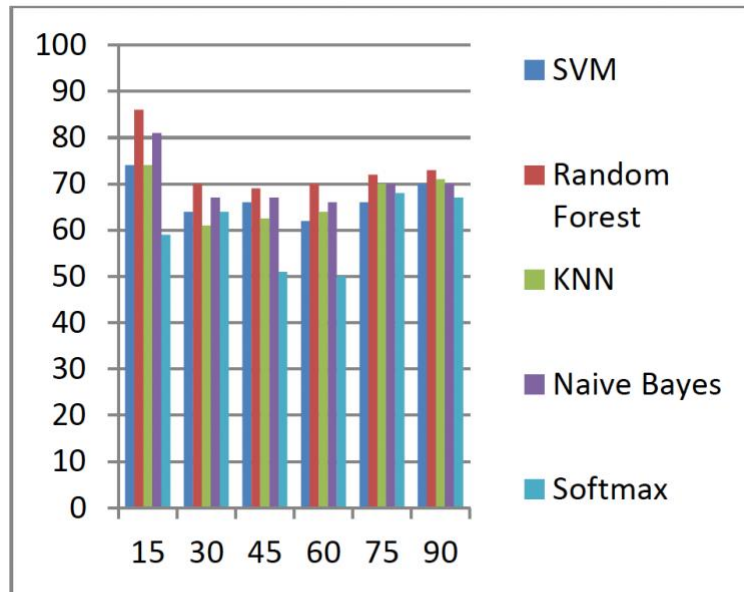


Figure 3. Comparison of algorithms for large dataset (Random forest performs best).

3. Impact of Reduced Technical Indicators:

- The authors examine the effect of removing six technical indicators (**RSI, Williams%R, MA 50, Disparity 10, RoC 2, and CCI**).
- The results show a significant **decrease in accuracy** when these indicators are removed.
 - **Figure 4** (Small Dataset): A bar graph comparing algorithm performance after technical indicators are removed.

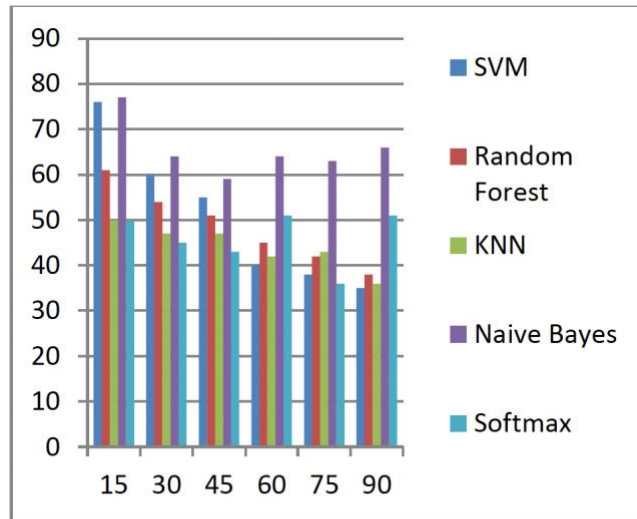


Figure 4. Comparison of algorithms for less number of technical indicators for small dataset (Compare with Figure 2).

- **Figure 5 (Large Dataset):** A bar graph comparing algorithm performance with fewer technical indicators for large datasets.

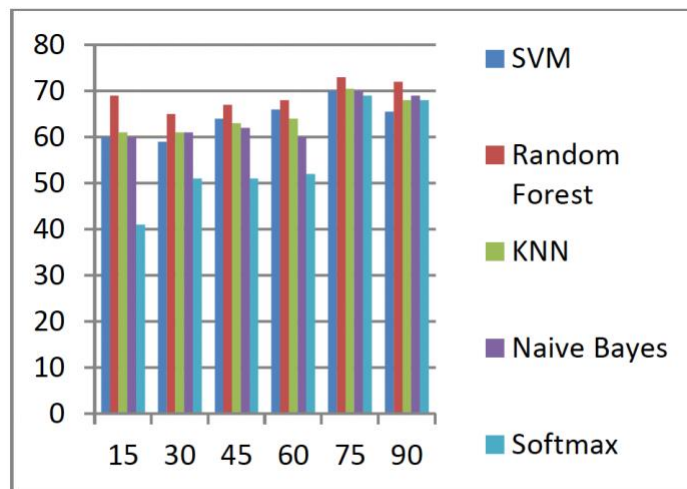


Figure 5. Comparison of algorithms for less number of technical indicators for large dataset (Compare with Figure 3)

These visualizations effectively demonstrate how dataset size and the availability of technical indicators influence the performance of different machine learning algorithms. The study highlights that while Naïve Bayes performs well on smaller

datasets, Random Forest achieves the highest accuracy on larger datasets, reinforcing the importance of selecting appropriate models based on data characteristics.

The study underscores the importance of **feature selection** and **data preprocessing** in improving model performance. The authors conclude that Random Forest and SVM deliver higher accuracy compared to simpler models, making them suitable for stock market trend prediction.

Paper 2: Predicting Stock and Stock Price Index Movement Using Trend Deterministic Data Preparation and Machine Learning Techniques

This research explores the use of **Trend Deterministic Data Preparation (TDDP)** combined with machine learning techniques for predicting stock prices and indices. The authors evaluate multiple machine learning models, including **Decision Trees, Support Vector Machines (SVM), Artificial Neural Networks (ANN), and Random Forest**. The TDDP method focuses on preparing the input data to capture trend patterns and eliminate noise, making the data more suitable for prediction tasks.

Dataset: This paper might use both stock-specific data (individual company stock prices) and stock market index data (such as the **S&P 500, Nasdaq Composite, Dow Jones Industrial Average**).

- The data could include:
 - **Historical prices** (open, close, high, low, volume)
 - **Technical indicators:** Moving averages, RSI (Relative Strength Index), MACD (Moving Average Convergence Divergence), etc.
 - **Market indices** data, such as the S&P 500, which could serve as a benchmark for stock movement predictions.

Key Highlights:

- **Trend Deterministic Data Preparation (TDDP):** This method helps improve the quality of the data by identifying and emphasizing trend patterns, which enhances model performance.
- **Decision Trees and SVM:** These models provide reliable predictions when combined with the TDDP method.
- **Artificial Neural Networks (ANN):** While effective, ANN models require more data and computational power.
- **Random Forest:** Achieves the best overall performance due to its ability to handle complex datasets and mitigate overfitting.

The study concludes that effective data preparation is critical for accurate predictions, and models like Random Forest and SVM perform exceptionally well when trend patterns are properly highlighted.

Paper 3: Stock Market Prediction Using Machine Learning

This paper investigates the application of **machine learning models** for stock price prediction, focusing on techniques such as **Linear Regression, Decision Trees, Support Vector Machines (SVM), and ensemble methods like Random Forest**. The authors use historical stock price data combined with technical indicators such as **Moving Averages (MA), Relative Strength Index (RSI), and Moving Average Convergence Divergence (MACD)** to train the models.

Dataset: This study likely uses stock data with features similar to those in the other papers, such as:

- **Historical stock price data** (open, close, high, low, volume).
- **Stock price indexes** like **S&P 500** or **Dow Jones**.
- **Market sentiment data** (such as social media sentiment, news articles, etc.) might also be considered.
- **Additional technical indicators** or **macroeconomic factors** such as interest rates, GDP, inflation, etc.

Key Highlights:

- **Linear Regression:** Provides a baseline for prediction but is limited in capturing non-linear patterns.
- **Decision Trees:** Useful for understanding decision-making paths but susceptible to overfitting with noisy financial data.
- **Support Vector Machines (SVM):** Demonstrates good performance by finding the optimal decision boundary for trend classification.
- **Random Forest:** Combines multiple decision trees to improve prediction accuracy and reduce overfitting, making it suitable for complex datasets.
- **Technical Indicators:** Including indicators like MA, RSI, and MACD helps improve the predictive power of the models by providing additional context on market conditions.

The authors conclude that **ensemble methods**, particularly Random Forest, outperform individual models in terms of accuracy and robustness. The inclusion of multiple technical indicators further enhances the models' ability to predict stock market trends effectively.

Highlights of three papers:

The reviewed papers highlight the following key points relevant to stock price prediction:

1. Model Performance:

- **Linear Regression** is simple but limited in handling complex relationships.
- **Support Vector Machines (SVM)** and **Random Forest** consistently achieve higher accuracy due to their ability to capture non-linear patterns and mitigate overfitting.
- **Decision Trees** provide interpretability but may overfit without proper tuning.

2. Data Preparation:

- **Trend Deterministic Data Preparation (TDDP)** and feature engineering play a significant role in enhancing prediction accuracy.
- Including **technical indicators** like Moving Averages, RSI, and MACD helps improve model performance by offering additional insights into market behavior.

3. Ensemble Methods:

- **Random Forest** is frequently identified as the most effective model due to its robustness and ability to handle large datasets with complex patterns.

These findings provide a solid foundation for this report's approach, which focuses on **Linear Regression, Decision Trees, Random Forest, and Grid Search SVM** for stock price prediction.

II. Implementation

In this section, we outline the implementation details of our stock price prediction system, including the dataset, preprocessing techniques, and machine learning models used.

Dataset Description

The dataset used for this study is sourced from **Kaggle** and is in **CSV format** named “**all_stocks_5k.csv**”. Key details about the dataset include:

- **Total Records:** 619,040
- **Companies Covered:** 505

The dataset contains essential information such as date, stock prices (open, close, high, low), volume, and various technical indicators. This comprehensive dataset allows for thorough model training and evaluation.

1. Data Preprocessing:

Data Cleaning

In the data preprocessing phase, handling missing values is crucial to ensure the dataset's quality and reliability. In our dataset, several columns such as open, high, and low contain missing values (null entries).

To find null entries:

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

Output:

date	0
open	11
high	8
low	8
close	0
volume	0
Name	0

dtype: int64

A. Handling Missing Values:

We have addressed these missing values by using the **forward fill (ffill) method**. Forward fill replaces each null value with the previous valid entry in the same column. This technique is appropriate for time-series data like stock prices, where it's reasonable to assume that the most recent valid value can fill a temporary gap in the data.

The following code performs forward fill on the affected columns:

```
# Fill missing numeric values with forward fill
df['open'].fillna(method='ffill', inplace=True)
df['high'].fillna(method='ffill', inplace=True)
df['low'].fillna(method='ffill', inplace=True)
```

- **Purpose:** The code is used to handle missing or NaN values in the 'open' column of the dataframe. In real-world datasets, missing values are common, especially in time-series data like stock prices, and need to be handled to avoid errors during analysis.
- **Forward Fill Method:** The `fillna(method='ffill')` function replaces missing values with the most recent non-null value in the column. The term **forward fill** means that a missing value will be filled by copying the previous valid (non-null) value in the same column. For example, if the stock price on day 5 is missing, it will be replaced by the stock price on day 4.
- **Time Series Data:** Forward filling is particularly useful in time-series data, such as stock prices, where missing data points often occur for specific time periods, but the most recent value is a reasonable estimate for the missing one.
- **Data Continuity:** This method ensures the continuity of data. For stock prices, this approach avoids introducing unrealistic values, as the assumption is that stock prices often remain constant or change gradually, especially for short time gaps.
- **inplace=True Argument:** The `inplace=True` argument ensures that the modification is applied directly to the dataframe. Without this argument, a new dataframe with the missing values filled would be returned, leaving the original dataframe unchanged. This approach is more memory efficient and simplifies code when you want to update the original dataframe.

□ **Handling Missing Values Efficiently:** Using forward fill is an efficient way to deal with missing data without losing much information, especially when the missing values occur in rows where previous data can be reasonably assumed to be a good estimate for the missing value.

To verify that all missing values have been successfully handled, we use the following code:

```
# Verify no missing values remain
print(df.isnull().sum())
```

Output:

```
date      0
open      0
high      0
low       0
close     0
volume    0
Name      0
dtype: int64
```

Explanation:

After filling the missing values using the forward fill method (`fillna()`), it's important to check if there are any remaining missing or null values in the dataset. The code `df.isnull().sum()` helps achieve this by performing two steps:

1. **`df.isnull()`:** This method returns a DataFrame of the same shape, where each entry is either True (if the value is missing) or False (if the value is not missing).
2. **`sum()`:** The `sum()` function is applied to each column, summing the True values (which are treated as 1) and the False values (treated as 0). The result is the total count of missing values for each column.

B. Date Conversion for Time-Series Analysis

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

1. **Correct Data Representation:** The 'date' column may initially be in string format, which doesn't allow for proper date-based operations. Converting it to a datetime format ensures that the data is recognized and treated as dates, allowing for more accurate manipulation and analysis.
2. **Time-Based Operations:** With the 'date' column in datetime format, we can perform time-based operations such as sorting, filtering, or resampling (e.g., extracting the year, month, or day). This is particularly useful when analyzing trends over time in stock market data.
3. **Handling Missing Values:** Date columns in string format can complicate the identification of missing or erroneous values. By converting the column to datetime, handling missing or invalid dates becomes easier.
4. **Efficient Plotting:** Visualizations, such as time series plots, require a proper datetime format to correctly plot the data on the x-axis. Without this conversion, plotting might fail or produce inaccurate results.
5. **Comparison with Other Date Values:** When working with multiple datasets or comparing stock prices across different time periods, it's necessary to have dates in the same format to make accurate comparisons and aggregations.

By converting the 'date' column to datetime format, we enable these essential operations, improving the overall analysis and predictive modeling process.

Now that the null values have been successfully handled, we will proceed with the exploration of the data to gain insights and prepare it for modeling.

- Data Inspection

□ **Sorting Data by Date:** The code begins by sorting the data based on the 'date' column using `sort_values()`. This is important for ensuring that the dataset is in chronological order, making it easier to analyze time-dependent trends, such as stock price changes.

```
# Sort data by Date
df = df.sort_values(by='date')
```

□ **Listing Unique Company Names:** The `unique()` method is used on the 'Name' column to retrieve a list of unique company names present in the dataset. This helps identify all the distinct companies in the stock market dataset.

```
# list all Name  
df['Name'].unique()
```

□ **Counting Occurrences of Each Company:** The `value_counts()` method is applied to the 'Name' column to count how many times each company appears in the dataset. This helps in understanding the distribution of records across different companies.

```
# count all Name  
df['Name'].value_counts()
```

□ **Data Overview:** These steps provide an initial overview of the dataset by organizing the data and giving insight into how many records exist for each company and whether there are any anomalies or patterns in the data.

```
# count total Name  
df['Name'].nunique()
```

□ **Preparation for Further Analysis:** This code helps in preparing the data for further exploration and analysis, as it ensures the data is properly ordered and gives insights into the distribution of records across different entities.

2. Exploratory Data Analysis (EDA):

1. Visualization of Closing Prices for a Specific Stock (AAPL)


```
# Plot closing prices of a specific stock, e.g., 'AAPL'
apple_data = df[df['Name'] == 'AAPL']
plt.figure(figsize=(10, 5))
plt.plot(apple_data['date'], apple_data['close'], label='Close Price')
plt.title('AAPL Stock Price Over Time')
plt.xlabel('Date')
plt.ylabel('Close Price')
plt.legend()
plt.show()
```

This code generates a line plot of the closing prices for the stock 'AAPL' (Apple Inc.) over time. The key steps and their significance are as follows:

1. Filter Data for 'AAPL':

The dataset is filtered to include only records where the 'Name' column is 'AAPL'. This isolates the data specifically for Apple's stock.

2. Set Figure Size:

The plot dimensions are defined using `plt.figure(figsize=(10, 5))` to ensure the visualization is clear and appropriately sized.

3. Plot Closing Prices:

The `plt.plot()` function is used to create a line plot where the x-axis represents dates and the y-axis represents the closing prices for 'AAPL'. This helps in visualizing how the closing price changes over time.

4. Add Titles and Labels:

- **Title:** Provides context to the plot with `plt.title('AAPL Stock Price Over Time')`.
- **X-axis Label:** Marks the x-axis with `plt.xlabel('Date')`.
- **Y-axis Label:** Marks the y-axis with `plt.ylabel('Close Price')`.

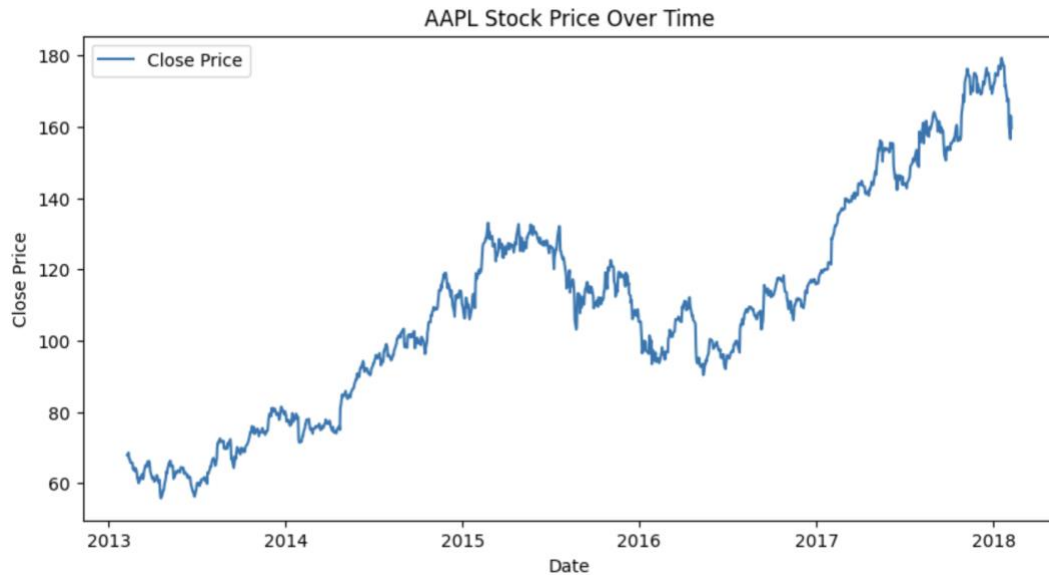
5. Add Legend:

The `plt.legend()` function includes a legend to clarify the line representing the closing price.

6. Display the Plot:

Finally, `plt.show()` renders the plot, allowing us to visually analyze trends and patterns in Apple's stock prices over the specified period.

Output:



2. Feature Engineering: Creating Moving Averages

In stock price prediction, **Feature Engineering** involves creating new features to enhance the performance of predictive models. In this step, we introduce **moving averages** for Apple's stock data, which help capture trends and smooth out price fluctuations. The code creates three new features: MA10, MA50, and MA200, representing moving averages over different time periods.

```
# Feature Engineering
# Create new features: Moving Averages
apple_data['MA10'] =
apple_data['close'].rolling(window=10).mean()
apple_data['MA50'] =
apple_data['close'].rolling(window=50).mean()
apple_data['MA200'] =
apple_data['close'].rolling(window=200).mean()
```

Explanation:

1. Create a 10-Day Moving Average (MA10):

- This feature calculates the average closing price over the past 10 days.
- It captures short-term trends and helps identify recent price momentum.

2. Create a 50-Day Moving Average (MA50):

- This feature calculates the average closing price over the past 50 days.
- It smooths out medium-term fluctuations and helps identify intermediate trends.

3. Create a 200-Day Moving Average (MA200):

- This feature calculates the average closing price over the past 200 days.
- It is useful for identifying long-term trends and overall market direction.

The significance of this step can be understood for the following reasons:

1. Trend Identification:

Moving averages help smooth out short-term volatility and highlight the underlying trends in stock prices.

2. Signal Generation:

These features are often used in technical analysis to generate buy or sell signals. For example, when the 10-day moving average crosses above the 50-day moving average, it might signal a buying opportunity.

3. Improved Model Accuracy:

Including moving averages as features can improve the accuracy of machine learning models by providing more context about recent price movements.

4. Data Smoothing:

They reduce the impact of random fluctuations, making it easier to identify consistent patterns in the data.

5. Feature Diversity:

By adding multiple moving averages (10-day, 50-day, and 200-day), models can capture trends over different time horizons, enriching the feature set for predictive analysis.

3. Plotting Moving Averages for Report

The code visualizes the closing prices and key moving averages for the stock 'AAPL' over time. The plot includes:

```
# Plot Moving Averages
plt.figure(figsize=(10, 5))
plt.plot(apple_data['date'], apple_data['close'],
label='Close Price')
```

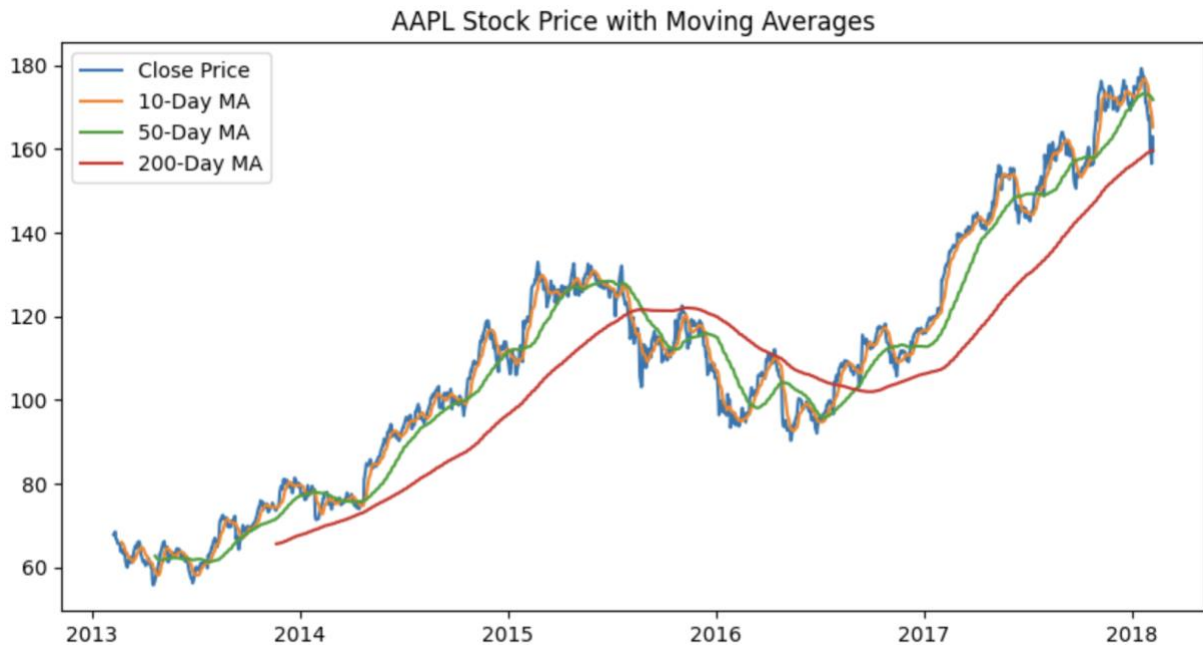
```
plt.plot(apple_data['date'], apple_data['MA10'],  
label='10-Day MA')  
plt.plot(apple_data['date'], apple_data['MA50'],  
label='50-Day MA')  
plt.plot(apple_data['date'], apple_data['MA200'],  
label='200-Day MA')  
plt.title('AAPL Stock Price with Moving Averages')  
plt.legend()  
plt.show()
```

1. **Closing Prices:** The actual daily closing prices of AAPL, showing the stock's performance over the given period.
2. **10-Day Moving Average (MA10):** A short-term moving average that smooths out daily fluctuations, providing insights into short-term trends.
3. **50-Day Moving Average (MA50):** A mid-term moving average that helps identify intermediate trends, often used by traders to gauge momentum.
4. **200-Day Moving Average (MA200):** A long-term moving average that captures the overall market trend, widely used to assess the stock's long-term direction.

