ASSET PRICE PREDICTION USING MULTIPLE LINEAR REGRESSION

Supervisor: Amir Eaman

Report Structure

I) Introduction

The stock market is a crucial component of the global economy, providing a platform for individuals and institutions to invest and trade shares of publicly traded companies. Its influence on the financial well-being of countries and individuals makes it a key area of interest for both economists and investors. However, predicting stock prices remains a complex task due to the inherent volatility of the market, driven by numerous factors such as supply and demand, geopolitical events, and investor sentiment. Understanding these dynamics is essential for making informed investment decisions.

Investors face significant challenges in accurately predicting stock market movements, largely due to the unpredictability of various influencing factors. Traditional methods of stock price analysis often fail to capture the complexities of the market, resulting in suboptimal investment decisions. There is a pressing need for robust statistical and machine learning tools that can analyze large datasets and provide more reliable forecasts.

This project seeks to address this need by utilizing multiple linear regression to predict the next day's higher price value of the S&P 500, offering investors a more data-driven approach to decision-making. The primary objective of this project is to develop a multiple linear regression model to predict the higher price value of the S&P 500 from 2013 to 2018. This model will be evaluated based on its accuracy and reliability in forecasting market trends. The project aims to demonstrate the potential of statistical techniques in financial forecasting and provide insights into how investors can use these tools to improve their investment strategies.

This study focuses on the S&P 500 index, one of the most significant benchmarks of the U.S. stock market, during the period from 2013 to 2018. The S&P 500 was chosen for its excellent representation of the broader market, comprising 500 of the largest companies in various industries across the U.S. economy. This makes it a highly reliable indicator of overall market performance, as it captures the trends and behaviors of both individual sectors and the economy as a whole. The index's historical data provides a robust foundation for price prediction models, making it ideal for a comprehensive analysis.

The 2013 to 2018 period was chosen for several reasons. Firstly, this timeframe includes key economic events, such as the post-2008 financial recovery, which brought gradual but significant market growth, and various geopolitical events that influenced market trends. Secondly, this period offers a relatively stable yet dynamic market environment, providing rich data for analysis while avoiding the extreme disruptions caused by more recent events like the COVID-19 pandemic. These factors make this timeframe ideal for studying price predictions using multiple linear regression, offering insights into how well the model performs under normal market conditions.

The 2019 to 2024 period was not chosen because this period is highly volatile and could introduce more noise into the data. The extreme market movements during this time could make predictions more difficult, especially with a linear regression method, which performs better in more stable environments. It is also likely that this period would be harder to generalize when predicting more "normal" market trends outside of crisis events.

II) Literature Review

Stock price forecasting is a key discipline in finance, relying on two main approaches: traditional methods and modern methods. Each of these approaches offers specific advantages and limitations in terms of accuracy, complexity, and speed.

Traditional methods are primarily represented by fundamental analysis and technical analysis. Fundamental analysis is based on studying the economic and financial performance of companies by examining indicators like revenues, net profits, cash flows, and the price-to-earnings (P/E) ratio. The goal is to understand the "intrinsic value" of a stock and compare this value to the current market price to determine if a stock is undervalued or overvalued [1]. Investors who adopt this approach focus on quantitative and qualitative elements that influence a company's long-term performance, such as the competitiveness of its products, its management team, or industry trends in which it operates [2].

Complementing this, technical analysis focuses on observing historical price charts and trading volumes. Technical analysts use indicators like moving averages, Bollinger Bands, and the Relative Strength Index (RSI) to detect short- and medium-term trends in stock prices [3]. These indicators are based on simple mathematical models but are effective for highly volatile markets. By studying past movements, they hope to identify patterns that could repeat and provide buy or sell signals.

While traditional approaches are robust and well-established, especially for long-term investments, they have certain limitations. Fundamental analysis allows for an in-depth understanding of companies and their fundamentals, which is particularly useful for value-focused investors, such as those following the principles of Benjamin Graham and Warren Buffet [1][4]. However, it can prove too slow for modern markets, where information circulates more rapidly and investment decisions need to be made in real-time [3]. Moreover, fundamental analysis depends on accounting and economic data that can be subject to subjective interpretations.

Similarly, technical analysis is often criticized for its retrospective approach. It relies exclusively on historical data, assuming that past price movements are the best indicators of future movements, which can be insufficient when facing unforeseen events like economic crises or radical changes in monetary policy [5]. That said, it remains very popular among active traders due to its simplicity and speed of execution.

With the evolution of financial markets and the rise of computing power, modern methods of stock price forecasting, such as statistical models and machine learning algorithms, have gained importance. Among the most common, Multiple Linear Regression (MLR) is a statistical technique that models the relationship between a dependent variable (the price of a stock) and several independent variables (economic and financial factors). MLR is widely used in stock forecasting because it can integrate a large number of variables, such as interest rates, a company's earnings, and market volatility to obtain an accurate prediction of future prices [6].

Furthermore, machine learning algorithms, like neural networks, decision trees, and random forests, allow for the analysis of massive datasets and the modeling of complex relationships between variables. Unlike MLR, which assumes a linear relationship between variables, these algorithms can capture nonlinear relationships and more subtle interactions in the data [7]. Neural networks, in particular, are capable of "training" on historical data to detect hidden patterns and provide accurate forecasts, even in volatile market environments [8]. Statistical and machine learning models enable the processing of large amounts of data in real-time and offer more precise forecasts than traditional methods, especially when complex relationships exist between different economic variables [6][9]. For example, in a highly volatile market context, these models can quickly adjust their forecasts based on new incoming data.

However, they also present limitations. Regression models, although useful, often assume linear relationships that do not always reflect the complexity of financial markets [10]. Moreover, machine learning algorithms, while powerful, can be considered "black boxes" due to their complexity, making it difficult to interpret the obtained results [7][8].

In summary, traditional methods like fundamental analysis and technical analysis offer an intuitive and qualitative understanding of the market but lack the speed and accuracy needed in modern stock environments where rapid price fluctuations require real-time adjustments. On the other hand, modern methods, such as Multiple Linear Regression and machine learning algorithms, provide greater accuracy in forecasts by exploiting large amounts of data and modeling complex relationships. However, they require advanced technical expertise to be correctly implemented and interpreted [6][9][10].

Multiple Linear Regression is an essential statistical method used in the financial field to model the relationship between a dependent variable and several independent variables. This technique is widely employed to analyze the impact of multiple economic and financial factors on variables such as stock prices, a company's earnings, or interest rates. Multiple Linear Regression is an extension of simple linear regression, which models the relationship between a single dependent variable (e.g., the price of a stock) and several explanatory or independent variables (such as the company's earnings, interest rates, or GDP growth) [3].

Mathematically, MLR can be expressed in the following form:
$\mathbf{Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \ldots + \beta_n X_n + \varepsilon}$
In this equation, Y is the dependent variable (e.g., the stock price), X_1, X_2, ..., X_n are the independent variables, β_0 is the intercept, β_1, β_2, ..., β_n are the regression coefficients associated with each independent variable, and ε is the random error term [1]. The role of MLR is to minimize the error ε by adjusting the coefficients β to optimize the accuracy of the prediction. The adjustment of the coefficients is often done by the method of Ordinary Least Squares (OLS), which consists of minimizing the sum of the squares of the differences between the observed values and the predicted values [11]. This process allows for a better approximation of the relationship between the variables and the target variable, thus increasing the predictive capability of the model.