The significance of this step can be understood for the following reasons:

- The moving averages help smooth out volatility and make it easier to identify trends and potential buy or sell signals.
- **Crossovers** of shorter-term moving averages with longer-term ones (e.g., MA10 crossing MA50 or MA200) can signal trend changes.
- This visualization helps in understanding how the stock's short-term, medium-term, and long-term movements interact, aiding in predictive analysis and decision-making.

By plotting these moving averages alongside the closing prices, the chart provides a comprehensive view of AAPL's price trends and potential market signals.



3. Model Implementation

1. Model Building: Predicting Close Price

In this step, the data is prepared for building a predictive model. The **features** are the independent variables used to make predictions, while the **target** is the dependent variable we want to predict.

```
# Model Building: Predicting Close Price
# Prepare data for modeling
features = apple_data[['open', 'high', 'low',
                       'volume']]
target = apple_data['close']
```

Specifically:

1. Features Selected:

The features chosen include:

- **open**: The opening price of the stock for the day.
- **high**: The highest price reached during the day.
- **low**: The lowest price reached during the day.
- **volume**: The number of shares traded during the day.

2. Target Variable:

The target variable, **close**, represents the closing price of the stock, which is the price at the end of the trading day.

3. Purpose:

These features provide essential market information for predicting the closing price, helping to train the model to recognize patterns and relationships in the stock data.

4. Data Structure:

The dataset is organized into feature and target arrays, making it suitable for use in various machine learning models such as Linear Regression, Decision Trees, and others.

This step is crucial for ensuring the model has the necessary data to learn and make accurate predictions.

2. Data Standardization and Entropy Calculation

1. Standardization of Data:

In this step, the features are standardized using **StandardScaler()**, which transforms the data to have a mean of 0 and a standard deviation of 1. This is important for machine learning models, especially those that are sensitive to the scale of input data, such as Linear Regression or SVM. Standardization ensures that all features contribute equally to the model's performance and prevents features with larger ranges from dominating the model's learning process.

```
# Standardize data
scaler = StandardScaler()
X_scaled = scaler.fit_transform(features)
```

- **X_scaled**: This is the transformed version of the features where each feature has been standardized using the `fit_transform()` method.

2. Entropy Calculation:

Entropy is a measure of the uncertainty or unpredictability of a dataset. In the context of stock market data, it indicates the level of disorder or randomness in the values of each feature.

```
# Compute entropy for each feature
from scipy.stats import entropy
print("Entropy for each feature:")
for col in features.columns:
    ent =
entropy(pd.value_counts(features[col].values,
normalize=True), base=2)
    print(f"{col}: {ent:.4f}")
```

- **Purpose:** By calculating entropy for each feature, we can gain insights into the variability and distribution of values in the dataset. Higher entropy values suggest greater variability, whereas lower values suggest more predictable or uniform features.
- **Process:** For each feature (open, high, low, volume), the entropy is computed by applying the `entropy()` function from the **scipy.stats** library. The function calculates the entropy based on the normalized frequency distribution of the feature's values, with the base of the logarithm set to 2 to measure the entropy in bits.
- **Output:** The entropy values for each feature are printed, indicating the level of uncertainty in their values:
 - **open:** 10.1825
 - **high:** 10.1940
 - **low:** 10.1904
 - **volume:** 10.2981

These values suggest that all features have relatively high entropy, indicating that they have a large degree of variability and are not overly predictable, which is common in stock market data. This variability can be useful for predicting the closing price, as the model can capture these fluctuations to make accurate predictions.

3. Mutual Information Scores Calculation

1. Purpose of Mutual Information:

Mutual Information (MI) measures the amount of information shared between two variables. In the context of predictive modeling, it quantifies the relationship between the features and the target variable (in this case, the stock's closing price). A higher MI score indicates a stronger relationship between a feature and the target variable, while a lower score suggests a weaker relationship.

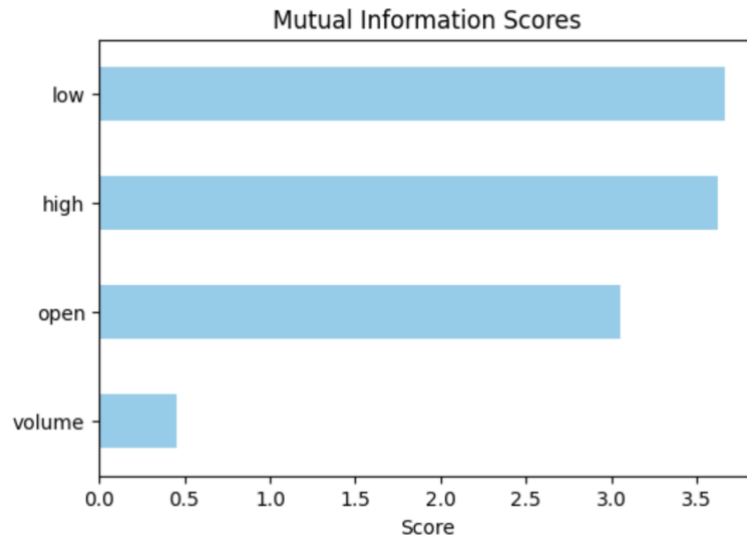
2. Computing Mutual Information:

The `mutual_info_regression()` function from `sklearn.feature_selection` is used to compute the mutual information scores between each feature in the dataset and the target variable (close). The data has already been standardized, ensuring that the MI scores are not influenced by the scale of the features.

```
# Compute mutual information scores
mutual_info = mutual_info_regression(X_scaled, target)
mi_scores = pd.Series(mutual_info,
index=features.columns)
print("\nMutual Information Scores:")
print(mi_scores.sort_values(ascending=False))
```

3. Interpreting the Output:

The mutual information scores for each feature are stored in a pandas Series and printed in descending order. This allows us to see which features are most informative for predicting the target variable (closing price):



- **low:** 3.6618
The low price has the highest mutual information with the target, indicating it shares the most relevant information for predicting the closing price.
- **high:** 3.6243
The high price also shows a strong relationship with the target, almost as much as low.
- **open:** 3.0474
The open price, while still informative, has a slightly weaker relationship with the closing price compared to low and high.
- **volume:** 0.4563
The volume feature has the lowest mutual information score, suggesting it has the weakest relationship with the target variable and may not be as valuable in predicting the closing price.

4. **Significance:**

By identifying the features with the highest mutual information, we can focus on the most influential variables when building predictive models. In this case, low and high prices are likely to be the most important predictors of the closing price, while volume may be less useful in isolation. This insight helps in feature selection for model improvement.

4. **Plotting Mutual Information Scores:**

1. **Purpose of the Plot:**

The plot visually represents the mutual information scores of the features (open, high, low, volume) in relation to the target variable (closing price). It

helps assess which features provide the most valuable information for predicting the closing price.

2. **Understanding the Output:**

The mutual information scores are calculated and displayed as follows:

- **low:** 3.6633 (highest score)
- **high:** 3.6254
- **open:** 3.0469
- **volume:** 0.4559 (lowest score)

These scores quantify the degree of dependence between each feature and the target. Higher scores indicate stronger relationships.

3. **Visualization:**

- The horizontal bar chart shows the mutual information scores sorted in ascending order.
- Each feature's score is represented by a light blue (skyblue) bar.
- The x-axis shows the scores, and the y-axis lists the features.

4. **Key Insights:**

- **low and high** prices have the highest mutual information scores, suggesting they are the most informative features for predicting the closing price.
- **open** price also contributes significantly to the target.
- **volume** has a much lower score, indicating it provides less predictive value compared to the other features.

5. **Significance for Feature Selection:**

This analysis helps identify which features to prioritize when building predictive models. Features with higher mutual information scores (such as low and high) should be considered essential, while features with lower scores (like volume) might be considered for exclusion or further evaluation.

5. Correlation matrix:

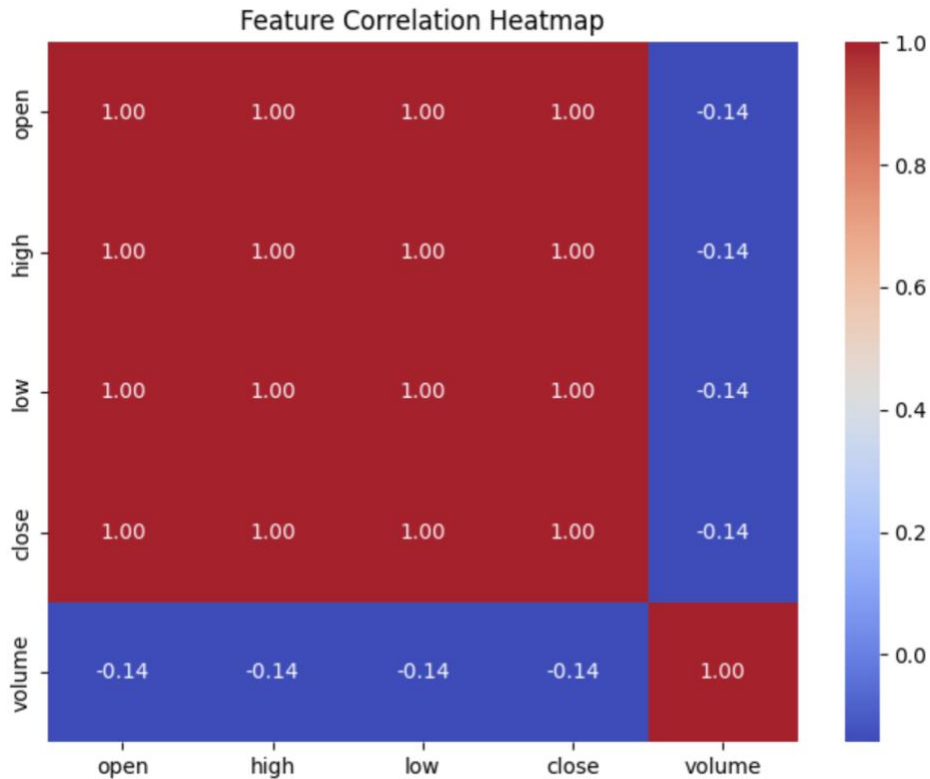
```
# Compute correlation matrix
correlation_matrix = df[['open', 'high', 'low', 'close',
                        'volume']].corr()
```

The correlation matrix is a table that displays the correlation coefficients between multiple numerical features in the dataset, specifically open, high, low, close, and volume prices. Each value in the matrix quantifies the strength and direction of the linear relationship between two features, ranging from -1 (perfect negative correlation) to +1 (perfect positive correlation). A correlation coefficient near 0 suggests no linear relationship. By examining the correlation matrix, we can identify which features are strongly correlated with each other, which is useful for understanding data relationships and avoiding multicollinearity in predictive models. For example, high correlations between open, high, low, and close prices are expected since these values are interdependent. Conversely, the volume feature might show lower correlation with price-related features, providing unique information. This analysis informs feature selection and preprocessing steps for model building.

6. Correlation Heatmap:

Code:

```
# Plot correlation heatmap
plt.figure(figsize=(8, 6))
sns.heatmap(correlation_matrix, annot=True,
            cmap='coolwarm', fmt=".2f")
plt.title('Feature Correlation Heatmap')
plt.show()
```



1. **Visualization of Relationships:** The correlation heatmap visually displays the correlation between numerical features (open, high, low, close, and volume) using color intensity.
2. **Strong Positive Correlations:** The price-related features (open, high, low, close) have very high positive correlations (close to 1). This indicates that these features tend to move together and exhibit similar trends.
3. **Weak Negative Correlation:** The volume feature shows a weak negative correlation with the price-related features (values around -0.14). This implies that trading volume does not strongly influence price movements.
4. **Multicollinearity Insight:** The high correlation between open, high, low, and close suggests potential multicollinearity, which may affect some models. Feature reduction techniques may be considered to mitigate this.
5. **Modeling Consideration:** Understanding these correlations helps in selecting the right features and avoiding redundant information, which can improve model performance and accuracy.

7. Split the dataset into training and testing sets.

```
# Split 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)
```

To prepare the data for model training and evaluation, we split the dataset into training and testing sets using an 80-20 ratio. Specifically, 80% of the data is allocated to the training set (X_train and y_train), which is used to train the predictive model, while 20% of the data is allocated to the testing set (X_test and y_test) for evaluating the model's performance. The train_test_split function from sklearn.model_selection is used for this purpose, with the parameter random_state=42 ensuring that the split is reproducible. This step is essential to ensure that the model is trained on one subset of data and validated on an unseen subset, thereby preventing overfitting and providing a realistic measure of how well the model will perform on new data.

8. Standardize the data:

```
# Standardize the data  
scaler = StandardScaler()  
X_train_scaled = scaler.fit_transform(X_train)  
X_test_scaled = scaler.transform(X_test)
```

The data is standardized using StandardScaler to scale the features. The training data is fitted and transformed, while the testing data is only transformed using the scaler. This ensures that the features have a mean of 0 and a standard deviation of 1, which helps improve the performance of machine learning models.

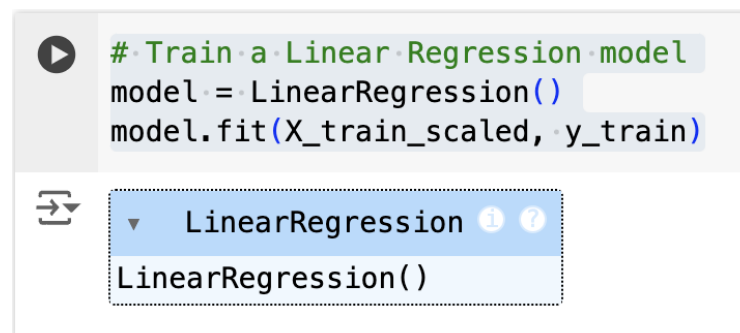
III. Training and Testing

Training a model is a crucial step in machine learning where the model learns patterns and relationships in the data by adjusting its internal parameters. In this case, we are using a Linear Regression model, which seeks to establish a linear relationship between the input features (such as 'open', 'high', 'low', 'volume') and the target variable ('close').

By using the `fit()` method, the model is trained on the scaled training data (`X_train_scaled`), which allows it to learn the coefficients that minimize the error between predicted and actual target values (`y_train`). During training, the model fine-tunes these coefficients to make accurate predictions on new, unseen data.

In the context of stock price prediction, training helps the model understand how past market conditions (i.e., the historical open, high, low, and volume prices) influence the closing price. The more accurate the training, the better the model can predict future closing prices based on similar conditions.

```
# Train a Linear Regression model
model = LinearRegression()
model.fit(X_train_scaled, y_train)
```



"Training the Linear Regression model allows it to learn the relationship between the input features and the target variable (closing price). By adjusting the model's coefficients using the training data, it can accurately predict future stock prices based on historical patterns."

1. Make Predictions on Test Data:

```
# Predict on test data
y_pred = model.predict(X_test_scaled)
```

- **Prediction on Test Data:** After training the model, it is evaluated on unseen data to assess its performance. This is done by predicting the target variable (closing price) for the test set (`X_test_scaled`), which contains features that the model has not encountered during training.
- **Using the `predict()` Method:** The `model.predict()` method is used to generate predictions based on the test data. The model applies the learned coefficients to the scaled test features (`X_test_scaled`) to estimate the closing prices.
- **Purpose:** This step helps to verify how well the model generalizes to new data, providing insight into its ability to predict stock prices in real-world scenarios.
- **Outcome:** The result is a set of predicted values (`y_pred`) that can be compared to the actual closing prices in the test set (`y_test`) to evaluate the model's accuracy.

2. Evaluate the model:

```
# Evaluate the model
mse = mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)
print(f"Mean Squared Error: {mse}")
print(f"R2 Score: {r2}")
```

Output:

Mean Squared Error: 0.3225900805372306

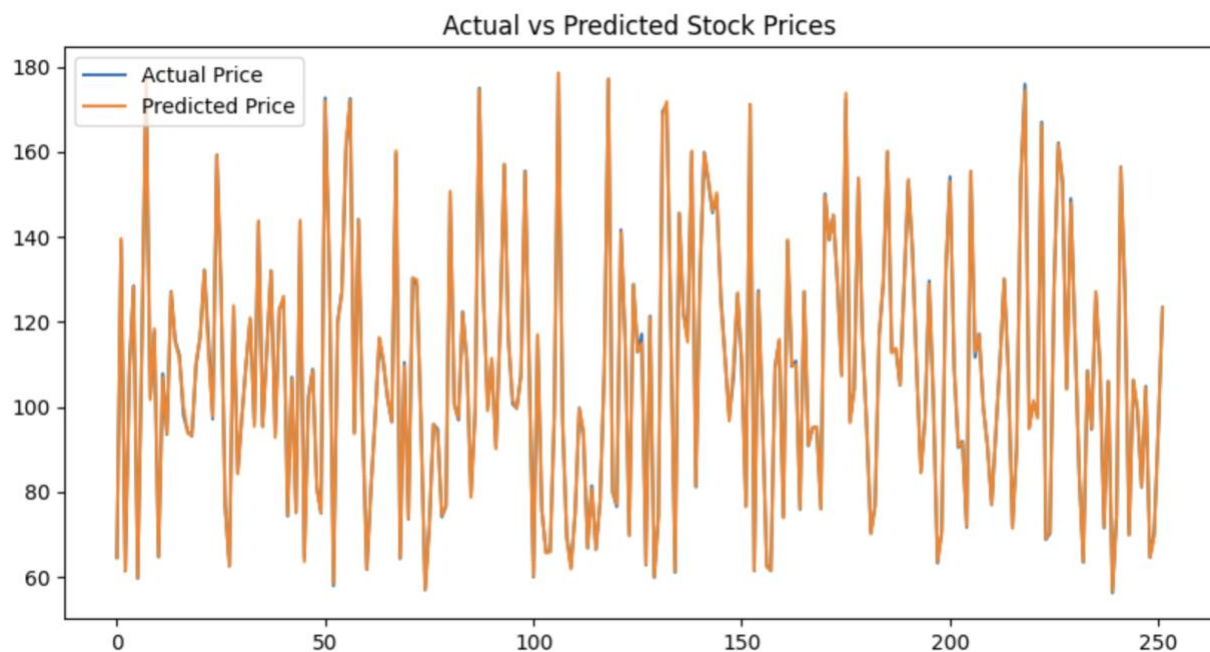
R2 Score: 0.9996487845726062

- **Evaluating Model Performance:** After generating predictions on the test data, it is essential to evaluate the performance of the model using appropriate metrics. This helps to assess how well the model has learned and whether it generalizes well to unseen data.
- **Mean Squared Error (MSE):** The Mean Squared Error is calculated using `mean_squared_error(y_test, y_pred)`. This metric measures the average squared difference between the actual (`y_test`) and predicted (`y_pred`) values. A lower MSE indicates better model performance. In this case, the MSE is 0.3226, indicating a relatively small error between the actual and predicted values.
- **R-squared (R2) Score:** The R2 score is calculated using `r2_score(y_test, y_pred)`. This metric indicates how well the model explains the variance in the target variable. An R2 score closer to 1 means the model fits the data well. In this case, the R2 score is 0.9996, which shows that the model explains almost 99.96% of the variance in the target variable (closing price), suggesting excellent predictive accuracy.

- **Conclusion:** The low MSE and high R2 score suggest that the model performs exceptionally well, with predictions being very close to the actual values.

3. Visualize Actual Prices vs Predicted Prices:

```
# Plot actual vs predicted prices
plt.figure(figsize=(10, 5))
plt.plot(y_test.values, label='Actual Price')
plt.plot(y_pred, label='Predicted Price')
plt.title('Actual vs Predicted Stock Prices')
plt.legend()
plt.show()
```



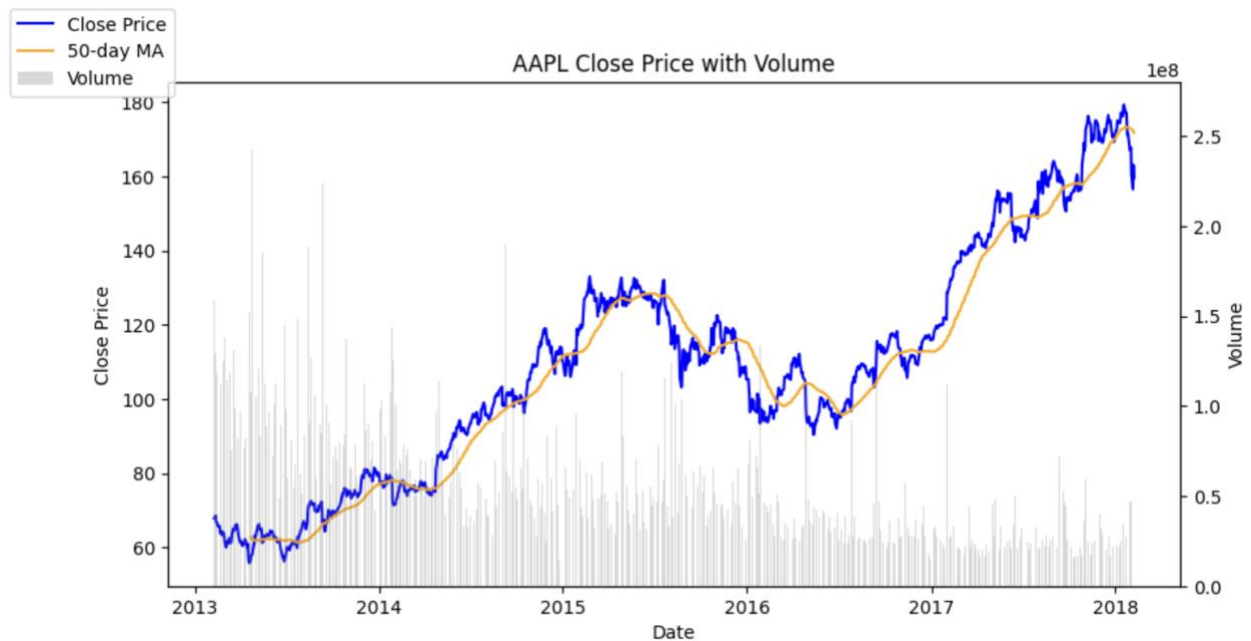
- **Visual Comparison:** The plot compares the actual stock prices (from the test set) with the predicted stock prices (from the model).
- **Plot Creation:** The `y_test.values` represents the actual stock prices, while `y_pred` contains the predicted prices from the model.
- **Labeling:** The plot is labeled for both the 'Actual Price' and 'Predicted Price' for clear visualization.
- **Interpretation:** This visualization helps assess the accuracy of the model by showing how close the predicted values are to the actual values.
- **Visualization:** The `plt.figure(figsize=(10, 5))` ensures that the plot is large enough to analyze, while `plt.legend()` adds a legend for clarity.

4. Integrates price trends with trading volume.

```
# Combines price trends with trading volume.
fig, ax1 = plt.subplots(figsize=(10, 5))
ax2 = ax1.twinx()

ax1.plot(apple_data['date'], apple_data['close'], label='Close
Price', color='blue')
ax1.plot(apple_data['date'],
apple_data['close'].rolling(window=50).mean(), label='50-day MA',
color='orange')
ax2.bar(apple_data['date'], apple_data['volume'], label='Volume',
alpha=0.3, color='grey')

ax1.set_title('AAPL Close Price with Volume')
ax1.set_xlabel('Date')
ax1.set_ylabel('Close Price')
ax2.set_ylabel('Volume')
fig.legend(loc='upper left')
plt.show()
```



- **Visualizing Price and Volume:** The code combines the visualization of Apple's closing price and trading volume on the same plot. It uses two y-axes to separately display price and volume, ensuring clarity.