However, MLR relies on several fundamental assumptions: linearity of the relationship between variables, absence of multicollinearity (the independent variables should not be highly correlated with each other), homoscedasticity (the variance of the errors must be constant), and normality of the errors [6].

Multiple Linear Regression has been widely used in the financial field to model and predict various financial variables. One of the earliest areas of application has been stock price forecasting. Researchers have often used MLR models to analyze the impact of economic factors, such as interest rates, GDP, and inflation, on stock prices in the short and long term [7].

For example, a multiple linear regression model can take into account variables like Earnings Per Share (EPS), the Price-to-Earnings (P/E) ratio, and interest rates to predict the variation in a stock's price [2].

In portfolio management, MLR is also applied to model the returns of a portfolio based on several risk factors. Factor regression models (based on MLR) consider different risk factors like the portfolio's sensitivity to the overall market, inflation rates, and interest rates to predict future returns [12]. These models are commonly used in the multifactor approach to portfolio management, such as the Fama and French model, which uses factors like company size and the book-to-market ratio to explain returns [9].

MLR has also been used to forecast macroeconomic indicators such as economic growth or unemployment rates by integrating independent variables like interest rates, inflation, and public spending [5]. This allows financial decision-makers to better understand how these factors influence financial market performance and to adjust their investment strategies accordingly.

In the field of derivatives, Multiple Linear Regression is used to model the relationship between option prices and factors such as implied volatility, risk-free interest rates, and the price of the underlying asset. This approach allows risk managers and traders to better evaluate option prices and design effective hedging strategies [10].

The main advantage of MLR is its simplicity and interpretability. Unlike more complex machine learning algorithms, MLR allows financial analysts to clearly understand the impact of each variable on the target variable, facilitating the interpretation of results and decision-making [8]. However, MLR also presents limitations. One of the main limitations is that it relies on the assumption of linearity, whereas relationships in financial markets are often nonlinear and influenced by factors difficult to model simply. Moreover, MLR is sensitive to the presence of multicollinearity among the explanatory variables, which can distort results and reduce the reliability of predictions [13].

In conclusion, Multiple Linear Regression remains a valuable tool for financial forecasting, offering a rigorous methodology to analyze the factors influencing the prices of financial assets. Its application continues to evolve with the improvement of statistical techniques and the increasing integration of massive data, but it requires rigorous control of the underlying assumptions to avoid biases in forecasts.

Stock prices are influenced by economic and company-specific factors. Macroeconomic variables like GDP affect the stock market; an increase in GDP boosts profits and stock prices [5], but excessive growth can cause inflation, prompting central banks to raise interest rates and slow market growth. Interest rates impact stock valuation; their rise increases borrowing costs, reducing profits and future profitability, which can lower stock prices [12]. High rates also encourage investors to shift to safer bonds with better returns [7]. Inflation raises operating costs, reducing margins if costs can't be passed on, often leading to lower stock prices, though companies in high-demand sectors like commodities may benefit [9].

Company-specific indicators, more measurable than macroeconomic variables, influence stock prices. Profits exceeding expectations raise stock prices, while disappointing results lower them [4]. Earnings per share (EPS) is a key profitability indicator. Debt levels affect perceptions of financial health; high debt can limit the ability to reinvest or pay dividends, negatively impacting stock prices [6]. However, well-utilized debt can generate superior returns [1].

Previous studies on the forecasting of stock prices using Multiple Linear Regression and other statistical techniques have encountered several methodological problems and structural limitations. One of the main problems encountered is multicollinearity among explanatory variables. When two or more independent variables are highly correlated with each other, it can make it difficult to determine the individual effects of each variable on the dependent variable [13]. Multicollinearity can also distort the results of regression models, making predictions less reliable.

We plan to address this problem by using Ridge regression, which penalizes large coefficients and reduces the impact of multicollinearity among variables. Another common issue is missing or incomplete data, which can bias forecasting results. We plan to address this limitation by using data imputation techniques to estimate missing values in our dataset [10]. Overfitting is another recurring problem in forecasting models based on historical data. An overfitted model may perfectly fit the training data but fail to generalize correctly to new data [8]. We plan to solve this problem by applying cross-validation techniques and regularizing models, such as using Ridge regression, to avoid excessive fitting to historical data [6].

III) Core Analysis and Modeling

We now move to the core of the project, where the focus shifts to applying advanced techniques for data analysis, preprocessing, and model building. This section involves preparing the dataset for predictive modeling, implementing statistical and machine learning algorithms, and evaluating the results to achieve the project's objectives.

```python
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
from scipy.interpolate import CubicSpline
import random
import statsmodels.api as sm
import warnings
from scipy import stats
from scipy.stats import boxcox
from sklearn.preprocessing import StandardScaler, MinMaxScaler
from sklearn.model_selection import train_test_split, cross_val_score
from sklearn.metrics import make_scorer, mean_squared_error,
mean_absolute_error, r2_score
from sklearn.linear_model import LinearRegression, Ridge
```

Importing Required Libraries

To facilitate the workflow, essential libraries were imported:

- **pandas**: For data manipulation and analysis.
- **matplotlib.pyplot** and **seaborn**: For data visualization.
- **numpy**: For numerical computations.

- **scipy.interpolate (CubicSpline)**: For implementing cubic spline interpolation to handle missing or imputed data.
- **random**: For simulating scenarios like missing values.
- **statsmodels.api** and **scipy.stats**: For statistical transformations and analysis.
- **sklearn**: For preprocessing (e.g., scaling), dataset splitting, model building (LinearRegression, Ridge), and performance evaluation (e.g., RMSE, R^2).
- **warnings**: To manage and suppress non-critical warnings during execution.

SECTION I

Data Collection

1) Independent Variables

Open

- **Definition**: The price at which the S&P 500 opens for the trading day.
- **Relevance**: Reflects overnight changes in market sentiment and investor behavior. The opening price is influenced by news and developments occurring after the previous day's close and is critical for analyzing daily market conditions.

High

- **Definition**: The highest price reached by the S&P 500 during the trading day.
- **Relevance**: Indicates the maximum level of optimism or demand during the trading session. This is useful for assessing intraday volatility and identifying potential resistance levels.

Low

- **Definition**: The lowest price reached by the S&P 500 during the trading day.
- **Relevance**: Shows the minimum level of pessimism or selling pressure during the trading session. This helps in understanding potential support levels.

Close

- **Definition**: The price at which the S&P 500 closes at the end of the trading day.
- **Relevance**: Summarizes the day's trading activity and is widely used as a benchmark for calculating technical indicators and forecasting future price movements.

MA_20 (20-period Moving Average)

- **Definition**: The average of the S&P 500 closing prices over the last 20 trading days.
- **Relevance**: Smoothens short-term price fluctuations, helping to identify trends and momentum over a short period. It provides a more stable measure of the market direction compared to daily closing prices.
- **Mathematical Formula**:

$$MA_{20} = \frac{\sum_{i=1}^{20} \text{Close}_i}{20}$$

Where:
Close_i: Closing price on day i.
20: The number of periods.

MA_50 (50-period Moving Average)

- **Definition**: The average of the S&P 500 closing prices over the last 50 trading days.
- **Relevance**: Tracks medium-term trends, providing insights into market direction over a longer timeframe. It is particularly useful for identifying significant market shifts.
- **Mathematical Formula**:

$$MA_{50} = \frac{\sum_{i=1}^{50} \text{Close}_i}{50}$$

MA_100 (100-period Moving Average)

- **Definition**: The average of the S&P 500 closing prices over the last 100 trading days.
- **Relevance**: Highlights long-term market direction and momentum. It is often used by analysts to confirm major market trends.
- **Mathematical Formula**:

$$MA_{100} = \frac{\sum_{i=1}^{100} \text{Close}_i}{100}$$

MA_200 (200-period Moving Average)

- **Definition**: The average of the S&P 500 closing prices over the last 200 trading days.
- **Relevance**: Tracks very long-term trends, showing overall market momentum and stability. It is considered a critical indicator for assessing market health.
- **Mathematical Formula**:

$$MA_{200} = \frac{\sum_{i=1}^{200} \text{Close}_i}{200}$$

Stoch_K (Stochastic Oscillator %K)

- **Definition**: Measures the position of the closing price relative to the high-low range over a specific period (commonly 14 days).
- **Relevance**: Identifies overbought or oversold conditions, indicating potential trend reversals. A high value suggests overbought conditions, while a low value indicates oversold conditions.

- **Mathematical Formula**:

$$Stoch_K = \frac{\text{Close} - \text{Lowest Low}}{\text{Highest High} - \text{Lowest Low}} \times 100$$

Where:
Close: The current closing price.
Lowest Low: The lowest price over the look-back period (e.g., 14 days).
Highest High: The highest price over the look-back period.

Stoch_D (Smoothed Stochastic Oscillator)

- **Definition**: The smoothed moving average of Stochastic %K, typically over 3 periods.
- **Relevance**: Reduces noise in the Stoch_K values, providing a more stable signal for overbought or oversold conditions.
- **Mathematical Formula**:

$$Stoch_D = \text{SMA of } Stoch_K \text{ over 3 periods}$$

RSI_28 (Relative Strength Index)

- **Definition**: Measures the strength of price changes over the past 28 periods to identify overbought or oversold conditions.
- **Relevance**: Helps predict potential trend reversals and market strength. Values above 70 indicate overbought conditions, while values below 30 indicate oversold conditions.
- **Mathematical Formula**:

$$RSI = 100 - \frac{100}{1 + RS}$$

Where:

$$RS = \frac{\text{Average Gain over 28 periods}}{\text{Average Loss over 28 periods}}$$

RSI_MA

- **Definition**: Moving Average of the RSI values over a specific number of periods.
- **Relevance**: Smoothens RSI trends to provide a more reliable assessment of market strength.
- **Mathematical Formula**:

$$RSI_M A = \frac{\sum \text{RSI Values over n periods}}{n}$$

ADX (Average Directional Index)

- **Definition**: Measures the strength of a trend regardless of direction.
- **Relevance**: Helps determine if the market is trending or ranging, providing valuable insight for trend-following strategies.
- **Mathematical Formula**:

$$ADX = 100 \cdot ¿¿$$

Where:
$DM^{+¿=max(\text{High}_t - \text{High}_{t-1}, 0)¿}$

$DM^{-} = max(\text{Low}_{t-1} - \text{Low}_t, 0)$

$TR = max(\text{High}_t - \text{Low}_t, |\text{High}_t - \text{Close}_{t-1}|, |\text{Low}_t - \text{Close}_{t-1}|)$

MACD (Moving Average Convergence Divergence)

- **Definition**: Measures the difference between two exponential moving averages (EMAs) to identify momentum and trend direction.
- **Relevance**: Detects trend reversals and momentum shifts in asset prices.
- **Mathematical Formula**:

$$MACD = EMA_{12} - EMA_{26}$$

Where:
EMA_{12}: 12-period Exponential Moving Average.
EMA_{26}: 26-period Exponential Moving Average.

VIX (Close_VIX)

- **Definition**: The closing price of the CBOE Volatility Index (VIX), which measures the market's expectation of volatility over the next 30 days.
- **Relevance**: Often referred to as the "fear index," the VIX reflects market uncertainty. High VIX values are associated with increased risk and fear, which can negatively impact the S&P 500. Conversely, low VIX values indicate stability and lower risk.

NASDAQ (Close_NASDAQ)

- **Definition**: The closing price of the NASDAQ Composite Index, which represents a broad range of technology and growth-focused stocks.
- **Relevance**: The NASDAQ's performance is often correlated with the S&P 500. Strong performance in technology stocks can drive overall market growth, influencing the S&P 500.

DAX (Close_DAX)

- **Definition**: The closing price of the German DAX Index, a key indicator of the German and broader European stock market performance.
- **Relevance**: As one of the major global indices, the DAX reflects the health of European markets. A strong DAX can signal positive global economic conditions, potentially boosting the S&P 500.

Gold (Close_GOLD)

- **Definition**: The closing price of gold, a commodity often used as a safe-haven investment.
- **Relevance**: Gold prices often move inversely to the stock market. Rising gold prices may indicate increased market risk and investor fear, potentially leading to a drop in the S&P 500.

CPI (Close_CPI)

- **Definition**: The closing value of the Consumer Price Index (CPI), which measures the average change in prices paid by consumers for goods and services over time.
- **Relevance**: The CPI is a key indicator of inflation. High inflation can negatively affect the S&P 500 by reducing consumer purchasing power and increasing costs for businesses.

NFP (Close_NFP)

- **Definition**: The closing value of the Non-Farm Payroll (NFP) report, which measures the change in the number of people employed in the U.S., excluding agricultural workers.
- **Relevance**: The NFP report is a key economic indicator. Strong employment growth often signals a healthy economy, which can drive the S&P 500 higher. Conversely, weak NFP numbers may indicate economic challenges and lead to market declines.

Unemployment Rate (Close_T_chomage)

- **Definition**: The closing value of the U.S. unemployment rate, which measures the percentage of the labor force that is unemployed but actively seeking employment.
- **Relevance**: The unemployment rate is a fundamental measure of economic health. A rising unemployment rate often indicates economic slowdowns, negatively impacting the S&P 500. Conversely, a low unemployment rate supports economic growth and market stability.

2) Dependent Variable

The Highest Level the Price of the S&P 500 Will Reach Tomorrow

- **Definition**: The highest price the S&P 500 will reach during the next trading day.
- **Relevance**:
  Selecting the highest price the S&P 500 will reach tomorrow as the dependent variable is a well-considered choice as it captures the peak market demand and optimism, offering critical insights into future price movements.
  - **Decision-Making**: This metric enables traders and investors to establish precise target prices and optimize risk management through stop-loss and take-profit

strategies.

- **Intraday Volatility**: It reflects intraday fluctuations and potential resistance levels, essential for understanding market trends and assessing the dynamics of price movements.