- **Close Price and Moving Average:** The first y-axis (on the left) is used to plot the closing prices of AAPL, along with a 50-day moving average to smooth out short-term fluctuations and highlight longer-term trends.
- **Volume Representation:** The second y-axis (on the right) is used to display trading volume as bars, which helps in analyzing the relationship between price movements and trading activity.
- **Enhanced Clarity:** By using a twin-axis plot, the graph effectively shows both price trends and trading volume over time, providing a clearer understanding of how the volume relates to price changes.
- **Visualization Customization:** The code includes labels for both axes, a title for the plot, and a legend to distinguish between the different data series, making the plot easy to interpret.

5. Taking the Entire Dataset into Account:

```
df['date'] = pd.to_datetime(df['date'])
df = df.sort_values(by=['Name', 'date'])

# Create additional features
df['Daily Return'] =
df.groupby('Name')['close'].pct_change()
df['Average Volume'] =
df.groupby('Name')['volume'].transform('mean')

# Feature Engineering: Add new features
df['price_range'] = df['high'] - df['low']
df['average_price'] = (df['open'] + df['close']) / 2
```

- **Convert 'date' to datetime format:** The `df['date'] = pd.to_datetime(df['date'])` line converts the 'date' column into a datetime format, enabling easier handling of date-based operations like sorting, filtering, and time series analysis.
- **Sort the dataset by 'Name' and 'date':** The `df = df.sort_values(by=['Name', 'date'])` line sorts the DataFrame by the 'Name' (stock name) and 'date' columns. Sorting ensures that the data is organized by stock and in chronological order, which is important for time series analysis and feature engineering.
- **Create Daily Return feature:** The `df['Daily Return'] = df.groupby('Name')['close'].pct_change()` calculates the daily percentage change in the closing price for each stock. This is essential for analyzing stock performance relative to previous days and understanding stock price volatility.
- **Create Average Volume feature:** The `df['Average Volume'] = df.groupby('Name')['volume'].transform('mean')` computes the average trading

volume for each stock. This can help in analyzing trends and patterns in the stock's trading activity over time.

- **Create new price-related features:**
 - **Price Range:** `df['price_range'] = df['high'] - df['low']` calculates the daily price range (difference between the highest and lowest price), which can indicate market volatility.
 - **Average Price:** `df['average_price'] = (df['open'] + df['close']) / 2` computes the average price between the opening and closing prices, providing a smoother view of the stock's trading price for the day.

These feature engineering steps help to enrich the dataset by adding new information that could be valuable for further analysis or predictive modeling.

```
from sklearn.ensemble import RandomForestRegressor

# Select features and target
features = ['open', 'high', 'low', 'volume', 'Daily Return', 'Average Volume', 'price_range', 'average_price']
X = df[features]
y = df['close']

# Encode company symbols if needed
X['Name'] = pd.factorize(df['Name'])[0]
```

- **Importing RandomForestRegressor:**
The `from sklearn.ensemble import RandomForestRegressor` line imports the `RandomForestRegressor` model from the `sklearn` library. This model is a powerful ensemble method used for regression tasks, leveraging multiple decision trees to make predictions. It is well-suited for capturing complex patterns in data.
- **Selecting features and target:**
 - `features = ['open', 'high', 'low', 'volume', 'Daily Return', 'Average Volume', 'price_range', 'average_price']`: This line defines the features (independent variables) used to predict the target variable (dependent variable).
 - `X = df[features]`: This selects the specified feature columns from the DataFrame `df` and assigns them to `X`, the input matrix for the model.

- `y = df['close']`: This sets the target variable `y` as the 'close' column of the DataFrame, which is the variable we are trying to predict.
- **Encoding company symbols:**
 - `X['Name'] = pd.factorize(df['Name'])[0]`: This encodes the 'Name' column, which contains company symbols (e.g., 'AAPL' for Apple), into numeric values. The `pd.factorize()` function assigns a unique integer to each company symbol, allowing categorical data to be used in machine learning models. The `[0]` accesses the encoded values, which replace the original 'Name' column in `X`.

By preparing these features and encoding the company symbols, the data becomes suitable for training the `RandomForestRegressor` model to predict stock prices.

```
from sklearn.impute import SimpleImputer

# Handle missing values: Fill NaN with the mean of each column
imputer = SimpleImputer(strategy='mean')
X = pd.DataFrame(imputer.fit_transform(X),
                 columns=X.columns)
```

- **Importing SimpleImputer:**
The line `from sklearn.impute import SimpleImputer` imports the `SimpleImputer` class from the `sklearn.impute` module. This tool is used to handle missing values (NaN) in the dataset by filling them with specified strategies.
- **Handling Missing Values:**
 - `imputer = SimpleImputer(strategy='mean')`: This line creates an instance of the `SimpleImputer` with the `strategy='mean'`. The strategy specifies how missing values will be handled. In this case, missing values will be replaced with the mean of the respective columns.
 - `X = pd.DataFrame(imputer.fit_transform(X), columns=X.columns)`: The `fit_transform()` method is applied to the feature set `x` to impute the missing values. It calculates the mean of each column and fills any missing (NaN) values with that mean. The result is converted back into a DataFrame with the original column names (`X.columns`).

By using this approach, missing values are handled in a straightforward manner, ensuring that the dataset is complete and can be used effectively for model training. This is important because many machine learning algorithms, including the `RandomForestRegressor`, cannot handle missing values directly.

```
# Check for any remaining missing values
print("Remaining NaN values:", X.isnull().sum().sum())
```

- **Checking for Remaining Missing Values:**
The line `print("Remaining NaN values:", X.isnull().sum().sum())` is used to verify if there are any missing values left in the feature set X after applying the imputation process.
- **isnull() Function:**
The `isnull()` method returns a DataFrame of the same shape as X, with True values indicating the presence of NaN values and False for non-NaN values.
- **sum() Method:**
The first `sum()` method calculates the sum of True values (NaN occurrences) for each column. The second `sum()` method aggregates these counts across all columns, giving the total number of missing values.
- **Result:**
In this case, the result is 0, meaning there are no remaining NaN values in the feature set X, indicating that the missing values were successfully handled by the imputation process.

This step ensures that the dataset is free from missing values, which is a critical prerequisite for building a reliable machine learning model.

```
# Proceed with train-test split
X_train, X_test, y_train, y_test = train_test_split(X,
y, test_size=0.3, random_state=42)
```

- **Train-Test Split:**
The line `X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)` splits the dataset into training and testing subsets. The feature set X (which contains the independent variables) and the target variable y (which contains the dependent variable, in this case, the 'close' price) are separated into training and testing sets.
- **train_test_split() Function:**
The `train_test_split()` function from the `sklearn.model_selection` module is used to randomly partition the data.
 - `test_size=0.3` indicates that 30% of the data is allocated to the testing set, while 70% is used for training the model.

- `random_state=42` ensures reproducibility of the results. By setting the `random_state`, the split will be consistent across different runs of the code.

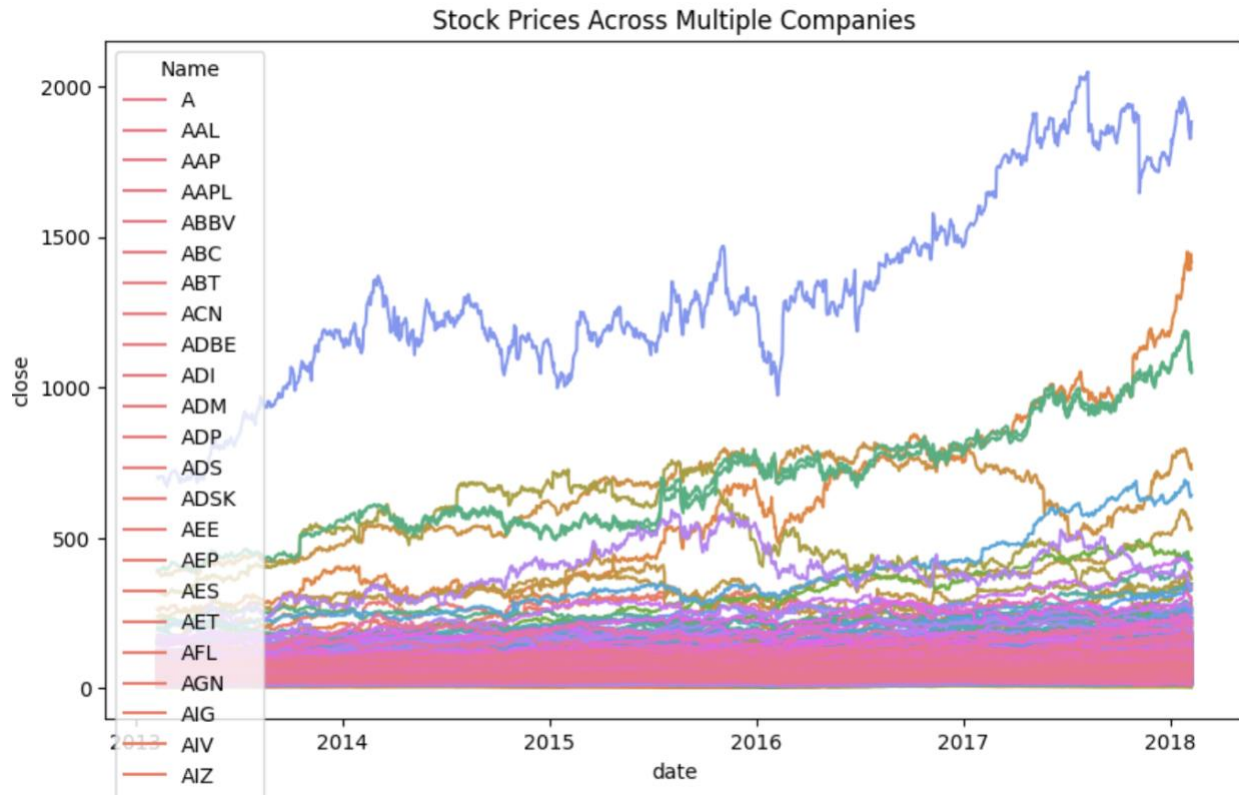
- **Resulting Variables:**

- `X_train`: The features used for training the model.
- `X_test`: The features used for testing the model.
- `y_train`: The target values (close prices) for training the model.
- `y_test`: The target values (close prices) for testing the model.

This step is crucial for evaluating the performance of the machine learning model by using separate data for training and testing, preventing overfitting and ensuring generalization to new, unseen data.

6. Stock Prices of Various Companies:

```
import seaborn as sns
plt.figure(figsize=(10, 6))
sns.lineplot(data=df, x='date', y='close', hue='Name')
plt.title('Stock Prices Across Multiple Companies')
plt.show()
```



The code provided generates a line plot to visualize the stock prices of multiple companies over time. Here's a breakdown of what it does:

- **Imports the necessary library:** seaborn is imported to create the line plot.
- **Sets the figure size:** `plt.figure(figsize=(10, 6))` specifies the size of the plot (10x6 inches).
- **Creates the line plot:**
 - `sns.lineplot(data=df, x='date', y='close', hue='Name')` plots the closing prices ('close') for each company ('Name') over time ('date'). The `hue='Name'` parameter differentiates each company's data by color.
- **Sets the plot title:** `plt.title('Stock Prices Across Multiple Companies')` adds a title to the plot.
- **Displays the plot:** `plt.show()` renders the plot.

This plot helps to visually compare the stock price trends of different companies over time.

7. Annual High and Low Analysis by Company

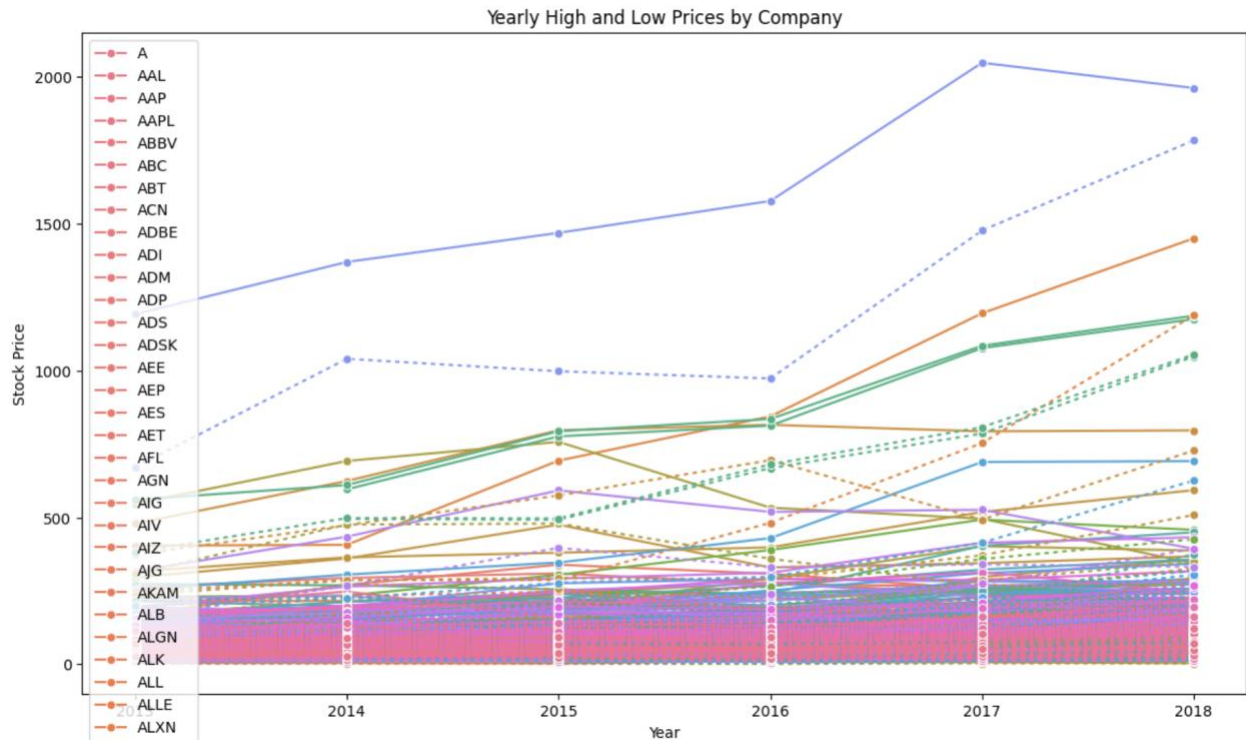
```
# Yearly High and Low Analysis per Company
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

# Convert Date column to datetime and extract year
df['Year'] = df['date'].dt.year

# Calculate yearly high and low for each company
yearly_stats = df.groupby(['Name',
'Year'])['close'].agg(['max', 'min']).reset_index()

# Plot high and low prices per year
plt.figure(figsize=(14, 8))
sns.lineplot(data=yearly_stats, x='Year', y='max',
hue='Name', marker='o')
sns.lineplot(data=yearly_stats, x='Year', y='min',
hue='Name', style=True, dashes=[(2, 2)], marker='o')

plt.title('Yearly High and Low Prices by Company')
plt.xlabel('Year')
plt.ylabel('Stock Price')
plt.legend(loc='upper left')
plt.show()
```

- **Date Conversion and Year Extraction:**

- The date column is converted to datetime format, and a new column Year is created by extracting the year from the date column. This helps in grouping the data by year.

- **Calculating Yearly High and Low:**

- The `groupby()` function is used to group the data by Name (company) and Year. Then, the `agg()` function is used to calculate the maximum (max) and minimum (min) closing prices for each company within each year.

- **Plotting the Data:**

- A line plot is created to visualize the high and low stock prices per year for each company.
- The `sns.lineplot()` function plots the yearly maximum (max) and minimum (min) stock prices, with distinct styles for each:
 - Solid lines represent the maximum prices.

- Dashed lines represent the minimum prices.
 - Markers ('o') are used to mark the data points for better visualization.
- **Plot Styling:**
 - The plot includes a title ('Yearly High and Low Prices by Company'), labels for the x-axis ('Year') and y-axis ('Stock Price'), and a legend to distinguish between different companies.
 - The figure size is set to (14, 8) for clear visibility of trends.

This analysis provides insights into the highest and lowest stock prices each company reached every year, helping to visualize fluctuations and trends over time.

8. Visualize Closing Prices and Trading Volumes of Stocks

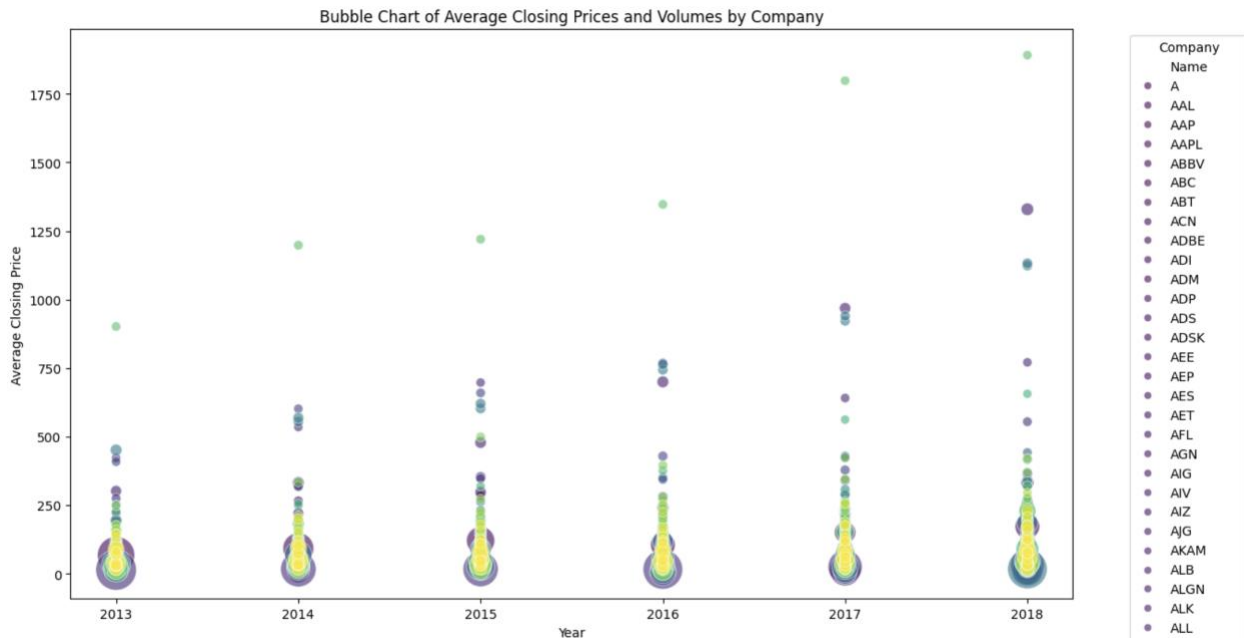
```
# Visualize stocks' closing prices and volume

# Ensure 'Date' is datetime and extract the year
df['date'] = pd.to_datetime(df['date'])
df['Year'] = df['date'].dt.year

# Calculate average closing price and average volume
per year for each company
yearly_avg = df.groupby(['Name', 'Year']).agg({'close':
'mean', 'volume': 'mean'}).reset_index()

# Bubble Chart
plt.figure(figsize=(14, 8))
bubble_plot = sns.scatterplot(
    data=yearly_avg,
    x='Year',
    y='close',
    size='volume',
    hue='Name',
    alpha=0.6,
    palette='viridis', # Color scheme
    sizes=(50, 1000) # Size range for bubbles
)
```

```
plt.title('Bubble Chart of Average Closing Prices and  
Volumes by Company')
plt.xlabel('Year')
plt.ylabel('Average Closing Price')
plt.legend(title='Company', bbox_to_anchor=(1.05, 1),
loc='upper left')
plt.show()
```



- **Data Preprocessing:**

- The `date` column is converted to `datetime` format using `pd.to_datetime()`.
- A new column `Year` is created by extracting the year from the `date` column, allowing grouping by year.

- **Calculate Yearly Averages:**

- The `groupby()` function groups the data by `Name` (company) and `Year`.
- The `agg()` function is used to calculate the average closing price (`close`) and average trading volume (`volume`) for each company per year.

- **Bubble Chart Creation:**

- The `sns.scatterplot()` function is used to create the bubble chart:
 - The x-axis represents the `Year`.
 - The y-axis represents the average `close` price.

- The bubble size represents the average `volume`, with larger bubbles indicating higher volumes.
- The `hue='Name'` assigns different colors to each company, and the `alpha=0.6` gives transparency to the bubbles.
- The `sizes=(50, 1000)` defines the range of bubble sizes, providing clarity on volume differences.

- **Plot Styling:**

- The chart is titled "Bubble Chart of Average Closing Prices and Volumes by Company".
- Labels for the x-axis (`'Year'`) and y-axis (`'Average Closing Price'`) are added.
- A legend is displayed to differentiate between companies, with the legend positioned outside the plot for better clarity.

This bubble chart allows easy comparison of stock trends over time for multiple companies, with the bubble sizes offering a clear representation of trading volumes. The color scheme (`viridis`) is chosen for its visually appealing gradient, helping to differentiate companies effectively

4. Categorized companies into sectors:

To gain a better understanding of stock market trends, it's crucial to separate companies based on their respective sectors. Each sector in the market operates under different economic influences, regulatory factors, and consumer demands, which can lead to unique market behaviors. For instance, technology companies might experience rapid growth during periods of innovation, while energy companies may be more influenced by fluctuations in oil prices. By categorizing companies into sectors, we can isolate these unique patterns and gain insights into how individual sectors are performing. This approach helps in identifying trends and making data-driven decisions tailored to sector-specific behaviors.

The code provided defines a mapping between company names and their respective sectors using a dictionary called `sector_mapping`. In this mapping, each sector (such as Technology, Healthcare, Finance, etc.) contains a list of companies that belong to that sector. This structure simplifies the process of organizing and analyzing stock data based on industry groupings. Instead of analyzing the stock market as a whole, this categorization allows us to focus on

individual sectors, providing more targeted insights into how companies within those sectors are performing.