- **Strategic Insights**: By focusing on this key indicator, the model provides a robust foundation for making informed and strategic trading or investment decisions.

3) Data Organization

In this project, organizing the independent variables into a single cohesive dataframe was one of the main challenges encountered during the data preparation phase. The data was initially sourced from TradingView, where each independent variable was provided in a separate dataset. The primary issue was ensuring that all independent variables had corresponding values for each trading day, as inconsistencies or missing data could significantly affect the model's performance.

This required careful synchronization of datasets to ensure that all variables aligned correctly by date. Furthermore, handling discrepancies, such as different formats or timeframes across datasets, added an additional layer of complexity. In the following section, the process of organizing and structuring the data into a unified format is detailed, highlighting the steps taken to address these challenges and ensure the integrity of the dataset for analysis.

```
snp=pd.read_csv("C:/Users/omar alaoui/Desktop/CSI4900/variables independantes/snp500 + indicators.csv")
VIX=pd.read_csv("C:/Users/omar alaoui/Desktop/CSI4900/variables independantes/VIX.csv")
T_chomage=pd.read_csv("C:/Users/omar alaoui/Desktop/CSI4900/variables independantes/CHOMAGE.csv")
CPI=pd.read_csv("C:/Users/omar alaoui/Desktop/CSI4900/variables independantes/CPI.csv")
NFP=pd.read_csv("C:/Users/omar alaoui/Desktop/CSI4900/variables independantes/NFP.csv")
NASDAQ=pd.read_csv("C:/Users/omar alaoui/Desktop/CSI4900/variables independantes/NASDAQ.csv")
DAX=pd.read_csv("C:/Users/omar alaoui/Desktop/CSI4900/variables independantes/DAX.csv")
GOLD=pd.read_csv("C:/Users/omar alaoui/Desktop/CSI4900/variables independantes/GOLD.csv")
```

PART A

I) Data Importation

The independent variables were imported from CSV files into separate dataframes using the `pandas` library. Each file corresponds to a specific variable, such as the primary independent variables, including `open`, `high`, `low`, `close`, and additional technical indicators such as `MA 20`, `MA 50`, `MA 100`, `MA 200`, `%K`, `%D`, `RSI`, `RSI-MA`, `ADX`, and `MACD` (snp dataframe), `VIX`, unemployment rate (`T_chomage`), `CPI`, `NFP`, `NASDAQ`, `DAX`, and `GOLD`. The files were read from their respective paths and stored in variables (`snp`, `VIX`, `T_chomage`, `CPI`, `NFP`, `NASDAQ`, `DAX`, `GOLD`) for further processing and integration into a unified dataset.

```
snp.head()
```

```
         time     open     high      low    close          MA
MA.1  \
0  2010-04-07  1188.23  1189.60  1177.25  1182.44  1065.28815
1120.0954
1  2010-04-08  1181.75  1188.55  1175.12  1186.43  1066.75510
1120.9746
2  2010-04-09  1187.47  1194.66  1187.15  1194.37  1068.25145
1122.0459
3  2010-04-12  1194.93  1199.20  1194.71  1196.48  1069.72915
1123.0759
4  2010-04-13  1195.93  1199.04  1188.82  1197.30  1071.11435
1123.9559

        MA.2       MA.3         %K  ...  Upper Bollinger Band  \
0  1125.5338  1167.0595  89.695146  ...                   NaN
1  1127.4190  1169.1005  85.354573  ...                   NaN
2  1129.3564  1171.3070  87.153008  ...                   NaN
3  1131.5954  1173.6315  92.765775  ...                   NaN
4  1134.0640  1175.9710  95.685915  ...                   NaN

   Lower Bollinger Band  Regular Bullish  Regular Bullish Label  \
0                   NaN              NaN                    NaN
1                   NaN              NaN                    NaN
2                   NaN              NaN                    NaN
3                   NaN              NaN                    NaN
4                   NaN              NaN                    NaN

   Regular Bearish  Regular Bearish Label        ADX  Histogram
MACD  \
0              NaN                    NaN  32.308754   0.074424
15.461132
1              NaN                    NaN  32.183480  -0.074375
15.293739
2              NaN                    NaN  32.517631   0.202864
15.621694
3              NaN                    NaN  33.133446   0.360082
15.868933
4              NaN                    NaN  32.833552   0.350686
15.947209

      Signal
0  15.386708
1  15.368115
```

```
2   15.418831
3   15.508851
4   15.596523

[5 rows x 23 columns]

# List of columns to drop
columns_to_drop = [
    'Upper Bollinger Band',
    'Lower Bollinger Band',
    'Regular Bullish',
    'Regular Bullish Label',
    'Regular Bearish',
    'Regular Bearish Label',
    'Histogram',
    'Signal'
]

# Drop the columns
snp = snp.drop(columns=columns_to_drop)
```

II) Data Cleaning

To streamline the dataset, unnecessary columns that do not contribute to the analysis were removed from the `snp` dataframe. These columns include indicators such as `Upper Bollinger Band`, `Lower Bollinger Band`, and other labels (`Regular Bullish`, `Regular Bearish`, etc.), as well as `Histogram` and `Signal`. The `drop` method from the `pandas` library was used to remove these columns, ensuring a cleaner and more focused dataset for analysis.

```
# Display the first rows of the DataFrame to verify
snp.head()

         time     open     high      low    close          MA
MA.1  \
0  2010-04-07  1188.23  1189.60  1177.25  1182.44  1065.28815
1120.0954
1  2010-04-08  1181.75  1188.55  1175.12  1186.43  1066.75510
1120.9746
2  2010-04-09  1187.47  1194.66  1187.15  1194.37  1068.25145
1122.0459
3  2010-04-12  1194.93  1199.20  1194.71  1196.48  1069.72915
1123.0759
4  2010-04-13  1195.93  1199.04  1188.82  1197.30  1071.11435
1123.9559

       MA.2       MA.3         %K         %D        RSI  RSI-based MA
\
0  1125.5338  1167.0595  89.695146  90.676762  63.113336     62.939133
```

```
1  1127.4190  1169.1005  85.354573  89.994974  63.968651     62.988517

2  1129.3564  1171.3070  87.153008  87.400909  65.614080     63.295363

3  1131.5954  1173.6315  92.765775  88.424452  66.041450     63.546927

4  1134.0640  1175.9710  95.685915  91.868232  66.210700     63.693926


         ADX       MACD
0  32.308754  15.461132
1  32.183480  15.293739
2  32.517631  15.621694
3  33.133446  15.868933
4  32.833552  15.947209

# Dictionary for renaming columns
columns_rename = {
    'time': 'Date',
    'open': 'Open',
    'high': 'High',
    'low': 'Low',
    'close': 'Close',
    'RSI': 'RSI_28',
    'RSI-based MA': 'RSI_MA',
    'MA': 'MA_20',
    'MA.1': 'MA_50',
    'MA.2': 'MA_100',
    'MA.3': 'MA_200',
    '%K': 'Stoch_K',
    '%D': 'Stoch_D'
}

# Rename the column
snp = snp.rename(columns=columns_rename)

# Display the first rows of the DataFrame to verify
snp.head()

         Date     Open     High      Low    Close       MA_20
MA_50  \
0  2010-04-07  1188.23  1189.60  1177.25  1182.44  1065.28815
1120.0954
1  2010-04-08  1181.75  1188.55  1175.12  1186.43  1066.75510
1120.9746
2  2010-04-09  1187.47  1194.66  1187.15  1194.37  1068.25145
1122.0459
3  2010-04-12  1194.93  1199.20  1194.71  1196.48  1069.72915
1123.0759
```

```
4  2010-04-13  1195.93  1199.04  1188.82  1197.30  1071.11435
1123.9559

       MA_100     MA_200    Stoch_K    Stoch_D     RSI_28     RSI_MA  \
0  1125.5338  1167.0595  89.695146  90.676762  63.113336  62.939133
1  1127.4190  1169.1005  85.354573  89.994974  63.968651  62.988517
2  1129.3564  1171.3070  87.153008  87.400909  65.614080  63.295363
3  1131.5954  1173.6315  92.765775  88.424452  66.041450  63.546927
4  1134.0640  1175.9710  95.685915  91.868232  66.210700  63.693926

         ADX       MACD
0  32.308754  15.461132
1  32.183480  15.293739
2  32.517631  15.621694
3  33.133446  15.868933
4  32.833552  15.947209
```