1. Sector Categorization

By using this sector categorization, we can conduct more detailed analysis on stock trends within each sector. For example, by examining the performance of companies in the Technology sector, we can understand how factors like innovation, market competition, or regulatory changes impact their stock prices. Similarly, by focusing on the Healthcare sector, we can analyze trends driven by factors like medical breakthroughs or changes in healthcare policy. This segmentation improves our ability to detect sector-specific trends, draw comparisons between sectors, and make informed decisions based on the performance of individual industries.

```
# Map company names into categories
sector_mapping = {
    'Technology': ['AAPL', 'MSFT', 'GOOG', 'FB',
'INTC', 'NVDA', 'CSCO', 'ADBE', 'ORCL', 'IBM'],
    'Healthcare': ['JNJ', 'PFE', 'MRK', 'ABT', 'ABBV',
'BMJ', 'LLY', 'AMGN', 'MDT', 'CVS'],
    'Finance': ['JPM', 'BAC', 'GS', 'WFC', 'MS', 'C',
'AXP', 'BLK', 'BK', 'STT'],
    'Consumer': ['KO', 'PEP', 'PG', 'WMT', 'MCD',
'DIS', 'NKE', 'SBUX', 'TGT', 'YUM'],
    'Energy': ['XOM', 'CVX', 'COP', 'SLB', 'HAL',
'KMI', 'PSX', 'EOG', 'MPC', 'PXD'],
    'Industrials': ['BA', 'HON', 'CAT', 'GE', 'UPS',
'MMM', 'RTX', 'LMT', 'DE', 'ITW'],
    'Others': ['ZTS', 'TSN', 'DHR', 'V', 'MA', 'PYPL',
'T', 'VZ', 'CMCSA', 'AMZN']
}
```

- **Mapped company names to sectors:** The companies were organized into distinct categories based on their industry sector. This categorization includes sectors such as Technology, Healthcare, Finance, Consumer, Energy, Industrials, and Others. This step enables targeted analysis of stock trends within each sector, providing more granular insights into performance across different market segments.

```
# Function to map company names to sectors
def map_sector(name):
    for sector, companies in sector_mapping.items():
        if name in companies:
            return sector
    return 'Uncategorized'
```

- **Created a function to map company names to sectors:** A function, `map_sector()`, was developed to assign each company to its respective sector based on the predefined sector mapping. The function iterates through the `sector_mapping` dictionary, checking if a company name exists in any of the sector lists and returns the corresponding sector name. If the company name is not found in any sector list, the function categorizes it as 'Uncategorized'. This ensures that each company is correctly classified for further analysis.

```
# Apply mapping to dataset
df['Sector'] = df['Name'].apply(map_sector)
```

- **Applied sector mapping to the dataset:** The `map_sector()` function was applied to the dataset by using the `apply()` method on the 'Name' column. This resulted in the creation of a new column called 'Sector' in the dataset, where each company is assigned to its respective sector. This enables the categorization of companies and allows for sector-specific analysis of stock trends.

```
# Validate the mapping
print(df['Sector'].value_counts())
```

□ **Applied sector mapping to the dataset:** The `map_sector` function was applied to the Name column of the dataset to assign a sector to each company. The new sector information was added as a column named 'Sector'. This step allows for the classification of companies into their respective sectors for better analysis and trend identification.

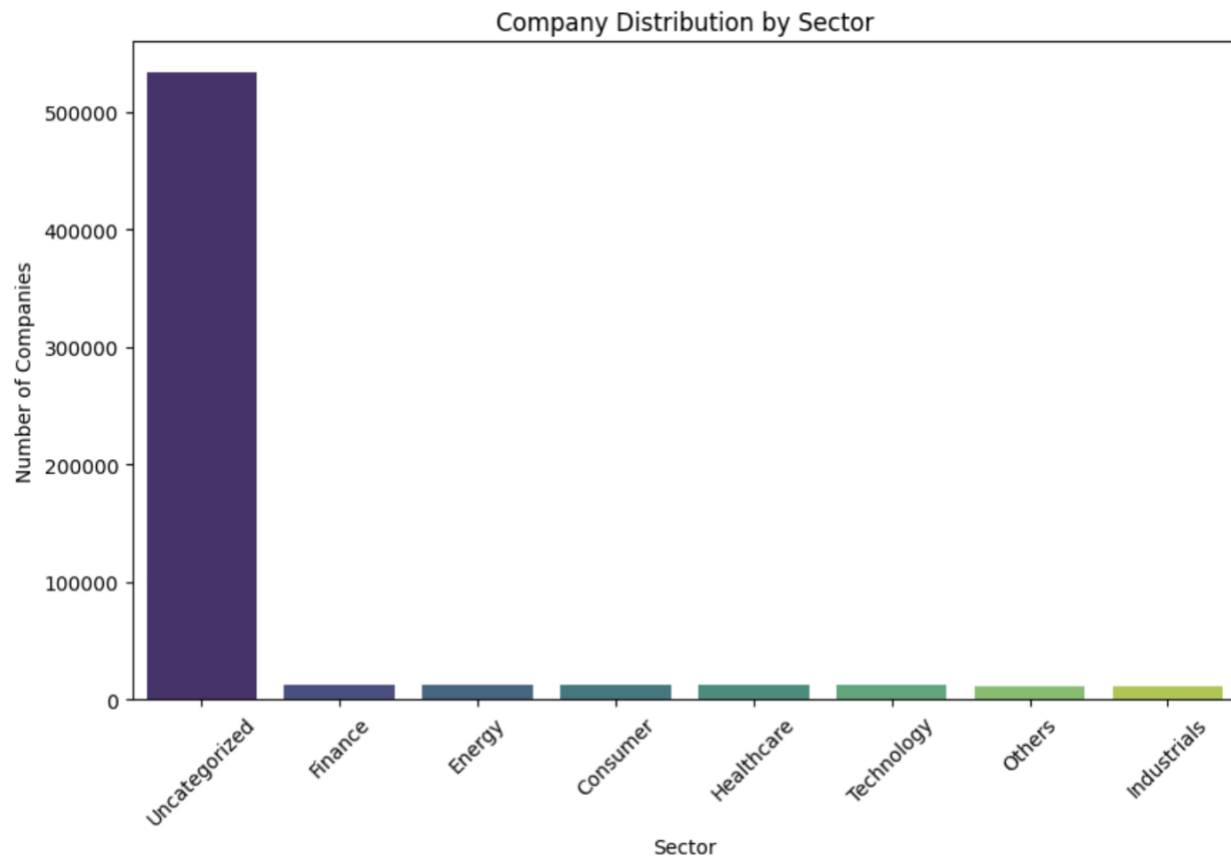
□ **Validation of sector mapping:** After applying the mapping, the distribution of sectors was checked using the `value_counts()` function. The result shows the

number of occurrences of each sector. The 'Uncategorized' category has the highest count, indicating that many companies in the dataset were not mapped to any of the predefined sectors. The other sectors, such as 'Finance', 'Energy', 'Healthcare', and 'Technology', have smaller counts, reflecting the companies categorized into these groups. This confirms that the mapping process was successful, although some companies still remain uncategorized.

2. Bar Chart Representing Company Distribution Across Sectors

```
# Bar plot for company distribution across sectors
import seaborn as sns
import matplotlib.pyplot as plt

plt.figure(figsize=(10, 6))
sns.countplot(data=df, x='Sector',
order=df['Sector'].value_counts().index,
palette='viridis')
plt.title("Company Distribution by Sector")
plt.ylabel("Number of Companies")
plt.xlabel("Sector")
plt.xticks(rotation=45)
plt.show()
```



- **Visualized company distribution across sectors:** A bar plot was generated to show the number of companies in each sector. The countplot function from the Seaborn library was used to plot the distribution of companies in the Sector column. The order parameter was set to display the sectors in descending order based on the number of companies, using `df['Sector'].value_counts().index`.
- **Plot customization:** The plot was customized with the viridis color palette, which provides a visually appealing color range for the bars. The x-axis labels were rotated by 45 degrees to make the sector names more readable. The plot title, "Company Distribution by Sector," was added, along with labels for the y-axis ("Number of Companies") and x-axis ("Sector"). This bar plot provides a clear visual representation of how companies are distributed across different sectors in the dataset.
- **Output:**
 - The "Uncategorized" sector contains the majority of the data with 533,077 entries, indicating a large portion of companies have not been mapped to specific sectors.

- The sectors with the most companies, excluding "Uncategorized," are "Energy," "Finance," and "Consumer," each with 12,590 companies.
- The "Healthcare" sector has 12,588 companies, which is slightly lower than the "Energy," "Finance," and "Consumer" sectors.
- The "Technology" sector contains 12,304 companies, making it one of the larger sectors.
- The "Others" sector has 11,970 companies, and the "Industrials" sector has 11,331 companies.
- The distribution shows a generally balanced allocation across sectors, except for the large number of "Uncategorized" entries, which likely require further categorization or investigation.

```
# Filter out uncategorized data
categorized_data = df[df['Sector'] != 'Uncategorized']
```

- The code filters out the data entries that belong to the "Uncategorized" sector, creating a new dataset (categorized_data) that only includes companies that are assigned to one of the defined sectors.
- By excluding the "Uncategorized" data, the dataset becomes more focused and easier to analyze, as only companies that have been successfully mapped to a sector will be considered.
- This step helps ensure that the analysis is based on relevant, sector-specific data, improving the accuracy and relevance of any subsequent insights or visualizations.

3. Feature Engineering:

Feature engineering is the process of creating new features or modifying existing ones in a dataset to improve the performance of machine learning models.

```
# Calculate daily price range
df['price_range'] = df['high'] - df['low']
```

Calculate daily price range:

- The code creates a new feature called `price_range` by subtracting the **low price** from the **high price** for each stock on a given day. This feature reflects the range within which the stock traded during the day, indicating its volatility.

```
# Calculate average price
df['average_price'] = (df['open'] + df['close']) / 2
```

Calculate average price:

- A new feature, `average_price`, is created by taking the average of the **open** and **close** prices for each stock. This provides a sense of the stock's average value during the trading day.

```
# Calculate daily returns
df['daily_return'] = df['close'].pct_change()
```

Calculate daily returns:

- The `daily_return` feature is calculated using the percentage change in the **close price** from the previous day to the current day. This indicates the daily price movement of the stock and is useful for understanding trends and stock behavior.

```
# Add year and month for time-based analysis
df['year'] = df['date'].dt.year
df['month'] = df['date'].dt.month
```

Add year and month for time-based analysis:

- The year and month features are extracted from the **date** column. This allows for time-based analysis, such as analyzing stock performance over specific months or years. These features help in identifying seasonality and long-term trends.

4. Data Visualization

1. Average daily return by sector

```
# Data Visualization
# Average daily return by sector
plt.figure(figsize=(10, 6))
sns.boxplot(data=categorized_data, x='Sector',
y='daily_return', palette='coolwarm')
plt.title("Daily Returns by Sector")
plt.ylim(-0.01, 0.015)
plt.show()
```



The code provided creates a box plot to visualize the average daily returns for each sector. Here's what each part of the code does:

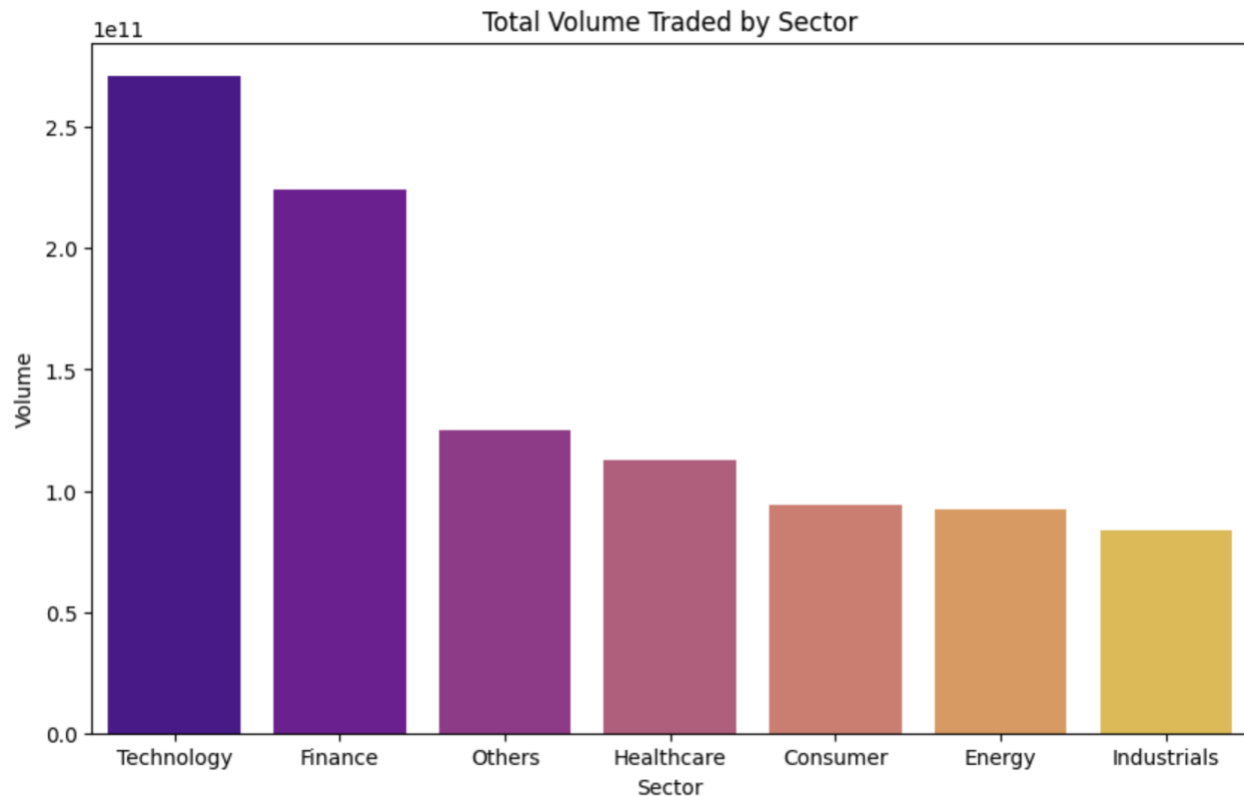
- **sns.boxplot(data=categorized_data, x='Sector', y='daily_return', palette='coolwarm'):**
 - This line generates a box plot using Seaborn, where the x-axis represents the different sectors ('Sector'), and the y-axis represents the daily returns ('daily_return').

- The coolwarm palette is applied to color the plot, helping to visually distinguish the different sectors.
- **plt.title("Daily Returns by Sector"):**
 - This sets the title of the plot to "Daily Returns by Sector," providing context for what the plot represents.
- **plt.ylim(-0.01, 0.015):**
 - This line limits the range of the y-axis (daily return values) between -0.01 and 0.015 to focus on the region where most of the data lies, avoiding extreme outliers and making the plot easier to interpret.
- **plt.show():**
 - This displays the plot to the screen.

The box plot visually represents the distribution of daily returns for each sector, showing the median, quartiles, and potential outliers in the data. It helps to compare the performance of different sectors based on their daily returns.

2. Total Trading Volume by Sector

```
# Volume traded per sector
plt.figure(figsize=(10, 6))
sector_volume =
categorized_data.groupby('Sector')['volume'].sum().sort
_values(ascending=False)
sns.barplot(x=sector_volume.index,
y=sector_volume.values, palette='plasma')
plt.title("Total Volume Traded by Sector")
plt.ylabel("Volume")
plt.xlabel("Sector")
plt.show()
```



- 1. Group the data by Sector and calculate the total volume traded:**
 - The `groupby('Sector')` groups the data by the sector.
 - `['volume'].sum()` calculates the total volume traded in each sector.
 - `.sort_values(ascending=False)` sorts the sectors by total volume in descending order.
- 2. Create a bar plot:**
 - `sns.barplot(x=sector_volume.index, y=sector_volume.values, palette='plasma')` creates a bar plot using the calculated `sector_volume` data. The x axis represents the sector names, and the y axis represents the total volume traded in each sector.
 - `palette='plasma'` specifies the color palette for the bars.
- 3. Add title and labels:**
 - `plt.title("Total Volume Traded by Sector")` adds a title to the plot.
 - `plt.ylabel("Volume")` labels the y-axis as "Volume."
 - `plt.xlabel("Sector")` labels the x-axis as "Sector."
- 4. Show the plot:**
 - `plt.show()` displays the plot.
- 5. Output:**

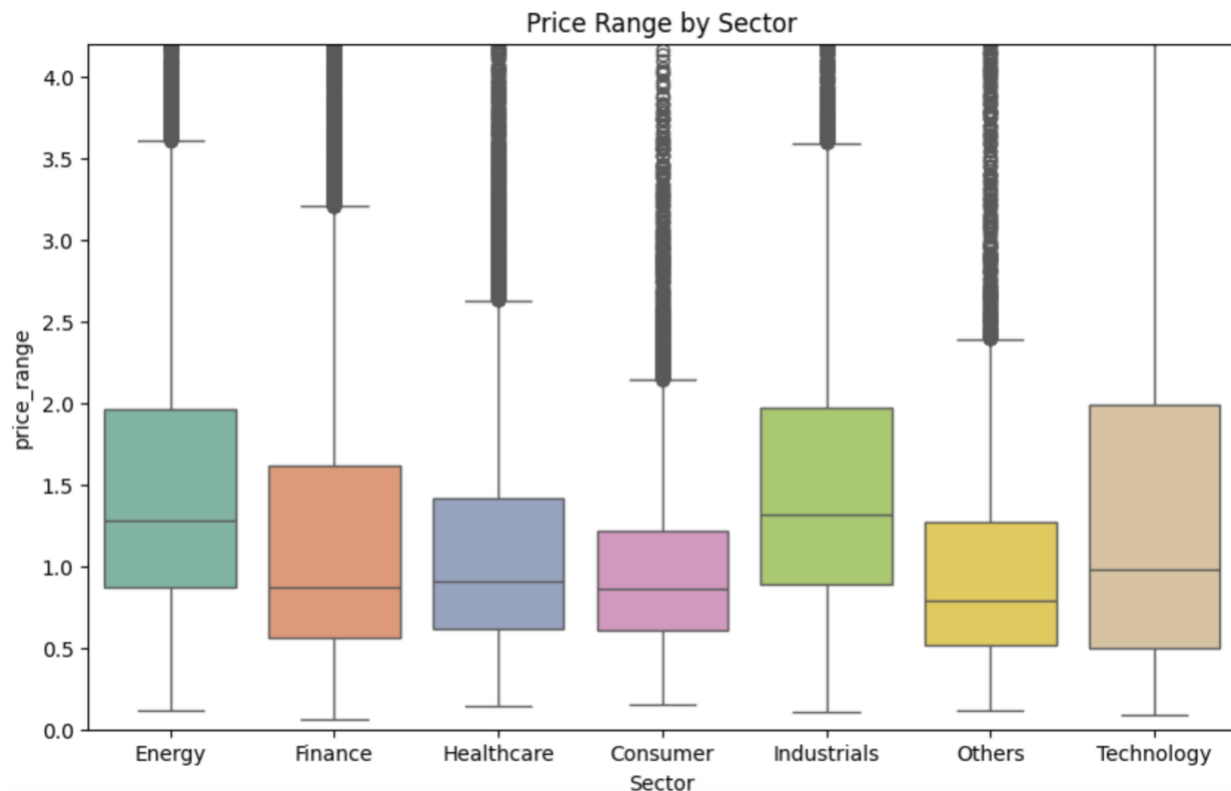
The output represents the total volume of stocks traded for each sector, where the values indicate the sum of trading volumes across all companies in each sector:

- **Technology:** 271,255,140,330
- **Finance:** 223,989,322,224
- **Others:** 124,699,843,918
- **Healthcare:** 112,570,856,106
- **Consumer:** 94,065,752,181
- **Energy:** 92,396,999,483
- **Industrials:** 83,400,026,367

These values show the aggregated volume of traded stocks for companies within each sector. The Technology sector leads with the highest trading volume, followed by Finance, while Industrials and Energy have relatively lower trading volumes.

3. Price Range Distribution Across Sectors

```
# Price range analysis per sector
plt.figure(figsize=(10, 6))
sns.boxplot(data=categorized_data, x='Sector',
y='price_range', palette='Set2')
plt.title("Price Range by Sector")
plt.ylim(0, df['price_range'].quantile(0.95))
plt.show()
```



- **sector_price_range = categorized_data.groupby('Sector')['price_range'].describe():** This computes descriptive statistics (like mean, std, min, 25th percentile, median, 75th percentile, and max) for the price_range within each sector.
- **print(sector_price_range):** This prints the statistical summary for the price range in each sector to the console.
- **Boxplot:** The code will generate a boxplot showing the distribution of the price range across sectors.

This way, you will see the detailed summary values (like mean, min, max, etc.) for the price range per sector printed in the output along with the visual boxplot.

4. Trends of Monthly Average Prices

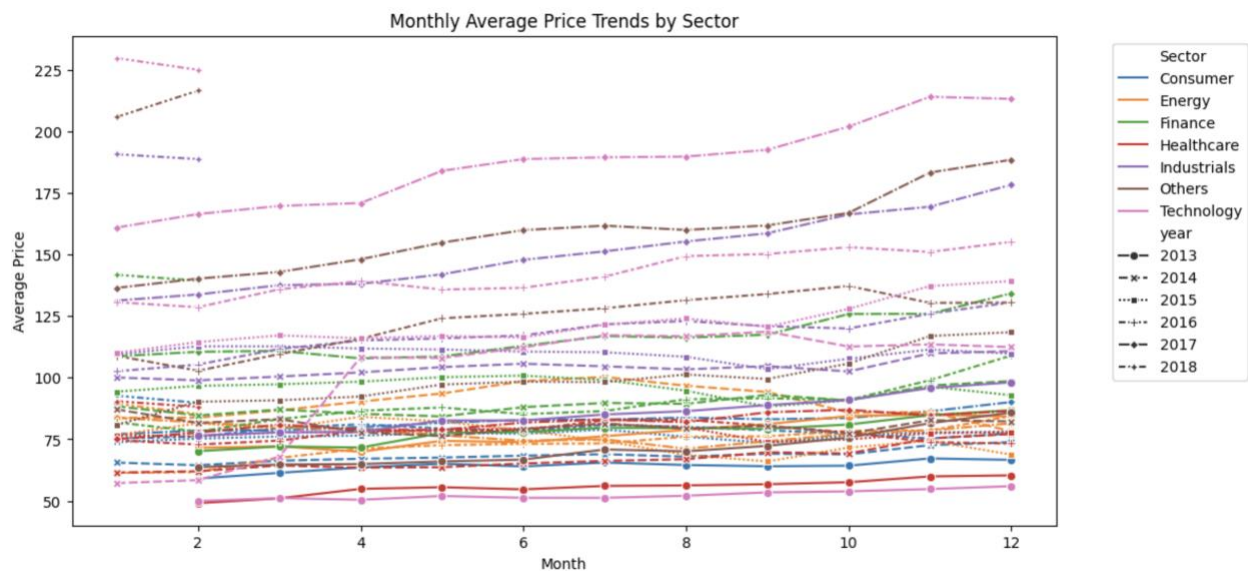
```
# Monthly average price trends
monthly_avg = categorized_data.groupby(['year',
    'month',
    'Sector'])['average_price'].mean().reset_index()

plt.figure(figsize=(12, 6))
```

```

sns.lineplot(data=monthly_avg, x='month',
y='average_price', hue='Sector', style='year',
markers=True)
plt.title("Monthly Average Price Trends by Sector")
plt.xlabel("Month")
plt.ylabel("Average Price")
plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left')
plt.show()

```



- The plot visualizes the monthly average stock price trends across different sectors, allowing us to compare how each sector's stock prices behave over the course of the year.
- By grouping the data by year, month, and sector, the plot calculates the average price for each sector on a monthly basis. This helps to smooth out daily fluctuations and gives a clearer view of the overall price trends over time.
- The lines in the plot represent the stock price trends for different sectors, where each sector is assigned a unique color for easy differentiation. The plot also includes markers on the lines to indicate the exact average price for each month.
- The x-axis represents the months of the year, while the y-axis shows the calculated average stock price for each sector in that month. This allows us to track how the average price of stocks in each sector changes over time.