III) Column Renaming

To improve clarity and consistency in the `snp` dataframe, the columns were renamed using a dictionary that maps the original names to more descriptive or standardized names. For example:

- `time` was renamed to `Date`
- `open` to `Open`, `high` to `High`, `low` to `Low`, and `close` to `Close`
- Technical indicators were renamed for better identification, such as `RSI` to `RSI_28`, `RSI-based MA` to `RSI_MA`, and moving averages like `MA` to `MA_20`.

This renaming ensures the column names are intuitive and aligned with the context of the analysis. The `rename` method from the `pandas` library was used, and the first rows of the dataframe were displayed using `snp.head()` to verify the changes.

```
# Ensure the 'Date' column is of datetime type
snp['Date'] = pd.to_datetime(snp['Date'])

# Define the start and end dates for filtering
start_date = '2013-01-01'
end_date = '2018-12-31'

# Identify the indices of rows to drop (outside the defined period)
indexes_to_drop = snp[(snp['Date'] < start_date) | (snp['Date'] >
end_date)].index

# Drop the identified rows
snp = snp.drop(indexes_to_drop)
```

```
# Display the first rows of the DataFrame
snp.head()

           Date     Open     High      Low    Close       MA_20
MA_50  \
690 2013-01-02  1426.19  1462.43  1426.19  1462.42  1390.67455
1421.6443
691 2013-01-03  1462.42  1465.47  1455.53  1459.37  1390.95055
1422.2158
692 2013-01-04  1459.37  1467.94  1458.99  1466.47  1391.23415
1422.8525
693 2013-01-07  1466.47  1466.47  1456.62  1461.89  1391.51600
1423.4127
694 2013-01-08  1461.89  1461.89  1451.64  1457.15  1391.78730
1423.9431

        MA_100     MA_200    Stoch_K    Stoch_D     RSI_28     RSI_MA
\
690  1411.5234  1424.9335  52.699795  35.738707  58.669169  53.309777

691  1411.5640  1427.5495  82.404153  55.412018  57.982361  53.554508

692  1412.2296  1430.4090  96.274507  77.126151  59.137162  54.056273

693  1412.7910  1432.8065  93.391723  90.690128  58.069521  54.590883

694  1413.6718  1434.7605  91.259726  93.641985  56.965733  54.764475


           ADX       MACD
690  13.206005   6.184123
691  14.232887   8.194474
692  15.299410  10.242532
693  16.086106  11.365056
694  16.387387  11.736890
```

IV) Data Filtering

To ensure the dataset is limited to the relevant analysis period (2013-01-01 to 2018-12-31), the following steps were performed:

1. The `Date` column was converted to the datetime format using the `pd.to_datetime` function for proper date comparisons.
2. A date range was defined with `start_date` set to `2013-01-01` and `end_date` set to `2018-12-31`.
3. Rows outside this period were identified by filtering the dataframe where dates were either before the `start_date` or after the `end_date`. The indices of these rows were stored in `indexes_to_drop`.
4. The identified rows were dropped using the `drop` method.

This step ensures the dataset focuses only on the specified time range for consistency and relevance in the analysis. The first rows of the filtered dataframe were displayed using `snp.head()` to confirm the changes.

V) Data Organization for Other Variables

The same data preprocessing steps applied to the `snp` dataframe were also performed on the other datasets (`VIX`, `T_chomage`, `CPI`, `NFP`, `NASDAQ`, `DAX`, and `GOLD`). These steps included:

1. **Data Cleaning**.
2. **Column Renaming**.
3. **Date Filtering**.

This process ensured all datasets were cleaned, renamed, and filtered consistently, resulting in uniform and well-structured data ready for integration and analysis.

---

1) Vix

```
VIX.head()
```

```
         time      open      high       low     close
0  2009-06-23  31.30000  31.53999  27.82999  30.57999
1  2009-06-24  30.58000  30.58000  28.78999  29.04999
2  2009-06-25  29.45000  29.56000  26.29999  26.35999
3  2009-06-26  27.09000  27.22000  25.75999  25.92999
4  2009-06-29  25.92999  27.17999  25.28999  25.34999
```

```
# List of columns to drop
columns_to_drop = [
    'open',
    'high',
    'low'
]

# Drop the columns
VIX = VIX.drop(columns=columns_to_drop)

# Display the first rows of the DataFrame to verify
VIX.head()
```

```
         time     close
0  2009-06-23  30.57999
1  2009-06-24  29.04999
2  2009-06-25  26.35999
3  2009-06-26  25.92999
4  2009-06-29  25.34999
```

```python
# Dictionary for renaming columns
columns_rename = {
    'time': 'Date_VIX',
    'close': 'Close_VIX'
}

# Rename the columns
VIX = VIX.rename(columns=columns_rename)

# Ensure the 'Date_VIX' column is of datetime type
VIX['Date_VIX'] = pd.to_datetime(VIX['Date_VIX'])

# Define the start and end dates for filtering
start_date = '2013-01-01'
end_date = '2018-12-31'

# Identify the indices of rows to drop (outside the defined period)
indexes_to_drop = VIX[(VIX['Date_VIX'] < start_date) |
(VIX['Date_VIX'] > end_date)].index

# Drop the identified rows
VIX = VIX.drop(indexes_to_drop)

# Display the first rows of the DataFrame
VIX.head()
```

```
      Date_VIX  Close_VIX
888 2013-01-02      14.68
889 2013-01-03      14.56
890 2013-01-04      13.83
891 2013-01-07      13.79
892 2013-01-08      13.62
```