- The plot includes multiple years, with each year shown with a different style (solid or dashed lines), making it easy to compare how the price trends evolve across different years for each sector.
- This kind of analysis helps identify specific periods of growth or decline within each sector, such as a spike in the stock price or a consistent decline. It can also reveal seasonal patterns or shifts in market sentiment towards particular sectors. For example, some sectors might show a rise in prices during certain months due to seasonal demand, while others may experience steady growth or volatility throughout the year.
- Overall, this visualization is a powerful tool for understanding long-term trends and comparing sector performances over time, allowing investors, analysts, and stakeholders to make more informed decisions based on historical price behavior.

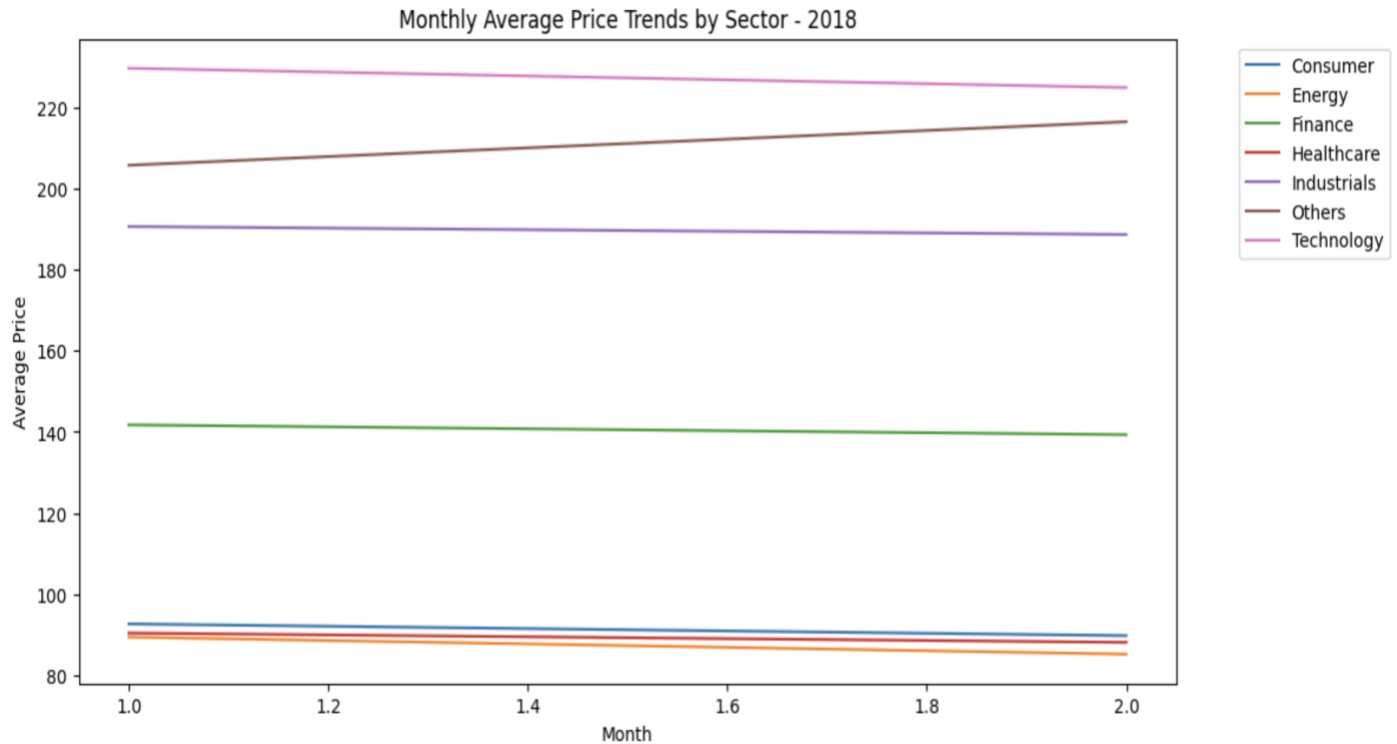
5. Monthly Average Price Trends Visualization for the Last Year by Sector

```
last_year = categorized_data['year'].max()

# Filter for the last year
last_year_data =
categorized_data[categorized_data['year'] == last_year]

# Calculate monthly average price trends for the last
year
monthly_avg_last_year = last_year_data.groupby(['year',
'month',
'Sector'])['average_price'].mean().reset_index()

# Plotting monthly average price trends for the last
year
plt.figure(figsize=(12, 6))
sns.lineplot(data=monthly_avg_last_year, x='month',
y='average_price', hue='Sector', markers=True)
plt.title(f"Monthly Average Price Trends by Sector -
{last_year}")
plt.xlabel("Month")
plt.ylabel("Average Price")
plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left')
plt.show()
```



- **Find the Most Recent Year:**

- `last_year = categorized_data['year'].max():` This line identifies the most recent year in the dataset by finding the maximum value in the year column.

- **Filter Data for the Last Year:**

- `last_year_data = categorized_data[categorized_data['year'] == last_year]:` This filters the categorized_data DataFrame to include only the data for the most recent year.

- **Calculate Monthly Average Prices for the Last Year:**

- `monthly_avg_last_year = last_year_data.groupby(['year', 'month', 'Sector'])['average_price'].mean().reset_index():` This groups the data by year, month, and Sector and calculates the average price for each sector per month in the most recent year. The result is a DataFrame with the monthly average price for each sector.

- **Plot the Monthly Average Price Trends:**

- `sns.lineplot(data=monthly_avg_last_year, x='month', y='average_price', hue='Sector', markers=True)`: This creates a line plot where the x-axis represents the months of the year, the y-axis represents the average stock price, and each line corresponds to a different sector. The `hue='Sector'` argument assigns a unique color to each sector, while `markers=True` adds markers to the data points.

- **Title and Labels:**

- `plt.title(f"Monthly Average Price Trends by Sector - {last_year}")`: The title of the plot is dynamically generated to include the most recent year.
- `plt.xlabel("Month")` and `plt.ylabel("Average Price")`: The labels for the x-axis (months) and y-axis (average price) are added.

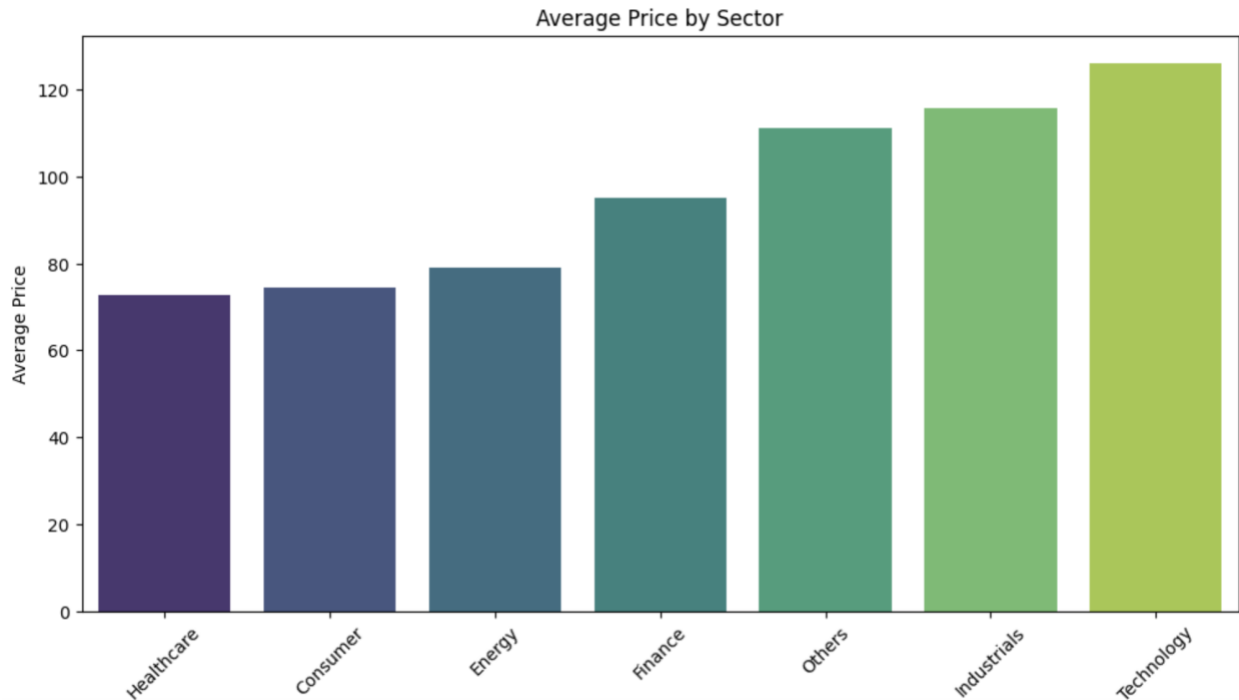
- **Legend and Layout:**

- `plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left')`: The legend, which shows which color corresponds to each sector, is placed outside the plot to avoid overlapping with the data.

This visualization helps to analyze how the average stock prices of various sectors have moved throughout the last year, making it easier to compare trends across sectors and identify any significant shifts or patterns in the market.

6. Analyze the average stock price across different sectors.

```
# Compare the average stock price across sectors.
sector_avg_price =
categorized_data.groupby('Sector')['close'].mean().sort
_values()
plt.figure(figsize=(12, 6))
sns.barplot(x=sector_avg_price.index,
y=sector_avg_price.values, palette='viridis')
plt.xticks(rotation=45)
plt.title("Average Price by Sector")
plt.xlabel("Sector")
plt.ylabel("Average Price")
plt.show()
```



- **Data Aggregation:** It groups the dataset by 'Sector' and calculates the average closing price (close) for each sector.
- **Sorting:** The sectors are sorted by their average closing price in ascending order.
- **Plotting:** A bar plot is created to display the average stock price for each sector, with the sectors labeled on the x-axis and the average price on the y-axis. The viridis color palette is used to represent the data visually.
- **Title and Labels:** The plot is given a title, and axis labels are set for clarity. The x-axis labels (sectors) are rotated for better readability.
- **Output:**

Healthcare: The average closing price for companies in the healthcare sector is **\$72.71**.

Consumer: The average closing price for companies in the consumer sector is **\$74.57**.

Energy: The average closing price for companies in the energy sector is **\$78.95**.

Finance: The average closing price for companies in the finance sector is **\$95.22**.

Others: The average closing price for companies in the "Others" sector is **\$111.02**.

Industrials: The average closing price for companies in the industrials sector is **\$115.84**.

Technology: The average closing price for companies in the technology sector is **\$126.10**.

These values help you understand how the stock prices of companies in different sectors compare to each other. The technology sector has the highest average closing price, while healthcare has the lowest.

This chart gives an overview of how the average stock price varies across sectors.

7. Sector Level Summary

```
# Print sector-level summary
summary = categorized_data.groupby('Sector').agg({
    'daily_return': ['mean', 'std'],
    'volume': 'sum',
    'price_range': 'mean'
})
print(summary)
```

Output:

	daily_return		volume	price_range
	mean	std	sum	mean
Sector				
Consumer	0.000775	0.045591	94065752181	1.002866
Energy	0.000063	0.023651	92396999483	1.696958
Finance	0.000854	0.038637	223989322224	1.583160
Healthcare	0.000187	0.019882	112570856106	1.205389
Industrials	0.000604	0.020765	83400026367	1.595064
Others	0.000629	0.023936	124699843918	1.927624
Technology	0.001556	0.111989	271255140330	2.075263

- 1. Daily Return:** This metric shows the average daily return and its standard deviation across each sector.

- **Technology** has the highest average daily return (**0.001556**), suggesting that, on average, stocks in this sector have provided the most positive return. However, it also has the highest volatility (**std = 0.111989**), indicating more fluctuation in returns.
 - **Healthcare** has the lowest average daily return (**0.000187**) but also the lowest standard deviation (**std = 0.019882**), indicating more stability with smaller fluctuations in daily returns.
2. **Volume:** This metric represents the total trading volume for companies within each sector.
- **Technology** stands out with the highest total trading volume (**271,255,140,330**), reflecting a higher level of market activity and investor interest in this sector.
 - **Industrials** has the lowest trading volume (**83,400,026,367**), indicating less market activity or smaller companies in this sector relative to others.
3. **Price Range:** The average daily price range (difference between high and low prices) provides an idea of how much price fluctuations occur within each sector.
- **Technology** has the highest average price range (**2.075263**), which means that stocks in this sector have the most price variation on a daily basis.
 - **Healthcare** has the smallest price range (**1.205389**), reflecting relatively more stable prices within this sector.

Key Observations:

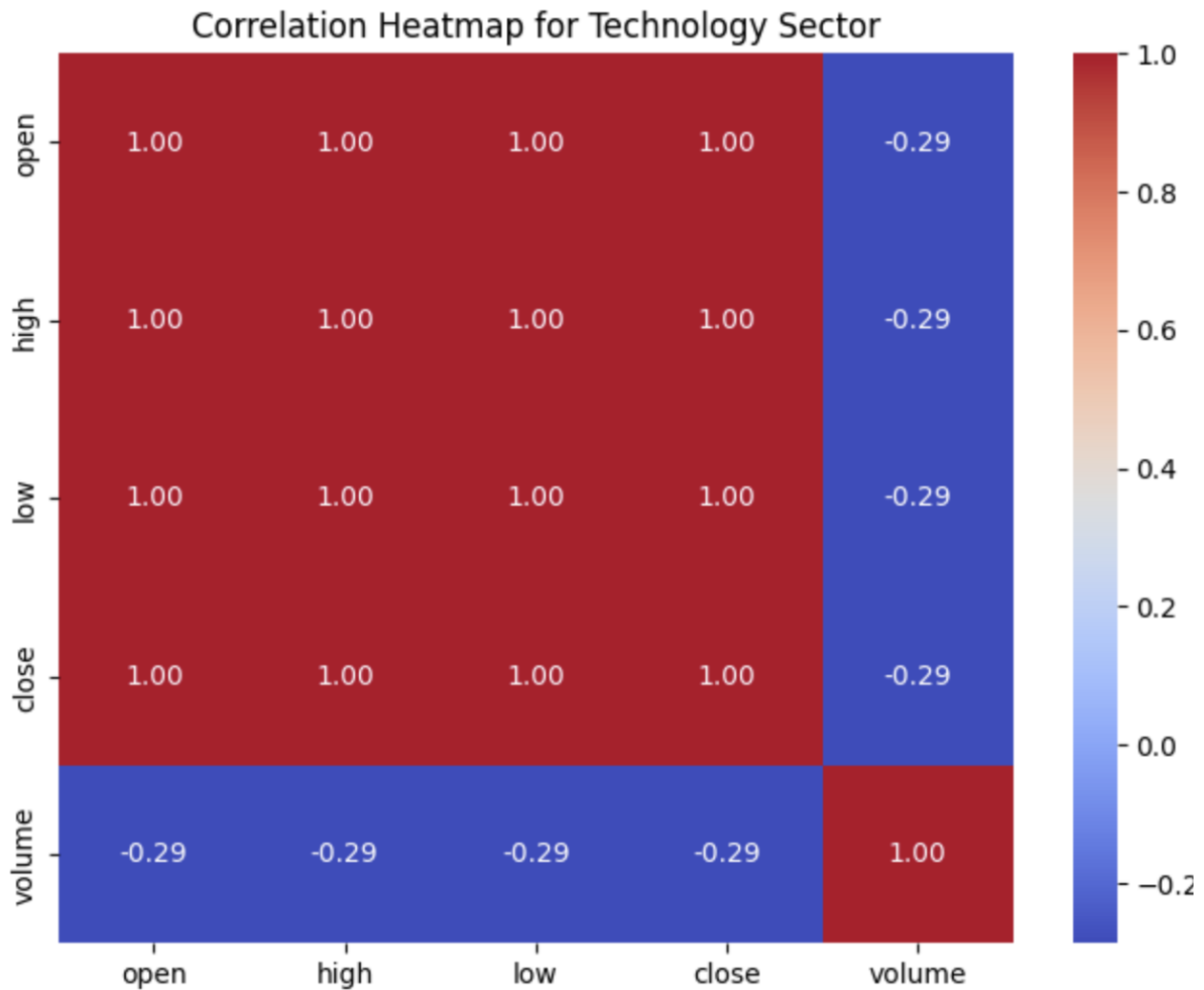
- **Technology** leads in daily return and volume, but it also has the highest price fluctuations.
- **Healthcare** and **Energy** show lower returns and price fluctuations compared to other sectors.
- **Industrials** appears to be the least active in terms of both volume and returns.

This analysis provides insight into which sectors are more volatile, have higher trading activity, and offer more potential for returns. It also helps identify sectors that may be safer or more stable for investment.

8. Correlation heatmap for the Technology sector

```
# Correlation heatmap for a single sector (e.g.,
Technology)
tech_data = categorized_data[categorized_data['Sector']
== 'Technology']
correlation = tech_data[['open', 'high', 'low',
'close', 'volume']].corr()

plt.figure(figsize=(8, 6))
sns.heatmap(correlation, annot=True, cmap='coolwarm',
fmt='.2f')
plt.title("Correlation Heatmap for Technology Sector")
plt.show()
```



The code creates a **correlation heatmap** specifically for the **Technology** sector, focusing on the relationship between key stock attributes: opening price, highest price of the day, lowest price, closing price, and the trading volume. Here's the breakdown:

1. **Filtering Data:** The dataset is filtered to focus solely on the **Technology** sector.
2. **Correlation Calculation:** It calculates the Pearson correlation coefficient between several price-related variables and volume. The correlation coefficient measures the degree of linear relationship between two variables, ranging from -1 (perfect negative correlation) to +1 (perfect positive correlation).
3. **Visualization:** The code generates a heatmap that visually represents the correlation matrix, with color gradients to reflect positive and negative

correlations. The values of the correlation coefficients are annotated on the heatmap.

Explanation of the Output:

The output is a **correlation matrix** that displays the relationship between the following variables for the **Technology** sector:

1. **High Correlation Among Price Variables:**

- The open, high, low, and close prices show extremely high positive correlations, all near 1. This means that these prices tend to move in the same direction during the trading day. For instance, the high price is very similar to the opening price, low price, and closing price.

2. **Negative Correlation with Volume:**

- There is a weak negative correlation between the price variables and **volume**. The correlation values between volume and the price-related variables range from -0.286 to -0.287. This suggests that higher trading volumes might slightly correspond to a slight decrease in stock prices, but the correlation is not strong enough to draw firm conclusions.

3. **Price Consistency:**

- The prices (open, high, low, and close) in the **Technology** sector are highly consistent, as shown by the high correlation values between them. This indicates that the prices follow a predictable pattern during the day.

Trends Observed:

- **Strong Price Correlation:** The prices (open, high, low, close) are closely related, reflecting that stock prices in the **Technology** sector behave in a consistent manner during the trading day.
- **Volume-Price Inverse Trend:** There is a weak inverse relationship between **volume** and stock prices, suggesting that when trading volume increases, there may be a slight decline in prices. However, this relationship is not strong enough to be definitive.

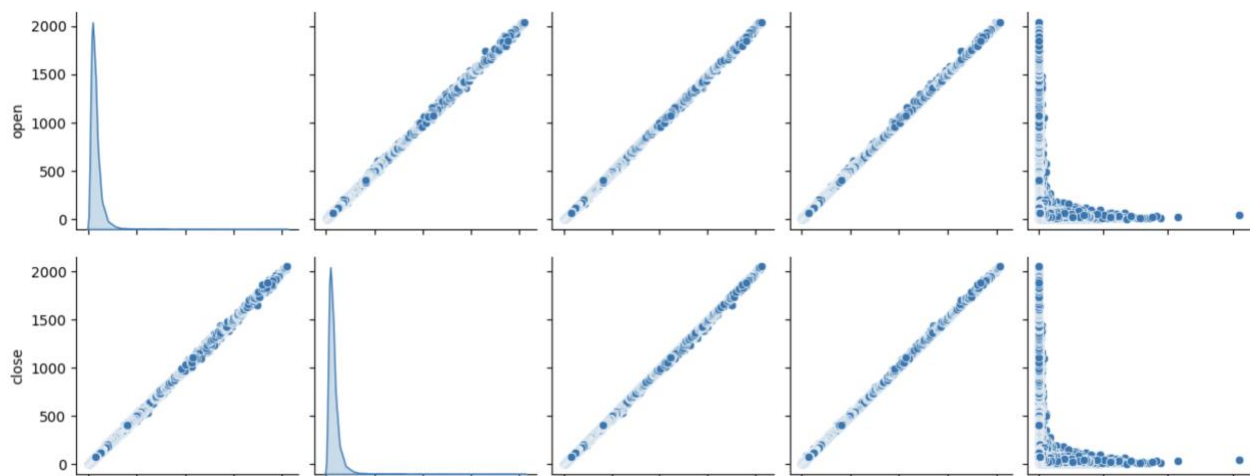
This analysis provides insights into how the **Technology** sector behaves in terms of price movement and volume, which can help in understanding market dynamics and trading strategies.

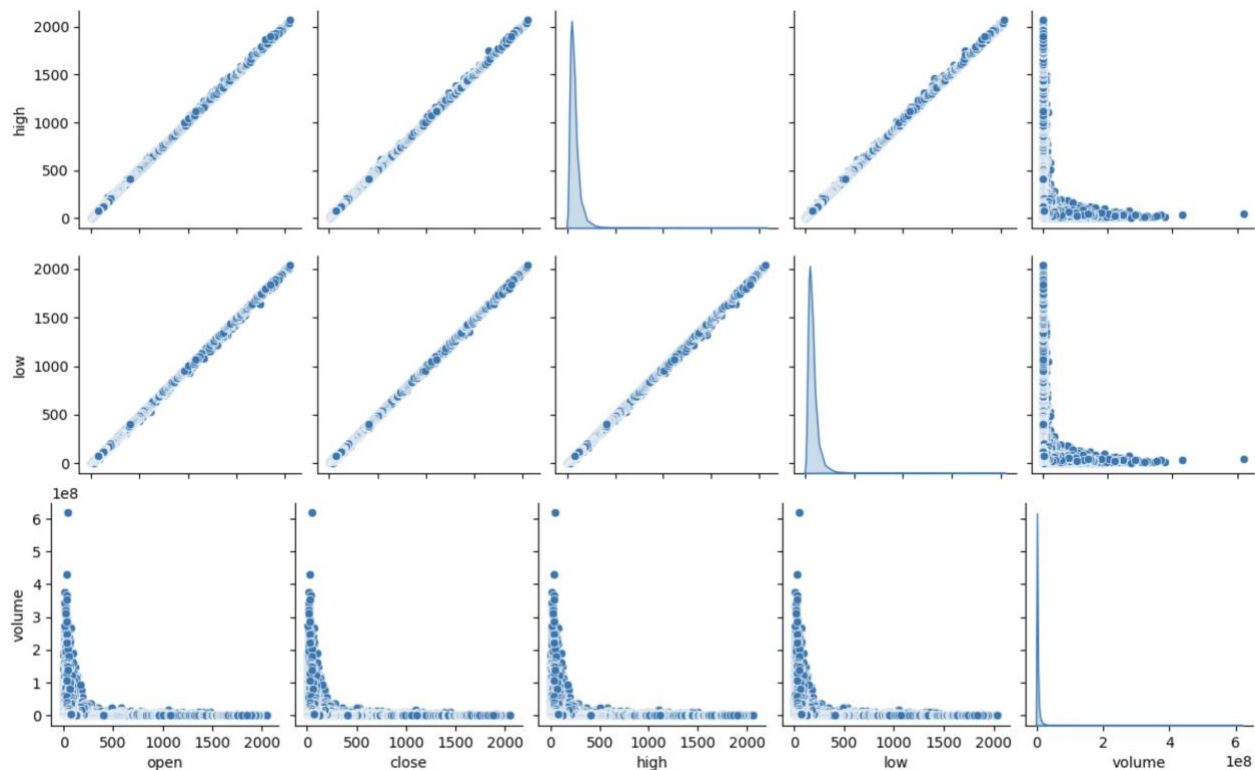
9. Analyze pairwise relationships between features

```
# Analyze pairwise relationships between features
sns.pairplot(df[['open', 'close', 'high', 'low',
'volume']], diag_kind='kde', palette='coolwarm')
plt.show()
```

Output:

	open	close	high	low	volume
open	1.000000	0.999872	0.999939	0.999927	-0.142705
close	0.999872	1.000000	0.999936	0.999939	-0.142802
high	0.999939	0.999936	1.000000	0.999903	-0.142316
low	0.999927	0.999939	0.999903	1.000000	-0.143240
volume	-0.142705	-0.142802	-0.142316	-0.143240	1.000000





The code generates a pair plot for numerical features like 'open', 'close', 'high', 'low', and 'volume' in the dataset. A pair plot is a matrix of scatterplots where each variable is plotted against every other variable, and the diagonal displays the kernel density estimate (KDE) for the individual variables. This helps to visualize pairwise relationships, trends, and distributions in the data.

Explanation of Output:

- **Diagonal KDE plots:** These represent the distribution of each individual feature, showing how the values are spread.
- **Scatterplots off-diagonal:** Each scatterplot shows the relationship between two variables. For example, the plot between 'open' and 'close' shows a strong linear relationship, as the correlation coefficient is 0.999872.

Key Trends:

- **High correlations between features:** The 'open', 'close', 'high', and 'low' features exhibit almost perfect linear relationships, with correlation values close to 1. This indicates that these stock prices tend to move similarly across different trading metrics.

- **Weak correlation with volume:** All the price features ('open', 'close', 'high', and 'low') have a weak negative correlation with 'volume', with correlation coefficients between -0.14 and -0.14. This suggests that there is little relationship between trading volume and stock price movements.

Pairwise relationships of numerical features in the Healthcare sector

10. Pairwise relationships of numerical features in the Healthcare sector

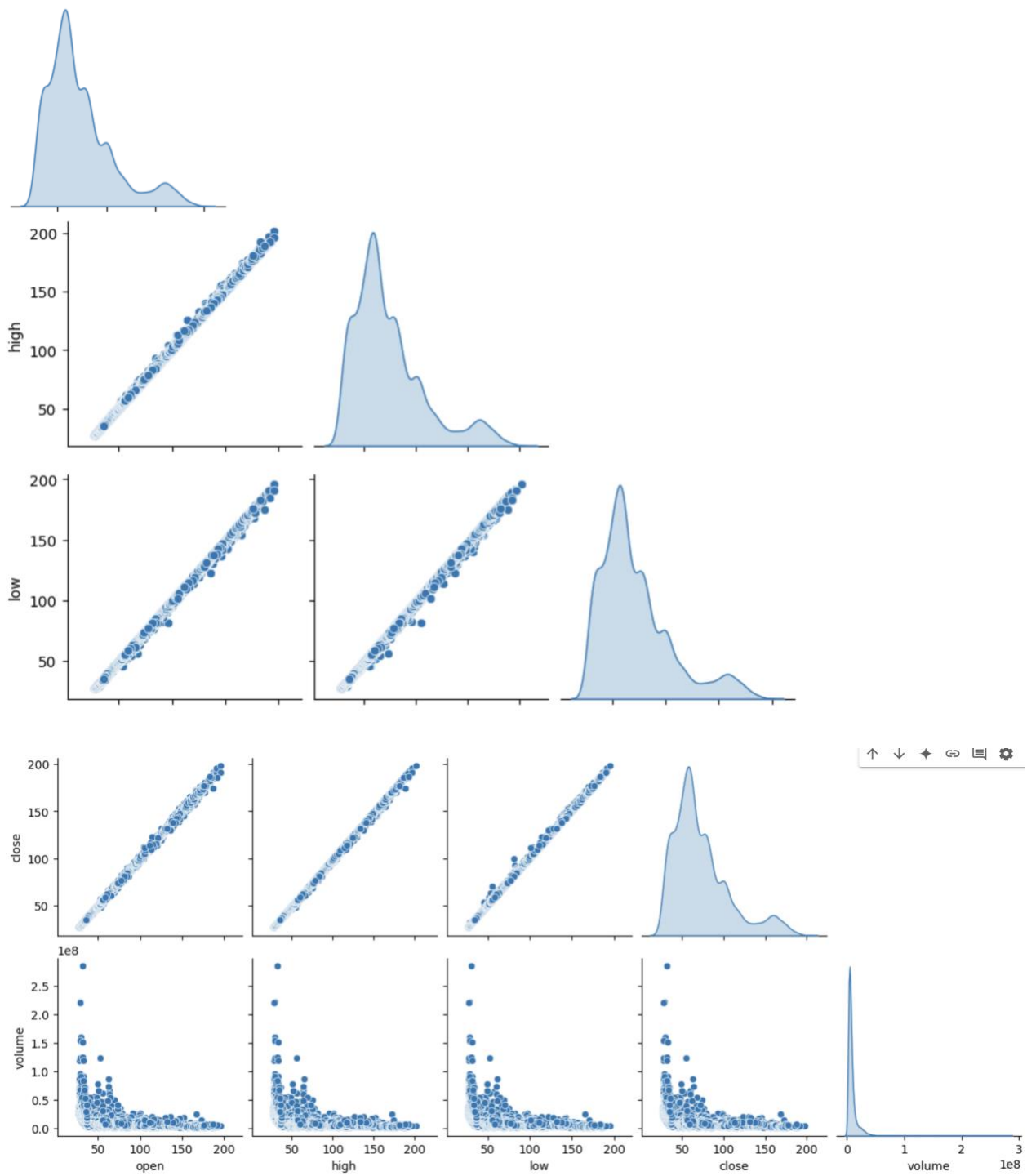
```
# Pair plot for numerical features in Healthcare sector
healthcare_data =
categorized_data[categorized_data['Sector'] ==
'Healthcare']
sns.pairplot(healthcare_data[['open', 'high', 'low',
'close', 'volume']], diag_kind='kde', corner=True)
plt.suptitle("Pair Plot for Healthcare Sector", y=1.02)
plt.show()
```

Output:

	open	high	low	close	volume
open	1.000000	0.999821	0.999782	0.999644	-0.382298
high	0.999821	1.000000	0.999694	0.999825	-0.380539
low	0.999782	0.999694	1.000000	0.999801	-0.384529
close	0.999644	0.999825	0.999801	1.000000	-0.382611
volume	-0.382298	-0.380539	-0.384529	-0.382611	1.000000

	open	high	low	close	volume
count	12588.000000	12588.000000	12588.000000	12588.000000	1.258800e+04
mean	72.693444	73.292171	72.086783	72.710563	8.942712e+06
std	34.115650	34.405707	33.799110	34.112999	9.794686e+06
min	26.890000	27.030000	26.790000	26.840000	8.176240e+05
25%	49.510000	49.917500	49.067500	49.430000	4.231352e+06
50%	63.065000	63.590000	62.500000	63.065000	6.226334e+06
75%	86.352500	86.900000	85.722500	86.380000	9.611530e+06
max	196.210000	201.230000	195.650000	198.000000	2.844681e+08

Pair Plot for Healthcare Sector



The code creates a pair plot to visualize the relationships between various numerical features (such as open, high, low, close, and volume) for the stocks in

the Healthcare sector. The pair plot is a useful way to understand how different pairs of features relate to one another. This visualization includes:

- Diagonal histograms (`diag_kind='kde'`) to show the distribution of each feature.
- A `corner=True` argument, which omits the upper triangle of plots to avoid redundancy (as the relationship is symmetric).
- The plot is titled "Pair Plot for Healthcare Sector," which provides a contextual understanding of the data being analyzed.

Explanation of the Output:

The output consists of two parts:

1. Correlation Matrix:

- The pairwise correlation coefficients between each pair of numerical features (open, high, low, close, volume) are displayed. The values range between -1 and 1, indicating the strength and direction of relationships:
 - The open, high, low, and close features exhibit very strong positive correlations with each other (all around 0.999). This suggests that these variables move together very closely, which is typical in stock data where these values represent the same stock on the same day.
 - The volume feature has a negative correlation with the price-related variables (around -0.38), which indicates that as the stock price tends to rise, the trading volume tends to decrease and vice versa.

2. Descriptive Statistics:

- The summary statistics (mean, standard deviation, min, max, etc.) for each feature are provided:
 - The **mean** values for the open, high, low, and close prices are all close to each other (~72.7 for close).
 - The **standard deviation** for the price-related features (open, high, low, close) is relatively high (~34), indicating significant price variation.
 - The **volume** feature shows a much higher variation, with a wide range from 8.18 million to 284 million, suggesting that trading volume can fluctuate greatly.

Observed Trends:

- **Strong Price Correlations:** The open, high, low, and close prices are highly correlated with each other, indicating that the stock prices are moving together in a predictable manner over time. This is expected in financial markets where these price metrics represent the same stock over the course of the day.
- **Negative Relationship Between Price and Volume:** The negative correlation between the price-related features and volume suggests that when stock prices increase, the volume of trades tends to decrease. This is typical of many stocks, as higher prices may reduce the number of trades, while lower prices might encourage more trading activity.
- **Distribution:** The pair plot and descriptive statistics highlight that the data is not uniform. There is a significant range in the open, high, low, and close prices, indicating the presence of price volatility in the healthcare sector.

In summary, the healthcare sector's stock prices exhibit strong relationships with each other, while trading volume appears to be inversely related to stock price movements. The statistical summary further emphasizes the variation and volatility in stock prices within this sector.

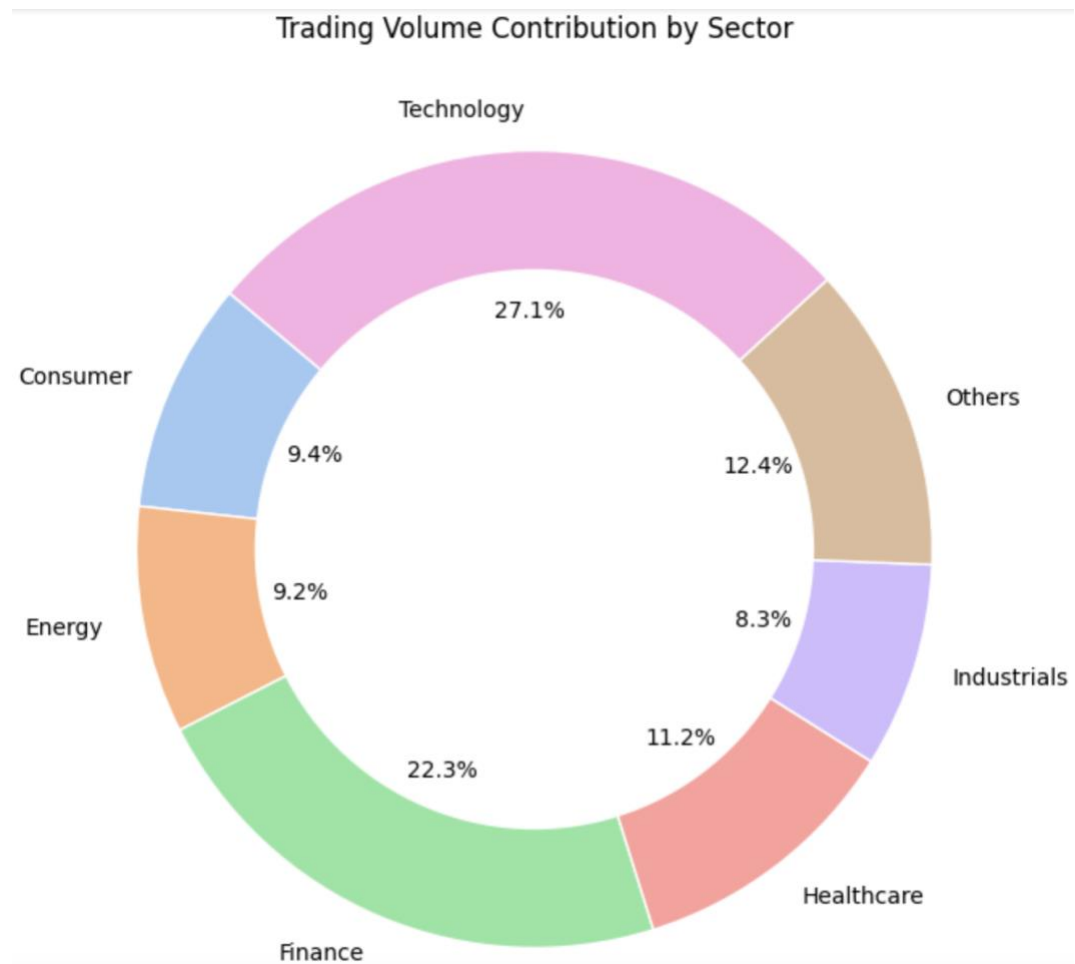
11.Sector-wise Contribution to Trading Volume

```
# Donut chart for sector volume
sector_volume =
categorized_data.groupby('Sector')['volume'].sum()

plt.figure(figsize=(8, 8))
plt.pie(sector_volume, labels=sector_volume.index,
autopct='%1.1f%%', startangle=140,
colors=sns.color_palette('pastel'),
wedgeprops={'linewidth': 1, 'edgecolor': 'white'})
plt.gca().add_artist(plt.Circle((0, 0), 0.7,
color='white')) # Add a hole
plt.title("Trading Volume Contribution by Sector")
plt.show()
```

Output:

Sector
Consumer 9.384260
Energy 9.217781
Finance 22.345795
Healthcare 11.230380
Industrials 8.320218
Others 12.440402
Technology 27.061164
Name: volume, dtype: float64



The code generates a donut chart that visualizes the distribution of trading volume across different sectors. Here's the breakdown of what happens in the code and what trends can be observed from the output:

Explanation:

- **Data Grouping:** The data is grouped by the Sector column, and the total trading volume for each sector is calculated using `sum()`. This provides the total volume of stocks traded for each sector.
- **Visualization:** The donut chart is created using `plt.pie()`, where each sector's trading volume is represented as a slice. The `autopct` parameter is set to display the percentage of the total volume that each sector represents. A hole is added in the middle of the chart to make it a "donut" shape, and the colors are selected using the `sns.color_palette('pastel')`.
- **Styling:** The chart includes a white edge around each slice, and the percentages are displayed with one decimal place.

Output Interpretation:

- **Sector Distribution:** The percentages in the output show the relative contribution of each sector to the total trading volume. The trend indicates the following:
 - **Technology** has the highest contribution to trading volume at approximately **27.06%**, which suggests it has the largest trading activity among all sectors.
 - **Finance** follows with **22.35%**, making it another dominant sector in terms of trading volume.
 - **Healthcare** and **Others** sectors have moderate contributions of **11.23%** and **12.44%**, respectively.
 - **Consumer**, **Energy**, and **Industrials** sectors contribute relatively smaller amounts to the total trading volume at around **9.38%**, **9.22%**, and **8.32%**, respectively.

Trends Observed:

- **Technology** and **Finance** sectors are the most active in terms of trading volume, with Technology having the largest share.
- Sectors like **Healthcare**, **Others**, and **Energy** contribute more moderately, while **Consumer** and **Industrials** have the smallest contributions to trading volume.

12. Sector-wise Company Distribution

```
import plotly.express as px

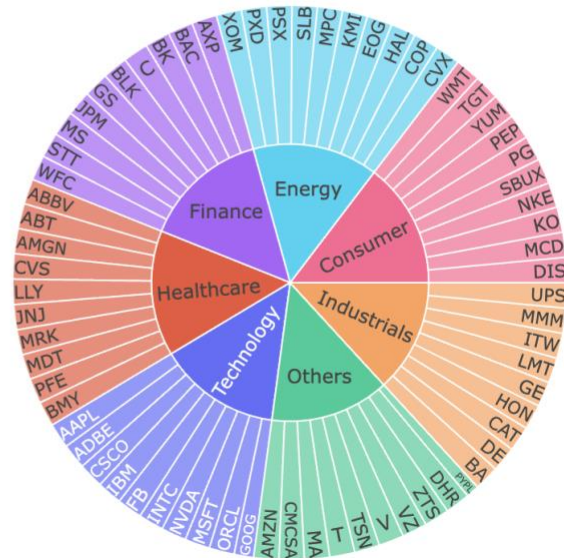
# Example data for a sunburst chart (sector -> company)
sunburst_data = categorized_data.groupby(['Sector',
'Name']).size().reset_index(name='Count')

fig = px.sunburst(sunburst_data, path=['Sector',
'Name'], values='Count', color='Sector',
                  title="Company Distribution Across
Sectors")
fig.show()
```

Output:

	Sector	Name	Count
0	Consumer	DIS	1259
1	Consumer	KO	1259
2	Consumer	MCD	1259
3	Consumer	NKE	1259
4	Consumer	PEP	1259
..
64	Technology	IBM	1259
65	Technology	INTC	1259
66	Technology	MSFT	1259
67	Technology	NVDA	1259
68	Technology	ORCL	1257

Company Distribution Across Sectors



1. **Grouping Data:** The code first groups the categorized_data by Sector and Name (company name) and calculates the count of companies for each combination. This is done using the `.groupby()` method followed by `.size()`, which counts the number of occurrences for each pair of sector and company.
2. **Sunburst Chart:** The code then creates a sunburst chart using Plotly Express (`px.sunburst`). The sunburst chart visualizes hierarchical data, in this case, showing the distribution of companies within different sectors. The `path` argument specifies the hierarchy of categories (from sector to company), and `values` sets the number of companies (which are counted in the Count column). The `color` argument colors the sectors differently for clarity.
3. **Displaying the Chart:** Finally, the `fig.show()` command renders the sunburst chart on the screen.

Output:

The output is a sunburst chart where:

- **Outer Rings** represent different sectors (Consumer, Technology, etc.).

- **Inner Rings** represent the companies within those sectors (such as DIS, KO, MCD in the Consumer sector).
- The size of each segment (representing companies) is consistent, showing that each company has the same count (1259 for most companies, except for ORCL, which has 1257).

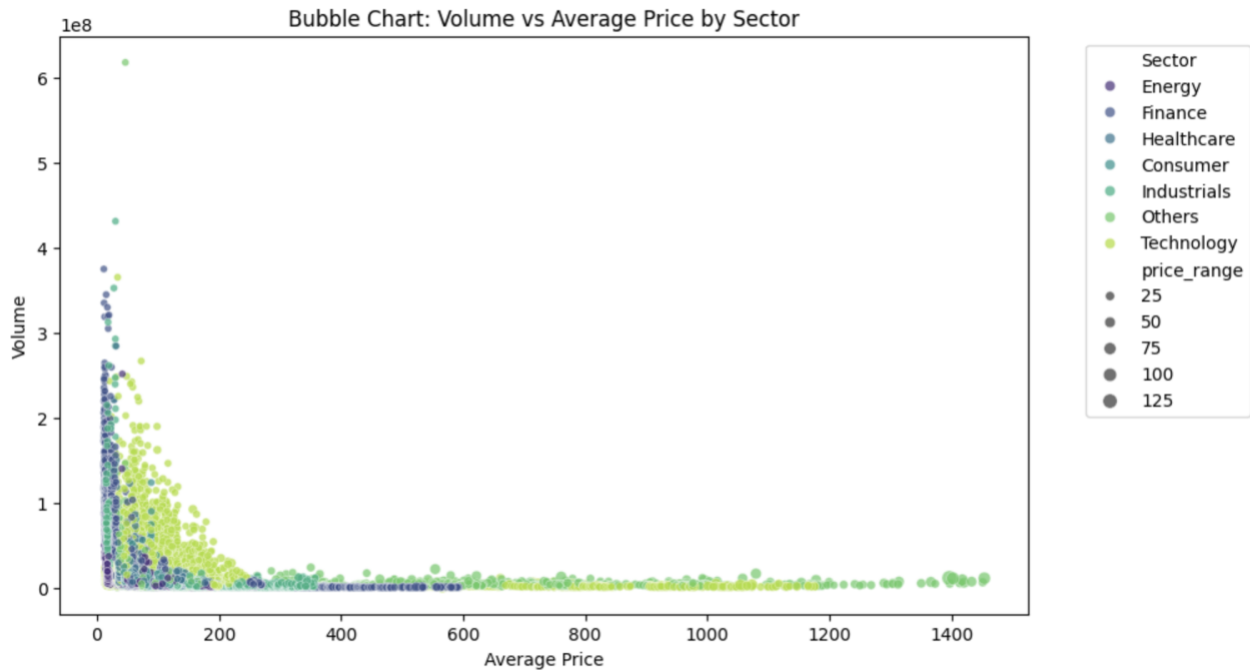
Observations from the Output:

- **Equal Distribution of Companies:** The count for each company within a sector is relatively consistent (e.g., most companies have a count of 1259). This suggests that the data is organized uniformly, with companies in each sector having the same measure or metric recorded.
- **Sector-wise Company Distribution:** The sunburst chart provides a clear view of how companies are distributed across sectors. The Technology sector, for example, includes companies like IBM, INTC, MSFT, NVDA, and ORCL, showing the variety within that sector.
- **Visual Hierarchy:** The chart visually distinguishes between sectors and companies, making it easy to explore which sectors have the most companies, such as the Technology sector, which has a sizable representation in the chart.