---

2) Unemployement Rate

```python
T_chomage.head()
```

```
         time  close
0  1948-01-01    3.4
1  1948-02-01    3.8
2  1948-03-01    4.0
3  1948-04-01    3.9
4  1948-05-01    3.5
```

```python
# Dictionary for renaming columns
columns_rename = {
    'time': 'Date_T_chomage',
    'close': 'Close_T_chomage'
}
```

```
# Rename the columns
T_chomage = T_chomage.rename(columns=columns_rename)

# Ensure the 'Date_T_chomage' column is of datetime type
T_chomage['Date_T_chomage'] =
pd.to_datetime(T_chomage['Date_T_chomage'])

# Define the start and end dates for filtering
start_date = '2013-01-01'
end_date = '2018-12-31'

# Identify the indices of rows to drop (outside the defined period)
indexes_to_drop = T_chomage[(T_chomage['Date_T_chomage'] < start_date)
| (T_chomage['Date_T_chomage'] > end_date)].index

# Drop the identified rows
T_chomage = T_chomage.drop(indexes_to_drop)

# Display the first rows of the DataFrame
T_chomage.head()
```

```
    Date_T_chomage  Close_T_chomage
780     2013-01-01              8.0
781     2013-02-01              7.7
782     2013-03-01              7.5
783     2013-04-01              7.6
784     2013-05-01              7.5
```

3) CPI

```
CPI.head()
```

```
         time  close
0  1950-01-31   23.5
1  1950-02-28   23.5
2  1950-03-31   23.6
3  1950-04-30   23.6
4  1950-05-31   23.7
```

```
# Dictionary for renaming columns
columns_rename = {
    'time': 'Date_CPI',
    'close': 'Close_CPI'
}

# Rename the columns
CPI = CPI.rename(columns=columns_rename)
```

```
# Ensure the 'Date_CPI' column is of datetime type
CPI['Date_CPI'] = pd.to_datetime(CPI['Date_CPI'])

# Define the start and end dates for filtering
start_date = '2013-01-01'
end_date = '2018-12-31'

# Identify the indices of rows to drop (outside the defined period)
indexes_to_drop = CPI[(CPI['Date_CPI'] < start_date) |
(CPI['Date_CPI'] > end_date)].index

# Drop the identified rows
CPI = CPI.drop(indexes_to_drop)

# Display the first rows of the DataFrame
CPI.head()
```

```
      Date_CPI  Close_CPI
756 2013-01-31    230.280
757 2013-02-28    232.166
758 2013-03-31    232.773
759 2013-04-30    232.531
760 2013-05-31    232.945
```

---

4) NFP

```
NFP.head()
```

```
         time   close
0  1939-02-28  177000
1  1939-03-31  180000
2  1939-04-30 -186000
3  1939-05-31  205000
4  1939-06-30  203000
```

```
# Dictionary for renaming columns
columns_rename = {
    'time': 'Date_NFP',
    'close': 'Close_NFP'
}

# Rename the columns
NFP = NFP.rename(columns=columns_rename)

# Ensure the 'Date_NFP' column is of datetime type
NFP['Date_NFP'] = pd.to_datetime(NFP['Date_NFP'])

# Define the start and end dates for filtering
start_date = '2013-01-01'
```

```
end_date = '2018-12-31'

# Identify the indices of rows to drop (outside the defined period)
indexes_to_drop = NFP[(NFP['Date_NFP'] < start_date) |
(NFP['Date_NFP'] > end_date)].index

# Drop the identified rows
NFP = NFP.drop(indexes_to_drop)

# Display the first rows of the DataFrame
NFP.head()
```

```
       Date_NFP  Close_NFP
887 2013-01-31     189000
888 2013-02-28     285000
889 2013-03-31     142000
890 2013-04-30     186000
891 2013-05-31     214000
```

---

5) NASDAQ

```
NASDAQ.head()
```

```
         time        open        high         low       close
0  2005-02-09  1534.40991  1534.47998  1504.59985  1506.80981
1  2005-02-10  1512.70996  1514.21997  1497.34985  1506.82983
2  2005-02-11  1505.09009  1535.54004  1498.67993  1530.50977
3  2005-02-14  1531.18994  1540.01001  1530.15991  1538.20996
4  2005-02-15  1538.37988  1561.16992  1535.33984  1547.29980
```

```
# List of columns to drop
columns_to_drop = [
    'open',
    'high',
    'low'
]

# Drop the columns
NASDAQ = NASDAQ.drop(columns=columns_to_drop)

# Dictionary for renaming columns
columns_rename = {
    'time': 'Date_NASDAQ',
    'close': 'Close_NASDAQ'
}

# Rename the columns
NASDAQ = NASDAQ.rename(columns=columns_rename)
```

```
# Ensure the 'Date_NASDAQ' column is of datetime type
NASDAQ['Date_NASDAQ'] = pd.to_datetime(NASDAQ['Date_NASDAQ'])

# Define the start and end dates for filtering
start_date = '2013-01-01'
end_date = '2018-12-31'

# Identify the indices of rows to drop (outside the defined period)
indexes_to_drop = NASDAQ[(NASDAQ['Date_NASDAQ'] < start_date) |
(NASDAQ['Date_NASDAQ'] > end_date)].index

# Drop the identified rows
NASDAQ = NASDAQ.drop(indexes_to_drop)

# Display the first rows of the DataFrame
NASDAQ.head()

     Date_NASDAQ  Close_NASDAQ
1987  2013-01-02    2746.46997
1988  2013-01-03    2732.26001
1989  2013-01-04    2724.48999
1990  2013-01-07    2724.21997
1991  2013-01-08    2718.71997
```

6) DAX

```
DAX.head()

         time     open     high      low    close
0  2006-12-08  6396.09  6435.38  6352.35  6427.41
1  2006-12-11  6449.76  6476.34  6447.94  6469.42
2  2006-12-12  6466.58  6484.10  6457.37  6476.17
3  2006-12-13  6479.04  6521.02  6470.86  6520.77
4  2006-12-14  6530.81  6559.38  6523.47  6552.58

# List of columns to drop
columns_to_drop = [
    'open',
    'high',
    'low'
]

# Drop the columns
DAX = DAX.drop(columns=columns_to_drop)

# Dictionary for renaming columns
columns_rename = {
    'time': 'Date_DAX',
    'close': 'Close_DAX'
}

# Rename the columns
DAX = DAX.rename(columns=columns_rename)

# Ensure the 'Date_DAX' column is of datetime type
DAX['Date_DAX'] = pd.to_datetime(DAX['Date_DAX'])

# Define the start and end dates for filtering
start_date = '2013-01-01'
end_date = '2018-12-31'

# Identify the indices of rows to drop (outside the defined period)
indexes_to_drop = DAX[(DAX['Date_DAX'] < start_date) |
(DAX['Date_DAX'] > end_date)].index

# Drop the identified rows
DAX = DAX.drop(indexes_to_drop)

# Display the first rows of the DataFrame
DAX.head()
```

```
       Date_DAX  Close_DAX
1541 2013-01-02    7778.78
1542 2013-01-03    7756.44
1543 2013-01-04    7776.37
1544 2013-01-07    7732.66
1545 2013-01-08    7695.83
```

---

7) GOLD

```
# List of columns to drop
columns_to_drop = [
    'open',
    'high',
    'low'
]

# Drop the columns
GOLD = GOLD.drop(columns=columns_to_drop)

# Dictionary for renaming columns
columns_rename = {
    'time': 'Date_GOLD',
    'close': 'Close_GOLD'
}
```

```python
# Rename the columns
GOLD = GOLD.rename(columns=columns_rename)

# Ensure the 'Date_GOLD' column is of datetime type
GOLD['Date_GOLD'] = pd.to_datetime(GOLD['Date_GOLD'])

# Define the start and end dates for filtering
start_date = '2013-01-01'
end_date = '2018-12-31'

# Identify the indices of rows to drop (outside the defined period)
indexes_to_drop = GOLD[(GOLD['Date_GOLD'] < start_date) |
(GOLD['Date_GOLD'] > end_date)].index

# Drop the identified rows
GOLD = GOLD.drop(indexes_to_drop)

# Display the first rows of the DataFrame
GOLD.head()
```

```
      Date_GOLD  Close_GOLD
1620 2013-01-02   1686.8860
1621 2013-01-03   1664.0250
1622 2013-01-04   1655.7705
1623 2013-01-07   1646.9935
1624 2013-01-08   1659.2565
```

PART B

I) Data Merging

After cleaning and preparing the datasets, the next step involved merging them into a unified dataframe to ensure all independent variables are aligned by date. This step allows for seamless analysis by combining relevant data from multiple sources.

```python
# Merge the 'snp' and 'VIX' DataFrames on their date columns
data = pd.merge(snp, VIX, left_on='Date', right_on='Date_VIX')
```

Merging Process

The `snp` and `VIX` dataframes were merged using the `pd.merge` function:

- The merge was performed on their respective date columns: `Date` in the `snp` dataframe and `Date_VIX` in the `VIX` dataframe.
- This ensured that the data from both sources is synchronized for each trading day.

The resulting dataframe, `data`, contains the combined information from both `snp` and `VIX`, ready for further processing.

The merging process was repeated for the remaining datasets (`GOLD`, `NASDAQ`, and `DAX`), ensuring all variables are aligned by their respective dates. Each dataset was sequentially merged into the unified `data` dataframe, enriching it with additional independent variables for comprehensive analysis.

```
data = pd.merge(data, NASDAQ, left_on='Date', right_on='Date_NASDAQ')

data = pd.merge(data, DAX, left_on='Date', right_on='Date_DAX')

data = pd.merge(data, GOLD, left_on='Date', right_on='Date_GOLD')

# Display the first rows of the merged DataFrame
data.head()

        Date     Open     High      Low    Close       MA_20
MA_50  \
0 2013-01-02  1426.19  1462.43  1426.19  1462.42  1390.67455
1421.6443
1 2013-01-03  1462.42  1465.47  1455.53  1459.37  1390.95055
1422.2158
2 2013-01-04  1459.37  1467.94  1458.99  1466.47  1391.23415
1422.8525
3 2013-01-07  1466.47  1466.47  1456.62  1461.89  1391.51600
1423.4127
4 2013-01-08  1461.89  1461.89  1451.64  1457.15  1391.78730
1423.9431

      MA_100     MA_200    Stoch_K  ...        ADX       MACD
Date_VIX  \
0  1411.5234  1424.9335  52.699795  ...  13.206005   6.184123 2013-01-
02
1  1411.5640  1427.5495  82.404153  ...  14.232887   8.194474 2013-01-
03
2  1412.2296  1430.4090  96.274507  ...  15.299410  10.242532 2013-01-
04
3  1412.7910  1432.8065  93.391723  ...  16.086106  11.365056 2013-01-
07
4  1413.6718  1434.7605  91.259726  ...  16.387387  11.736890 2013-01-
08

   Close_VIX  Date_NASDAQ Close_NASDAQ   Date_DAX Close_DAX  Date_GOLD
\
0      14.68   2013-01-02   2746.46997 2013-01-02   7778.78 2013-01-02

1      14.56   2013-01-03   2732.26001 2013-01-03   7756.44 2013-01-03

2      13.83   2013-01-04   2724.48999 2013-01-04   7776.37 2013-01-04

3      13.79   2013-01-07   2724.21997 2013-01-07   7732.66 2013-01-07

4      13.62   2013-01-08   2718.71997 2013-01-08   7695.83 2013-01-08
```

```
  Close_GOLD
0  1686.8860
1  1664.0250
2  1655.7705
3  1646.9935
4  1659.2565

[5 rows x 23 columns]
```

II) Merging Monthly Data

In the second part of the merging process, datasets with monthly data were integrated into the unified dataframe. Since some variables, like the CPI, are reported monthly, additional steps were required to ensure proper alignment with daily data.

```
# Add a 'Month' column to each DataFrame to allow merging
data['Month'] = data['Date'].dt.to_period('M')
CPI['Month'] = CPI['Date_CPI'].dt.to_period('M')

# Shift 'Close_CPI' by 1 to align with the previous month's data
CPI['Close_CPI'] = CPI['Close_CPI'].shift(1)

# Merge the DataFrames with a left join based on the month
data = pd.merge(data, CPI[['Month', 'Close_CPI']], on='Month',
how='left')

# Fill missing values using forward fill to propagate monthly values
for each day
data['Close_CPI'] = data['Close_CPI'].ffill()
```

1. **Adding a 'Month' Column**:
   A `Month` column was added to both `data` and `CPI` dataframes using the `dt.to_period('M')` method, allowing the datasets to be merged based on the corresponding month rather than specific dates.

2. **Shifting CPI Values**:
   The `Close_CPI` column in the `CPI` dataframe was shifted by one period using the `shift(1)` function to align each value with the previous month's data. This ensures that the monthly CPI value corresponds to the correct analysis period.

3. **Merging Dataframes**:
   A left join was performed to merge the `data` and `CPI` dataframes based on the `Month` column. This ensured all daily rows in `data` included the relevant CPI values for their corresponding month.

4. **Filling Missing Values**:
   Missing values in the `Close_CPI` column were filled using forward fill (`ffill`) to propagate monthly CPI values across all days within the same month.