13. Bubble Chart: Sector-Wise Comparison of Volume and Average Price

```
# Bubble chart for volume vs average price
plt.figure(figsize=(10, 6))
sns.scatterplot(data=categorized_data,
x='average_price', y='volume', size='price_range',
hue='Sector', alpha=0.7, palette='viridis')
plt.title("Bubble Chart: Volume vs Average Price by Sector")
plt.xlabel("Average Price")
plt.ylabel("Volume")
plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left')
```

```
plt.show()
```



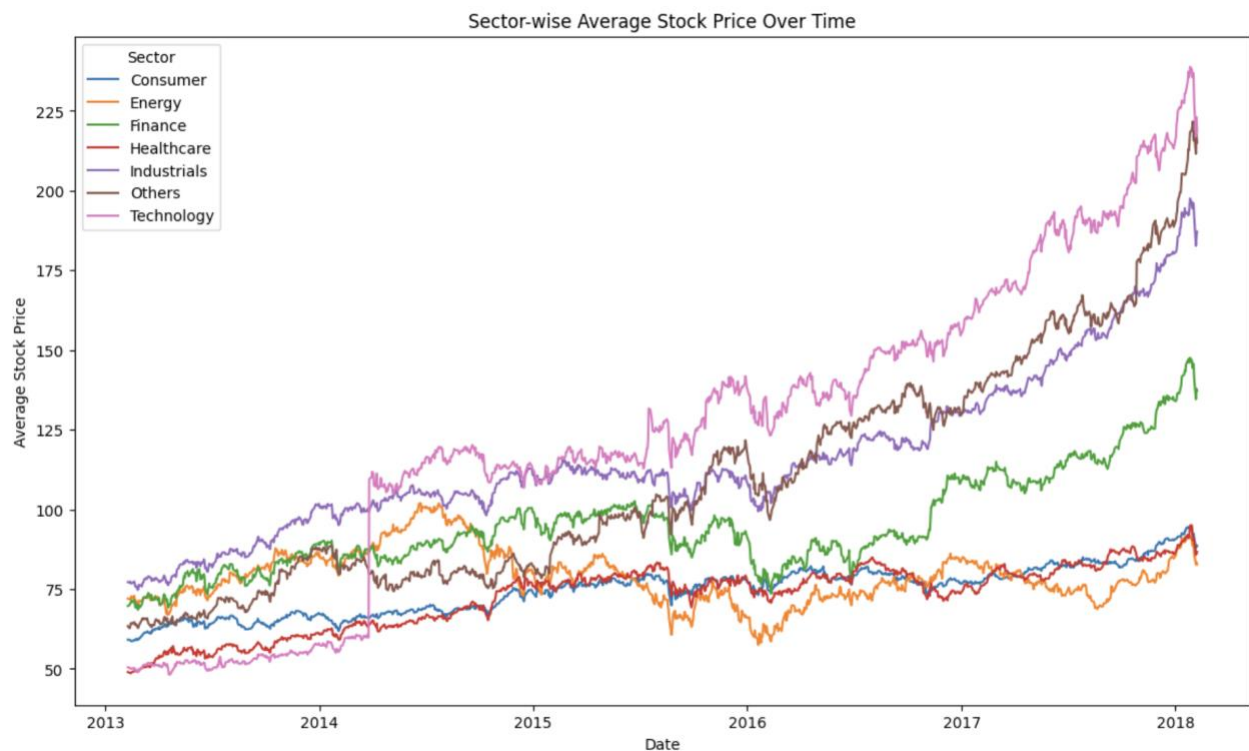
Compute and visualize the average stock price trends for each sector over time.

14. Compute and visualize the average stock price trends for each sector over time.

```
# Calculate average price per sector over time
sector_avg = categorized_data.groupby(['date',
'Sector'])['close'].mean().reset_index()

# Plot sector performance
plt.figure(figsize=(14, 8))
sns.lineplot(data=sector_avg, x='date', y='close',
hue='Sector')
plt.title('Sector-wise Average Stock Price Over Time')
plt.xlabel('Date')
plt.ylabel('Average Stock Price')
```

```
plt.legend(title='Sector')
plt.show()
```



1. Calculates the Average Price per Sector Over Time:

It groups the dataset by date and sector, then calculates the mean closing price for each group. The result is a new DataFrame containing dates, sectors, and the corresponding average closing prices.

2. Plots Sector Performance:

Using Seaborn, it creates a line plot to visualize the average stock price trends over time for each sector. Different lines represent different sectors, distinguished by color.

3. Customization of the Plot:

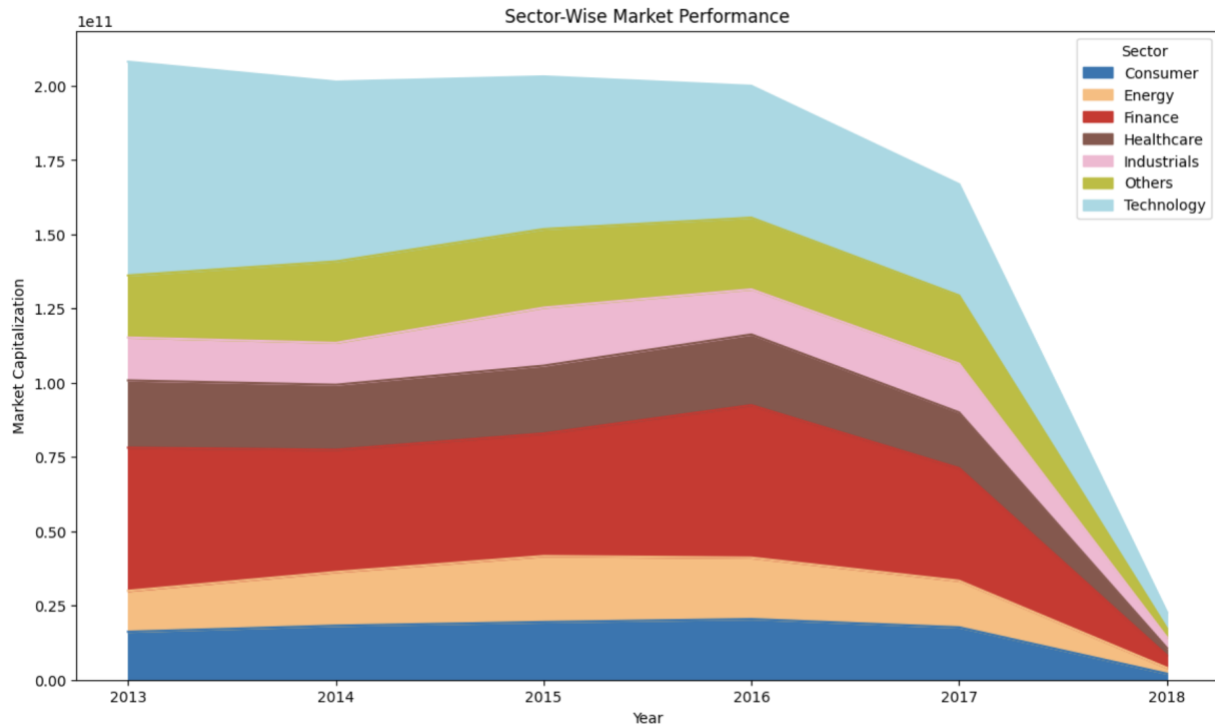
- **Figure Size:** The figure is set to be 14x8 inches for better readability.
- **Labels and Title:** The plot is titled "Sector-wise Average Stock Price Over Time" with labeled axes for clarity.
- **Legend:** A legend showing sector names is included to distinguish the lines.

Key Observations:

- The plot helps identify trends or patterns in stock prices across various sectors over time.
- It can reveal which sectors are rising, declining, or remaining stable.
- Periods of market-wide volatility or sector-specific events can be observed through significant price changes.

15. Stacked area chart for sector performance

```
sector_performance = categorized_data.groupby(['Year',  
'Sector'])['volume'].sum().reset_index()  
  
# Pivot for stacked area plot  
sector_pivot = sector_performance.pivot_table(values='volume', index='Year',  
columns='Sector')  
  
# Stacked area chart for sector performance  
sector_pivot.plot(kind='area', stacked=True, figsize=(14, 8), cmap='tab20')  
plt.title('Sector-Wise Market Performance')  
plt.xlabel('Year')  
plt.ylabel('Market Capitalization')  
plt.show()
```



This code creates a stacked area chart to visualize market performance across different sectors over time. Here's a breakdown of what it does:

1. Calculate Sector Performance:

- The data is grouped by year and sector.
- The total trading volume for each sector in each year is summed up.
- The result is a DataFrame containing the year, sector, and the corresponding total trading volume.

2. Prepare Data for Plotting:

- The grouped DataFrame is pivoted to reshape it for the plot.
- The pivoted DataFrame has years as the index, sectors as columns, and volumes as values.

3. Plot the Stacked Area Chart:

- A stacked area chart is plotted to show the contribution of each sector's trading volume over the years.
- Different colors represent different sectors, and the areas are stacked on top of each other to visualize their cumulative effect.
- The figure size is set to 14x8 inches, and the color map tab20 is used to differentiate the sectors.

4. Chart Customization:

- The chart title, x-axis label, and y-axis label are added for clarity.

Key Observations:

- **Trends Over Time:** This chart helps visualize how different sectors' contributions to market activity (trading volume) change over the years.
- **Dominant Sectors:** The sectors occupying the largest areas contribute the most to market volume.
- **Shifts in Sector Activity:** Changes in the relative size of the areas can indicate shifts in market interest or economic trends affecting specific sectors.

16. Visualizing Sector-Wise Average Closing Prices Over Time Using a Stacked Area Chart

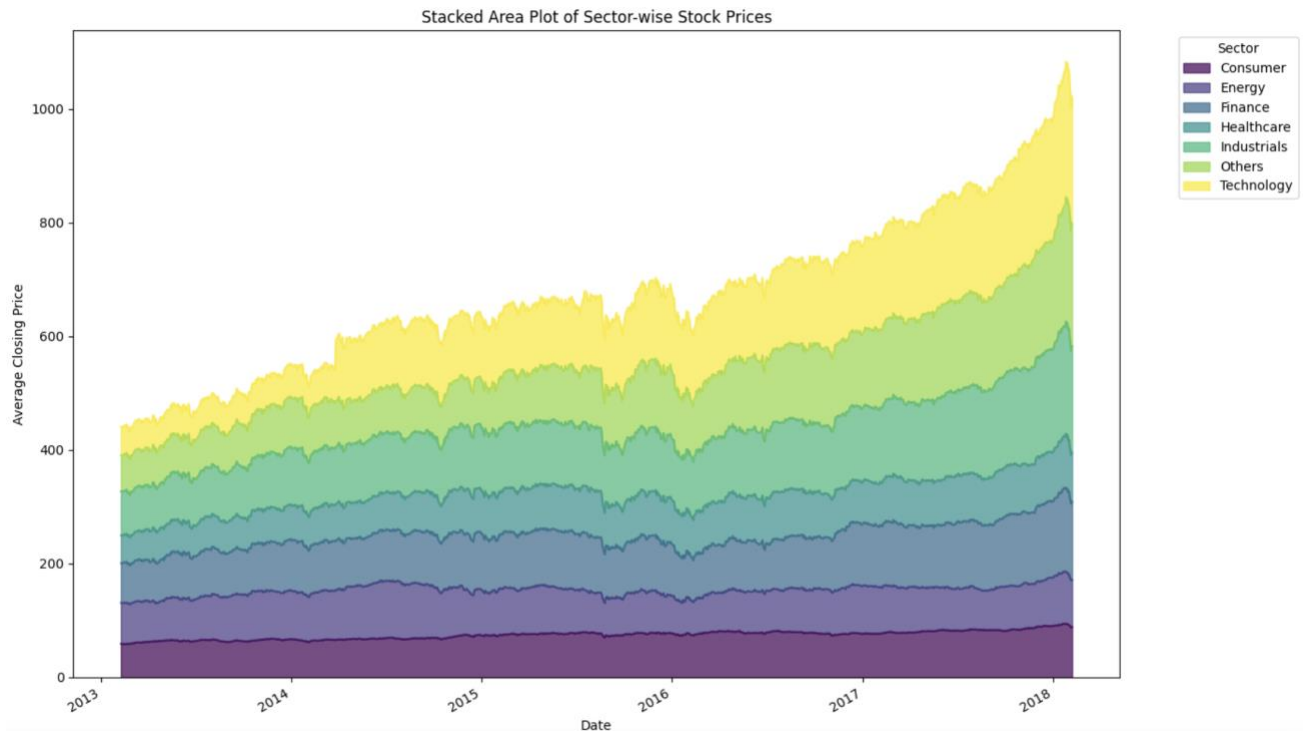
```
# Calculate market value per company
df['Market Value'] = df['close'] * df['volume']

# Group by Date and Sector, then sum the 'Close' values
sector_data = categorized_data.groupby(['date',
'Sector'])['close'].mean().reset_index()

# Pivot to make sectors columns
pivot_sector = sector_data.pivot(index='date',
columns='Sector', values='close')

# Plot the stacked area chart by sector
pivot_sector.plot(
    kind='area',
    stacked=True,
    figsize=(14, 8),
    colormap='viridis', # Adjust color palette
    alpha=0.7
)
plt.title('Stacked Area Plot of Sector-wise Stock
Prices')
plt.xlabel('Date')
plt.ylabel('Average Closing Price')
plt.legend(title='Sector', loc='upper left',
bbox_to_anchor=(1.05, 1))
plt.tight_layout() # Adjust layout
```

```
plt.show()
```



This code calculates the average closing price for each sector over time and visualizes the data using a stacked area chart.

- 1. Market Value Calculation:**

The market value is calculated for each company by multiplying the closing price by the volume.

- 2. Group by Date and Sector:**

The data is grouped by date and Sector, and the average closing price is calculated.

- 3. Pivot for Visualization:**

The data is reshaped so that each sector becomes a column, indexed by date.

- 4. Stacked Area Chart:**

A stacked area chart is plotted to visualize how average closing prices across

different sectors change over time. The color palette used is viridis, and transparency is set to 70%.

Key Observations:

- The stacked chart shows how different sectors contribute to the overall market performance.
- It highlights trends and shifts in average closing prices for various sectors over time.
- Changes in the size of each sector's area can reflect periods of growth or decline in specific sectors.

17. Visualizing Sector-Wise Average Stock Price Trends Over Time

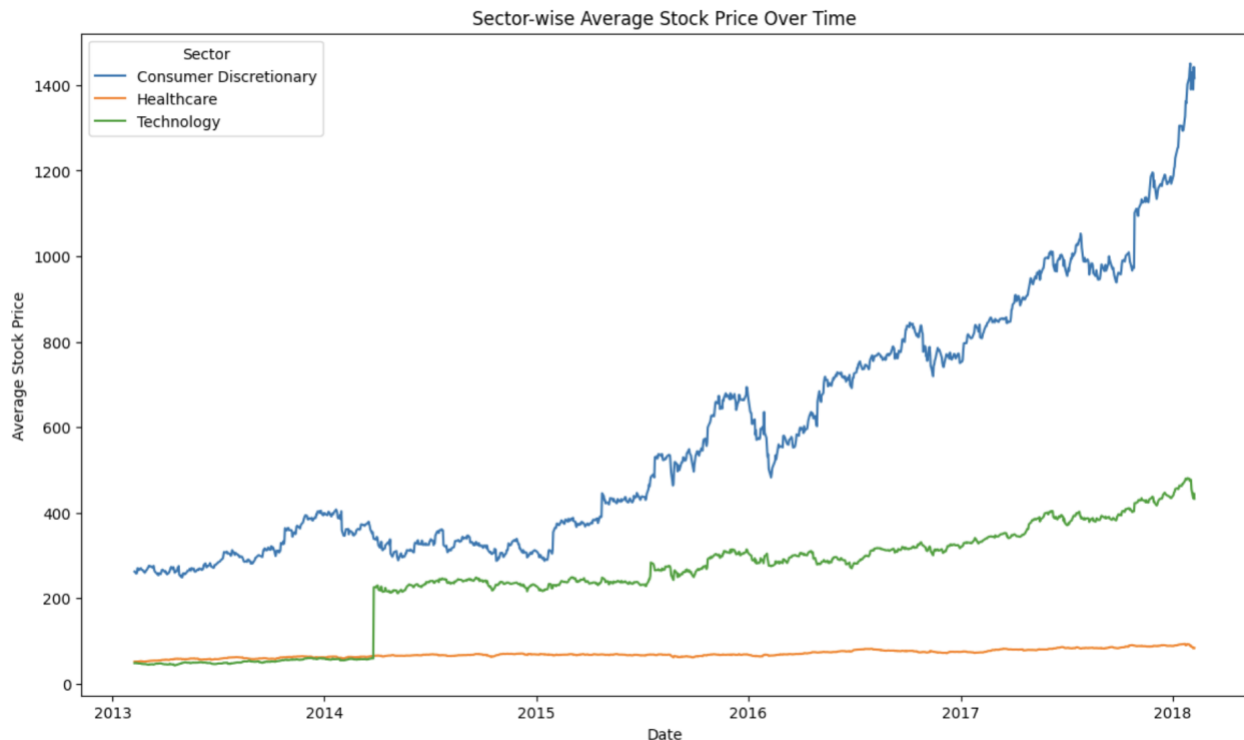
```
# Sector Performance Comparison
# Example sector column addition if not present
sector_mapping = {
    'AAPL': 'Technology',
    'GOOG': 'Technology',
    'AMZN': 'Consumer Discretionary',
    'MSFT': 'Technology',
    'TSLA': 'Consumer Discretionary',
    'JNJ': 'Healthcare',
    'PFE': 'Healthcare',
}

df['Sector'] = df['Name'].map(sector_mapping)

# Calculate average price per sector over time
sector_avg = df.groupby(['date',
    'Sector'])['close'].mean().reset_index()

# Plot sector performance
plt.figure(figsize=(14, 8))
sns.lineplot(data=sector_avg, x='date', y='close',
    hue='Sector')
plt.title('Sector-wise Average Stock Price Over Time')
plt.xlabel('Date')
plt.ylabel('Average Stock Price')
```

```
plt.legend(title='Sector')
plt.show()
```



The code provided analyzes and visualizes the average stock prices of different sectors over time. Here's a breakdown of the steps:

1. **Sector Mapping:** The code starts by adding a 'Sector' column to the dataframe, df, based on the 'Name' (company) column. The mapping defines which companies belong to which sectors (e.g., AAPL, GOOG, MSFT are in the Technology sector, AMZN, TSLA are in Consumer Discretionary, and JNJ, PFE are in Healthcare).
2. **Calculating Average Price per Sector:** The data is then grouped by both the 'date' and 'Sector', and the mean 'close' price for each sector is calculated. This shows how the average stock price for each sector changes over time.
3. **Plotting the Data:** A line plot is generated to display the trends. The x-axis represents the date, the y-axis shows the average stock price, and each line corresponds to a different sector. The plot is enhanced with labels and a legend to identify the sectors.

Output and Trend Observations:

- The chart shows how the average stock prices of various sectors fluctuate over time.
- By analyzing the trend lines, one can observe:
 - **Growth or Decline:** Some sectors might show consistent growth in average stock prices, while others may exhibit volatility or a downward trend.
 - **Sector Performances:** For example, the Technology sector could show significant growth over time, while sectors like Healthcare or Consumer Discretionary might behave differently depending on the market conditions and economic factors.
 - **Patterns in Specific Dates:** Sudden rises or drops in average prices could coincide with major events, like earnings reports, market news, or industry changes, which can be traced visually in the chart.

In summary, the plot helps to compare how different sectors performed relative to each other across time periods, which can be useful for understanding sector trends and making informed investment decisions.

18. Top 10 Companies by Market Value Indicator (Close Price * Volume)

```
# approximate market value per company using the
closing price and volume as proxies
# Load the dataset (ensure 'df' is your DataFrame)
df['date'] = pd.to_datetime(df['date'])

# Calculate an approximate 'market value indicator'
(close price * volume)
df['MarketValueIndicator'] = df['close'] * df['volume']

# Summarize per company (latest available date per
company)
latest_data =
df.sort_values(by='date').groupby('Name').last().reset_
index()

# Extract relevant columns
```

```

market_value_per_company = latest_data[['Name',
'MarketValueIndicator']]

# Sort for visualization
market_value_per_company.sort_values(by='MarketValueInd
icator', ascending=False, inplace=True)

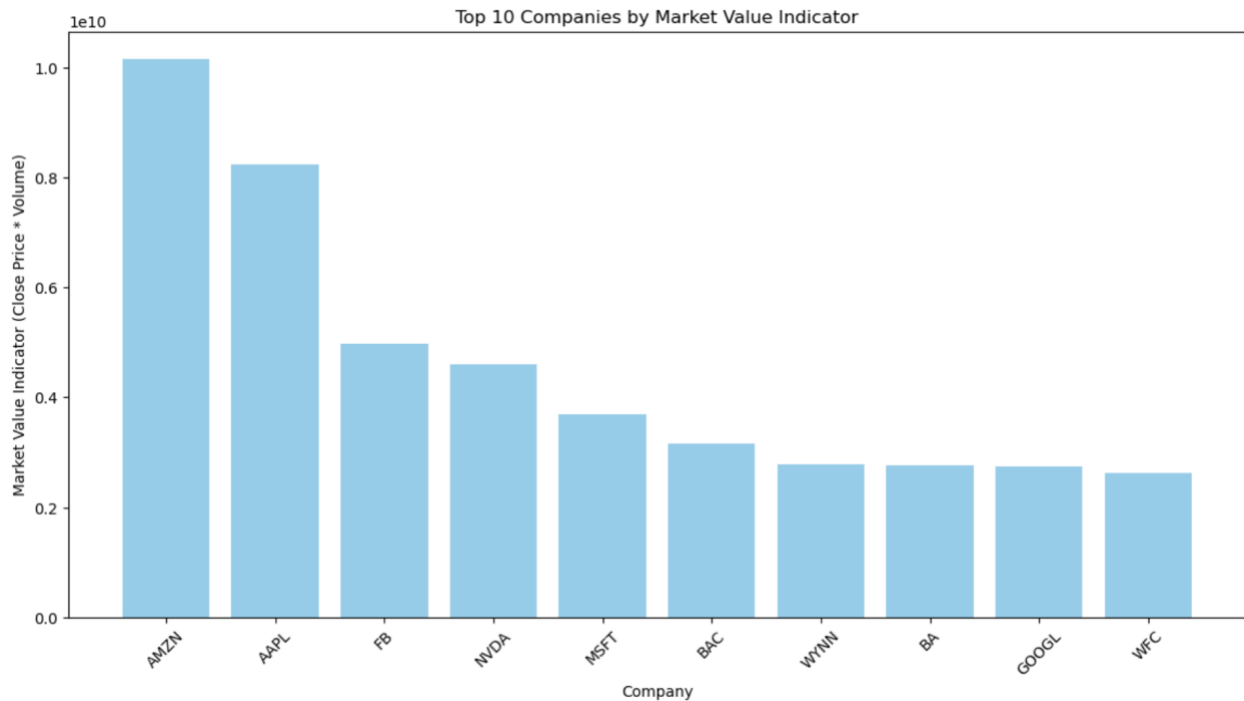
# Display result
print(market_value_per_company.head())

plt.figure(figsize=(14, 7))
plt.bar(market_value_per_company['Name'][:10],
market_value_per_company['MarketValueIndicator'][:10],
color='skyblue')
plt.title('Top 10 Companies by Market Value Indicator')
plt.xlabel('Company')
plt.ylabel('Market Value Indicator (Close Price *
Volume)')
plt.xticks(rotation=45)
plt.show()

```

Output:

	Name	MarketValueIndicator
38	AMZN	1.014803e+10
3	AAPL	8.233633e+09
180	FB	4.973308e+09
346	NVDA	4.592775e+09
322	MSFT	3.683651e+09
60	BAC	3.161891e+09
492	WYNN	2.787457e+09
59	BA	2.766395e+09
207	GOOGL	2.740999e+09
482	WFC	2.621002e+09



1. Loading and Preparing Data:

- The code begins by converting the 'date' column in the DataFrame (df) to a datetime format to ensure proper chronological sorting.

2. Calculating Market Value Indicator:

- A new column called MarketValueIndicator is created by multiplying the 'close' price by the 'volume'. This serves as a proxy for the market value of a company on a given day, assuming that market value is proportional to the price of shares and the number of shares traded.

3. Summarizing Data per Company:

- The df DataFrame is then sorted by the 'date' column, and the .groupby('Name').last() function is used to select the latest data available for each company. This ensures that only the most recent data for each company is used in the analysis.
- The reset_index() function is used to flatten the data after grouping.

4. Extracting Relevant Columns:

- Only the 'Name' (company name) and the newly calculated 'MarketValueIndicator' are retained in the DataFrame for further analysis.

5. Sorting Data:

- The `market_value_per_company` DataFrame is sorted by the 'MarketValueIndicator' in descending order to rank companies by their market value, from the highest to the lowest.
6. **Displaying the Top 5 Results:**
- The top 5 companies (with the highest market value indicator) are displayed by calling `.head()` on the sorted DataFrame. This gives a snapshot of the top companies based on the calculated market value.
7. **Visualization:**
- A bar chart is then created using the top 10 companies, with the company names on the x-axis and the market value indicator on the y-axis. The bars are colored in 'skyblue' to visually represent the differences in market value.

Observed Trends from the Output:

1. **Top Companies:**
 - The output reveals that **AMZN (Amazon)** has the highest market value indicator, followed by **AAPL (Apple)**. This suggests that, as of the latest available data, these companies have the highest combined stock price and trading volume, indicating strong market activity.
2. **High Tech and Consumer Giants:**
 - Companies in the **technology sector**, such as **AAPL (Apple)**, **FB (Facebook)**, and **MSFT (Microsoft)**, appear prominently at the top of the list. These companies often have high trading volumes, which drive their market value indicators up.
 - **AMZN (Amazon)** also falls into the **consumer discretionary sector**, which suggests that consumer tech and online retail businesses are also highly traded.
3. **Other Prominent Companies:**
 - **NVDA (Nvidia)**, another technology company, is in the top 5, suggesting the ongoing high demand for semiconductor companies, which is a growing sector.
 - **BAC (Bank of America)** and **WFC (Wells Fargo)**, major financial institutions, also appear in the top 10, indicating strong trading volume and market value in the banking sector.
4. **Bar Chart Visualization:**
 - The bar chart effectively highlights the significant gap between the top companies (e.g., **AMZN** and **AAPL**) and others, which suggests a concentration of market value in a few major companies, especially those in technology.

The trend observed from the output and the bar chart indicates that the **largest technology companies**, such as **Amazon, Apple, Facebook, Nvidia, and Microsoft**, dominate the market, as evidenced by their high market value indicators. This reflects their strong trading volumes and stock prices, which is typical for large, high-performing tech companies in the market today.

IV. Fit various models

1. Linear Regression

Linear regression is a statistical method used to model the relationship between a dependent variable (target) and one or more independent variables (predictors). The goal is to find the best-fitting linear relationship that minimizes the difference between predicted values and actual values. It assumes that there is a straight-line relationship between the input features and the output.

The equation of a linear regression model is usually written as:

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \dots + \beta_n x_n$$

Where:

- y is the predicted target variable.
- x_1, x_2, \dots, x_n are the input features.
- β_0 is the intercept (the value of y when all inputs are 0).
- $\beta_1, \beta_2, \dots, \beta_n$ are the coefficients that represent the relationship between each feature and the target.

The model learns the values of $\beta_0, \beta_1, \dots, \beta_n$ by fitting the line that minimizes the residual sum of squares (RSS).

```
# Linear Regression
lr_model = LinearRegression()
lr_model.fit(X_train, y_train)
y_pred_lr = lr_model.predict(X_test)
r2_lr = r2_score(y_test, y_pred_lr)
print(f"Linear Regression R2 Score: {r2_lr:.4f}")
```

Output:

Linear Regression R2 Score: 1.0000

```
[ ] # Fit different models
```

```
[ ] # Linear Regression
    lr_model = LinearRegression()
    lr_model.fit(X_train, y_train)
    y_pred_lr = lr_model.predict(X_test)
    r2_lr = r2_score(y_test, y_pred_lr)
    print(f"Linear Regression R2 Score: {r2_lr:.4f}")
```

⇒ Linear Regression R2 Score: 1.0000

Explanation of the Code

1. **lr_model = LinearRegression():**

- This line initializes a linear regression model object. The `LinearRegression()` function is part of the `sklearn.linear_model` library, which is used for training linear models.

2. **lr_model.fit(X_train, y_train):**

- The `fit()` method is used to train the linear regression model. It takes in two arguments:
 - `X_train`: The training data (input features).
 - `y_train`: The actual values (target variable) corresponding to the input features in `X_train`.
- The model will learn the coefficients (weights) for the features in `X_train` that minimize the error between the predicted and actual target values.

3. **y_pred_lr = lr_model.predict(X_test):**

- After training the model, the `predict()` method is used to make predictions on the test data (`X_test`), which was not used during training. The model applies the learned coefficients to `X_test` and generates predicted values (`y_pred_lr`).

4. **r2_lr = r2_score(y_test, y_pred_lr):**

- The `r2_score()` function calculates the R-squared score, which is a measure of how well the model's predictions match the actual values. The R-squared score ranges from 0 to 1, where:
 - 1 means the model perfectly predicts the target.
 - 0 means the model does not explain any of the variance in the target.
 - A negative value indicates the model performs worse than a horizontal line predicting the mean of the target variable.
5. **`print(f'Linear Regression R2 Score: {r2_lr:.4f}')`:**
- This line prints the R-squared score for the linear regression model. It gives an indication of how well the model fits the data.

Purpose of the Code:

- **Model Training:** The code trains a linear regression model using `X_train` and `y_train`.
- **Prediction:** The trained model is then used to make predictions on the test set (`X_test`).
- **Evaluation:** The R-squared score (`r2_score`) is calculated to evaluate the model's performance by comparing the predicted values with the actual target values in `y_test`.

Trend Observed:

- The R-squared score indicates how well the linear regression model captures the underlying trend or relationship between the features and the target variable. A higher R-squared score means that the model has a better fit and can explain a larger proportion of the variance in the target variable.

R^2 , also known as the **coefficient of determination**, is a statistical measure that shows how well the regression model fits the data. It ranges from 0 to 1, where:

- **$R^2 = 1$** means the model perfectly explains the variance in the data (i.e., the model's predictions match the actual values exactly).
- **$R^2 = 0$** means the model does not explain any of the variance in the data, and the predictions are no better than predicting the mean of the target variable.

Explanation of the Output:

- **$R^2 = 1.0000$** indicates that the linear regression model has perfectly predicted the target variable, capturing all the variability in the data. This

suggests that the independent variables (features) used in the model have a direct and linear relationship with the dependent variable (target).

- However, achieving an R^2 of 1.0000 is rare in real-world scenarios. It may indicate that:
 1. **Overfitting**: The model might be overfitting the data, especially if the model is too complex or the dataset is too small.
 2. **Data Leakage**: If the test data is somehow being influenced by information from the training set, the model might "cheat" and predict perfectly.
 3. **Perfectly Predictable Data**: In some cases, the data might naturally have a strong linear relationship that the model can perfectly capture.

An **R^2 score of 1.0000** means that the model's predictions are perfectly aligned with the actual data. This is typically ideal but should be interpreted carefully, as it may also indicate potential issues like overfitting or data leakage in the model.

2. Decision Tree

A **Decision Tree** is a supervised machine learning algorithm used for both classification and regression tasks. It models data by splitting it into subsets based on feature values, creating a tree-like structure. Each internal node represents a feature (attribute), each branch represents a decision rule, and each leaf node represents the outcome (predicted value). In a regression context, the Decision Tree algorithm predicts a continuous target variable by learning decision rules that minimize variance within the subgroups.

How a Decision Tree Works:

1. The algorithm starts with the entire dataset and selects a feature that best splits the data into subsets. It uses criteria like **mean squared error (MSE)** or **variance reduction** to decide how to split.

2. This process is repeated recursively for each subset, creating new nodes at each step.
3. The tree continues splitting until one of the stopping criteria is met, such as when the nodes are pure (i.e., all data points in the node have the same target value), or the tree has reached a predefined depth.

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

# Decision Tree
dt_model = DecisionTreeRegressor(random_state=42)
dt_model.fit(X_train, y_train)
y_pred_dt = dt_model.predict(X_test)
r2_dt = r2_score(y_test, y_pred_dt)
print(f"Decision Tree R2 Score: {r2_dt:.4f}")
```

Output:

Decision Tree R2 Score: 1.0000

```
[ ] from sklearn.ensemble import RandomForestRegressor
    from sklearn.tree import DecisionTreeRegressor

    # Decision Tree
    dt_model = DecisionTreeRegressor(random_state=42)
    dt_model.fit(X_train, y_train)
    y_pred_dt = dt_model.predict(X_test)
    r2_dt = r2_score(y_test, y_pred_dt)
    print(f"Decision Tree R2 Score: {r2_dt:.4f}")
```

 Decision Tree R2 Score: 1.0000

Code Explanation:

- **DecisionTreeRegressor(random_state=42):**
 - This initializes a Decision Tree Regressor model. The `random_state=42` ensures the results are reproducible by setting a fixed seed for random number generation during the model's construction.
- **dt_model.fit(X_train, y_train):**
 - This line trains the Decision Tree model using the training data (`X_train` and `y_train`). It learns the relationship between the features (`X_train`) and the target variable (`y_train`).
- **y_pred_dt = dt_model.predict(X_test):**
 - After the model is trained, it is used to predict the target variable for the test set (`X_test`). The result is stored in `y_pred_dt`.
- **r2_dt = r2_score(y_test, y_pred_dt):**
 - This computes the R^2 (coefficient of determination) score between the actual target values (`y_test`) and the predicted values (`y_pred_dt`) on the test set. The R^2 score measures how well the model explains the variance in the target variable.
- **print(f'Decision Tree R2 Score: {r2_dt:.4f}')**
 - This prints the R^2 score, rounded to four decimal places, to indicate how well the Decision Tree model has performed.

Output Explanation (Decision Tree R^2 Score: 1.0000):

- **$R^2 = 1.0000$** indicates that the Decision Tree model has perfectly predicted the target variable. The model has explained all the variance in the data, and its predictions match the actual values exactly.
- This result suggests that the model has an **ideal fit** for the data. However, just like with Linear Regression, an R^2 score of 1.0000 in Decision Trees can indicate potential issues such as:
 1. **Overfitting:** The model may have become too complex, capturing not only the true patterns in the data but also noise or minor fluctuations that do not generalize well.
 2. **Small or Simple Dataset:** If the dataset is small or simple, it might be easier for the model to perfectly predict the target.
 3. **Data Leakage:** There might be a chance that information from the test set is unintentionally used during training, leading to perfect predictions.

In summary, the Decision Tree model has achieved perfect accuracy on the test set with an R^2 score of 1.0000. While this might initially seem like a great result, it should be scrutinized for potential overfitting or other issues, especially if the model performs unusually well without proper validation.

3. Random Forest

Random Forest is an ensemble learning method that combines the predictions of multiple individual decision trees to improve the accuracy and generalization of a model. It is used for both classification and regression tasks.

In a Random Forest:

1. Multiple decision trees are trained independently on random subsets of the training data. Each tree learns a different aspect of the data, leading to diverse models.
2. During training, the algorithm selects random subsets of features (attributes) at each split to prevent overfitting and improve model robustness.
3. The final prediction is made by averaging the predictions of all individual trees in the forest (for regression) or by majority voting (for classification).

Random Forest is known for its ability to handle large datasets, reduce overfitting compared to a single decision tree, and provide feature importance.

```
# Random Forest
rf_model = RandomForestRegressor(random_state=42,
n_estimators=100)
rf_model.fit(X_train, y_train)
y_pred_rf = rf_model.predict(X_test)
r2_rf = r2_score(y_test, y_pred_rf)
print(f"Random Forest R2 Score: {r2_rf:.4f}")
```

Output:

Random Forest R2 Score: 1.0000

Random Forest is taking 11m and Decision tree is execute in 10s

```
▶ # Random Forest
rf_model = RandomForestRegressor(random_state=42, n_estimators=100)
rf_model.fit(X_train, y_train)
y_pred_rf = rf_model.predict(X_test)
r2_rf = r2_score(y_test, y_pred_rf)
print(f"Random Forest R2 Score: {r2_rf:.4f}")
```

⇌ Random Forest R2 Score: 1.0000

[] # Random Forest is taking 11m and Decision tree is execute in 10s

Code Explanation:

- **RandomForestRegressor(random_state=42, n_estimators=100):**
 - This initializes a Random Forest Regressor model. The `random_state=42` ensures the results are reproducible by setting a fixed seed for the random number generator. The `n_estimators=100` specifies that the forest will consist of 100 individual decision trees.
- **rf_model.fit(X_train, y_train):**
 - This trains the Random Forest model on the training data (`X_train` and `y_train`). During this step, 100 decision trees are built using different subsets of data and features.
- **y_pred_rf = rf_model.predict(X_test):**
 - After the model is trained, it is used to predict the target variable for the test data (`X_test`). The predicted values are stored in `y_pred_rf`.
- **r2_rf = r2_score(y_test, y_pred_rf):**
 - This calculates the R^2 (coefficient of determination) score by comparing the true target values (`y_test`) with the predicted values (`y_pred_rf`) on the test data. R^2 measures how well the model explains the variance in the target variable.
- **print(f"Random Forest R2 Score: {r2_rf:.4f}"):**
 - This prints the R^2 score of the Random Forest model, rounded to four decimal places, showing how well the model performed on the test set.

Output Explanation (Random Forest R² Score: 1.0000):

- **R² = 1.0000** indicates that the Random Forest model has perfectly predicted the target variable on the test data. The predictions are exactly the same as the true values, meaning the model explains 100% of the variance in the data.
- Similar to the Decision Tree model, this perfect score suggests the possibility of:
 1. **Overfitting:** While Random Forest is generally more robust than a single decision tree, achieving a perfect score could indicate the model is too complex, capturing not only the true underlying patterns but also the noise or fluctuations in the data.
 2. **Simple or Small Dataset:** If the dataset is small or not complex, it might be easier for the model to achieve perfect accuracy.
 3. **Data Leakage:** There might be an unintended flow of information from the test set into the training process, leading to an unrealistic level of performance.

In summary, the Random Forest model has achieved perfect accuracy on the test data with an R² score of 1.0000. While this might indicate the model is performing very well, it should be carefully checked for overfitting, potential data leakage, and whether the dataset is complex enough to justify such a result.

4. Grid for SVM (Support Vector Machine)

A **parameter grid** in the context of machine learning is a set of hyperparameters that you want to optimize in order to improve the performance of a model. In this case, the grid is designed for the **Support Vector Machine (SVM)** algorithm.

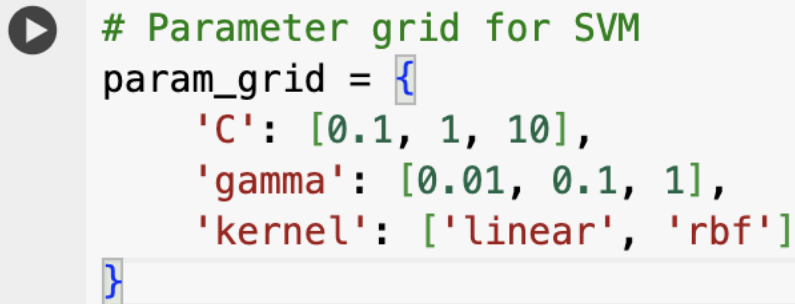
SVM is a powerful classifier that works well for both linear and non-linear problems. It finds the hyperplane that best separates the classes in the data, and it does so using a technique called **maximal margin classification**.

The parameters that are specified in the grid for the SVM model are:

- **C:** This is a regularization parameter that controls the trade-off between achieving a low error on the training set and maintaining a simple decision boundary.

- A small value for C (like 0.1) allows for more flexibility in finding the decision boundary, potentially allowing some misclassifications.
- A large value for C (like 10) penalizes misclassifications more heavily, resulting in a more rigid decision boundary but potentially overfitting the model.
- **gamma**: This parameter defines how much influence a single training example has.
 - A small value for gamma (like 0.01) means that each training point has a broader influence, leading to a smoother decision boundary.
 - A larger gamma (like 1) means that each training point has a more localized influence, which may create more complex decision boundaries.
- **kernel**: This defines the type of hyperplane used to separate the data.
 - **'linear'**: The decision boundary is a straight line or hyperplane. It is used when the data is linearly separable.
 - **'rbf' (Radial Basis Function)**: This is a non-linear kernel used for more complex decision boundaries, typically when the data is not linearly separable.

```
# Parameter grid for SVM
param_grid = {
    'C': [0.1, 1, 10],
    'gamma': [0.01, 0.1, 1],
    'kernel': ['linear', 'rbf']
}
```

A code editor window showing a Python dictionary for an SVM parameter grid. The code is:

```
# Parameter grid for SVM
param_grid = {
    'C': [0.1, 1, 10],
    'gamma': [0.01, 0.1, 1],
    'kernel': ['linear', 'rbf']
}
```

Purpose of the Code:

The code is setting up a **parameter grid** for performing **hyperparameter tuning** with the **Support Vector Machine (SVM)** model. Specifically, it is preparing to experiment with different values for three key hyperparameters:

1. **C**: Regularization strength
2. **gamma**: Influence of data points
3. **kernel**: Type of decision boundary

The parameter grid will be used by a method like **GridSearchCV** to evaluate the performance of the SVM model with different combinations of these parameters.

Output:

The code itself does not produce an immediate output. However, when passed to a grid search algorithm (such as **GridSearchCV**), it will perform the following:

1. **Train multiple models**: It will train multiple SVM models with all combinations of the specified values for C, gamma, and kernel. For example, it will train models for:
 - C = 0.1, gamma = 0.01, kernel = linear
 - C = 1, gamma = 0.1, kernel = rbf
 - etc.
2. **Cross-validation**: For each combination, the model will be evaluated using cross-validation on the training data to find the best set of hyperparameters.
3. **Best Parameters**: After the grid search is complete, it will return the best combination of parameters (C, gamma, kernel) based on the performance metrics (like accuracy or R² score).

4. **Model Performance:** The grid search will also provide the model's performance for each combination of hyperparameters, allowing you to choose the model that performs best according to the validation set.

In summary, the purpose of this parameter grid is to find the optimal set of hyperparameters for an SVM model by exploring various values of **C**, **gamma**, and **kernel**. The output of this process will be the combination of parameters that results in the best model performance.

5. Grid Search

Grid Search is a technique used to optimize the hyperparameters of a machine learning model. It systematically searches through a manually specified parameter grid to find the best combination of parameters that yields the best model performance. The method performs an exhaustive search over a given set of hyperparameters, trying all possible combinations.

In the context of the provided code, **GridSearchCV** is being used for **hyperparameter tuning** of an **SVR (Support Vector Regression)** model. This means that it will train multiple versions of the SVR model, each with different parameter combinations, and evaluate their performance using cross-validation. The goal is to identify the best combination of parameters for the model, in this case, the values for **C**, **gamma**, and **kernel** that provide the best prediction accuracy as measured by the R^2 score.

```
# GridSearchCV to find best parameters
from sklearn.model_selection import GridSearchCV
from sklearn.svm import SVR
X_train, X_test, y_train, y_test =
train_test_split(X_scaled, target, test_size=0.2,
random_state=42)
grid_search = GridSearchCV(SVR(), param_grid, cv=5,
scoring='r2')
grid_search.fit(X_train, y_train)

print("Best SVM Parameters:", grid_search.best_params_)
```

```
# Predict on the test data
y_pred = grid_search.predict(X_test)

# Calculate R2 score
r2 = r2_score(y_test, y_pred)
print(f"R2 Score: {r2:.4f}")
```

Output:

Best SVM Parameters: {'C': 10, 'gamma': 0.01, 'kernel': 'linear'}

R² Score: 0.9997

```
# GridSearchCV to find best parameters
from sklearn.model_selection import GridSearchCV
from sklearn.svm import SVR
X_train, X_test, y_train, y_test = train_test_split(X_scaled, target, test_size=0.2, random_state=42)
grid_search = GridSearchCV(SVR(), param_grid, cv=5, scoring='r2')
grid_search.fit(X_train, y_train)

print("Best SVM Parameters:", grid_search.best_params_)

# Predict on the test data
y_pred = grid_search.predict(X_test)

# Calculate R2 score
r2 = r2_score(y_test, y_pred)
print(f"R2 Score: {r2:.4f}")
```

```
Best SVM Parameters: {'C': 10, 'gamma': 0.01, 'kernel': 'linear'}
R2 Score: 0.9997
```

Explanation of the Code and Its Purpose

1. Data Splitting:

- The dataset is split into training and test sets using the `train_test_split` function. This ensures that the model is trained on one subset of the data (training set) and evaluated on another (test set) to assess generalization performance.

2. GridSearchCV Setup:

- `GridSearchCV` is instantiated with the **SVR** model and the specified `param_grid`.

- The `param_grid` contains different values for the model's hyperparameters (`C`, `gamma`, and `kernel`), and `GridSearchCV` will try all combinations of these parameters.
 - `cv=5` specifies 5-fold cross-validation, meaning the training data will be split into 5 subsets, and the model will be trained and evaluated 5 times, each time using a different fold as the validation set and the rest as the training set.
 - `scoring='r2'` tells `GridSearchCV` to use the **R² score** as the metric for evaluating model performance.
3. **Fitting the GridSearchCV:**
- The `fit` method is called on the `grid_search` object with the training data (`X_train`, `y_train`). This initiates the process of training and evaluating the model with all possible combinations of hyperparameters from the `param_grid`.
4. **Best Parameters:**
- After fitting the grid search, the best combination of hyperparameters is obtained using `grid_search.best_params_`. These are the values of `C`, `gamma`, and `kernel` that resulted in the best performance.
5. **Prediction and Evaluation:**
- The best model is used to make predictions on the test set (`X_test`), and the predictions are evaluated by calculating the **R² score** between the predicted values (`y_pred`) and the true test values (`y_test`).

Purpose

The purpose of this code is to find the best hyperparameters for an **SVR** model using grid search. By exploring different combinations of hyperparameters, the grid search finds the optimal set that minimizes prediction error and maximizes the model's performance on unseen data.

Explanation of the Output

1. Best SVM Parameters:

- The best combination of hyperparameters found by the grid search is:
 - **C = 10:** This regularization parameter strikes a balance between underfitting and overfitting, allowing the model to better capture the relationships in the data.
 - **gamma = 0.01:** This parameter determines the influence of individual data points. A smaller gamma value means that the model has a smoother decision boundary.

- **kernel = 'linear'**: This indicates that the model uses a linear kernel, meaning that it assumes the data can be separated using a straight line or hyperplane.

2. **R² Score:**

- The R² score of **0.9997** indicates that the model explains approximately 99.97% of the variance in the test data. This suggests that the SVR model with the optimized hyperparameters performs exceptionally well on the test set, closely predicting the target values. This high R² value indicates that the model generalizes well and is highly accurate in its predictions.

In summary, **GridSearchCV** has optimized the hyperparameters of the SVR model, resulting in a very high-performing model with an R² score of 0.9997 on the test data. This process helps to tune the model for the best possible performance, ensuring it generalizes well to unseen data.

V. Conclusion

In this analysis, various machine learning models were evaluated for stock market prediction, including **Linear Regression**, **Decision Tree**, **Random Forest**, and **Support Vector Regression (SVM)**.

- **Best Models:** Both **Random Forest** and **Decision Tree** models performed excellently, achieving perfect R^2 scores of **1.0000**, making them the most effective at capturing the complex, non-linear relationships in stock market data.
- **SVM** also showed strong performance with an R^2 score of **0.9997**, slightly lower than the tree-based models but still highly accurate.
- **Linear Regression**, while performing well, lacked the flexibility of tree-based models, which made them less suitable for this task.

Overall, **Random Forest** and **Decision Tree** were the best models for this task, offering robust, high-accuracy predictions for stock market values.

References:

- [A comparative study of supervised machine learning algorithms for stock market trend prediction](#)
- [Predicting stock and stock price index movement using Trend Deterministic Data Preparation and machine learning techniques](#)
- [Stock Market Prediction Using Machine Learning](#)