The same process was applied to the `NFP` and `T_chomage` datasets.

```
# Add a 'Month' column to the NFP DataFrame to allow merging
NFP['Month'] = NFP['Date_NFP'].dt.to_period('M')

# Shift 'Close_NFP' by 1 to align with the previous month's data
NFP['Close_NFP'] = NFP['Close_NFP'].shift(1)

# Merge the DataFrames with a left join based on the month
data = pd.merge(data, NFP[['Month', 'Close_NFP']], on='Month',
how='left')

# Fill missing values using forward fill to propagate monthly values
for each day
data['Close_NFP'] = data['Close_NFP'].ffill()

# Add a 'Month' column to the T_chomage DataFrame to allow merging
T_chomage['Month'] = T_chomage['Date_T_chomage'].dt.to_period('M')

# Merge the DataFrames with a left join based on the month
data = pd.merge(data, T_chomage[['Month', 'Close_T_chomage']],
on='Month', how='left')

# Fill missing values using forward fill to propagate monthly values
for each day
data['Close_T_chomage'] = data['Close_T_chomage'].ffill()

# Display the final merged DataFrame
data.head()

        Date     Open     High      Low    Close       MA_20
MA_50  \
0 2013-01-02  1426.19  1462.43  1426.19  1462.42  1390.67455
1421.6443
1 2013-01-03  1462.42  1465.47  1455.53  1459.37  1390.95055
1422.2158
2 2013-01-04  1459.37  1467.94  1458.99  1466.47  1391.23415
1422.8525
3 2013-01-07  1466.47  1466.47  1456.62  1461.89  1391.51600
1423.4127
4 2013-01-08  1461.89  1461.89  1451.64  1457.15  1391.78730
1423.9431

      MA_100     MA_200    Stoch_K  ...  Date_NASDAQ  Close_NASDAQ
Date_DAX  \
0  1411.5234  1424.9335  52.699795  ...   2013-01-02    2746.46997
2013-01-02
```

```
1  1411.5640  1427.5495  82.404153  ...   2013-01-03    2732.26001
2013-01-03
2  1412.2296  1430.4090  96.274507  ...   2013-01-04    2724.48999
2013-01-04
3  1412.7910  1432.8065  93.391723  ...   2013-01-07    2724.21997
2013-01-07
4  1413.6718  1434.7605  91.259726  ...   2013-01-08    2718.71997
2013-01-08

   Close_DAX  Date_GOLD Close_GOLD    Month Close_CPI  Close_NFP  \
0    7778.78 2013-01-02  1686.8860  2013-01       NaN        NaN
1    7756.44 2013-01-03  1664.0250  2013-01       NaN        NaN
2    7776.37 2013-01-04  1655.7705  2013-01       NaN        NaN
3    7732.66 2013-01-07  1646.9935  2013-01       NaN        NaN
4    7695.83 2013-01-08  1659.2565  2013-01       NaN        NaN

  Close_T_chomage
0             8.0
1             8.0
2             8.0
3             8.0
4             8.0

[5 rows x 27 columns]
```

PART C

Finalizing the Dataset

In the finalization step, adjustments were made to ensure the completeness and accuracy of the dataset. This involved handling any remaining missing or unmatched values critical for analysis.

```
# Assign values to 'Close_CPI' and 'Close_NFP' for January 2013
data.loc[data['Month'] == '2013-01', 'Close_CPI'] = 229.6
data.loc[data['Month'] == '2013-01', 'Close_NFP'] = 243000.0

# Display the updated DataFrame
data.head()
```

```
        Date     Open     High      Low    Close       MA_20
MA_50  \
0 2013-01-02  1426.19  1462.43  1426.19  1462.42  1390.67455
1421.6443
1 2013-01-03  1462.42  1465.47  1455.53  1459.37  1390.95055
1422.2158
2 2013-01-04  1459.37  1467.94  1458.99  1466.47  1391.23415
1422.8525
3 2013-01-07  1466.47  1466.47  1456.62  1461.89  1391.51600
1423.4127
```

```
4 2013-01-08  1461.89  1461.89  1451.64  1457.15  1391.78730
1423.9431

      MA_100     MA_200    Stoch_K  ...  Date_NASDAQ  Close_NASDAQ
Date_DAX  \
0  1411.5234  1424.9335  52.699795  ...   2013-01-02    2746.46997
2013-01-02
1  1411.5640  1427.5495  82.404153  ...   2013-01-03    2732.26001
2013-01-03
2  1412.2296  1430.4090  96.274507  ...   2013-01-04    2724.48999
2013-01-04
3  1412.7910  1432.8065  93.391723  ...   2013-01-07    2724.21997
2013-01-07
4  1413.6718  1434.7605  91.259726  ...   2013-01-08    2718.71997
2013-01-08

   Close_DAX  Date_GOLD Close_GOLD    Month Close_CPI  Close_NFP  \
0    7778.78 2013-01-02  1686.8860  2013-01      229.6   243000.0
1    7756.44 2013-01-03  1664.0250  2013-01      229.6   243000.0
2    7776.37 2013-01-04  1655.7705  2013-01      229.6   243000.0
3    7732.66 2013-01-07  1646.9935  2013-01      229.6   243000.0
4    7695.83 2013-01-08  1659.2565  2013-01      229.6   243000.0

  Close_T_chomage
0             8.0
1             8.0
2             8.0
3             8.0
4             8.0

[5 rows x 27 columns]
```

1. **Assigning Values for January 2013**:
   Specific values were manually assigned to the `Close_CPI` and `Close_NFP` columns for January 2013, as these data points were missing from the dataset. The values used were:
   - `Close_CPI`: 229.6
   - `Close_NFP`: 243,000.0
2. **Updating the Dataframe**:
   The `data.loc` method was used to target rows where the `Month` column equals `2013-01`, ensuring the correct rows were updated.

```
# List of columns to drop
columns_to_drop = [
    'Date_VIX',
    'Date_NASDAQ',
    'Date_DAX',
    'Date_GOLD',
    'Month'
]
```

```
# Drop the columns
data = data.drop(columns=columns_to_drop)
```

Unnecessary columns (`Date_VIX`, `Date_NASDAQ`, `Date_DAX`, `Date_GOLD`, and `Month`) were dropped using the `drop` method to streamline the dataset, keeping only relevant variables for analysis.

```
# Display the first rows of the DataFrame to verify
data.head()
```

```
        Date     Open     High      Low    Close       MA_20
MA_50  \
0 2013-01-02  1426.19  1462.43  1426.19  1462.42  1390.67455
1421.6443
1 2013-01-03  1462.42  1465.47  1455.53  1459.37  1390.95055
1422.2158
2 2013-01-04  1459.37  1467.94  1458.99  1466.47  1391.23415
1422.8525
3 2013-01-07  1466.47  1466.47  1456.62  1461.89  1391.51600
1423.4127
4 2013-01-08  1461.89  1461.89  1451.64  1457.15  1391.78730
1423.9431

      MA_100     MA_200    Stoch_K  ...     RSI_MA        ADX
MACD  \
0  1411.5234  1424.9335  52.699795  ...  53.309777  13.206005
6.184123
1  1411.5640  1427.5495  82.404153  ...  53.554508  14.232887
8.194474
2  1412.2296  1430.4090  96.274507  ...  54.056273  15.299410
10.242532
3  1412.7910  1432.8065  93.391723  ...  54.590883  16.086106
11.365056
4  1413.6718  1434.7605  91.259726  ...  54.764475  16.387387
11.736890

   Close_VIX  Close_NASDAQ  Close_DAX  Close_GOLD  Close_CPI
Close_NFP  \
0      14.68    2746.46997    7778.78   1686.8860      229.6
243000.0
1      14.56    2732.26001    7756.44   1664.0250      229.6
243000.0
2      13.83    2724.48999    7776.37   1655.7705      229.6
243000.0
3      13.79    2724.21997    7732.66   1646.9935      229.6
243000.0
4      13.62    2718.71997    7695.83   1659.2565      229.6
243000.0
```

```
   Close_T_chomage
0              8.0
1              8.0
2              8.0
3              8.0
4              8.0

[5 rows x 22 columns]

#Deleting the date column
data = data.drop(columns='Date')

#Displaying the first rows of the dataframe to check
data.head()

      Open     High      Low    Close       MA_20      MA_50
MA_100  \
0  1426.19  1462.43  1426.19  1462.42  1390.67455  1421.6443
1411.5234
1  1462.42  1465.47  1455.53  1459.37  1390.95055  1422.2158
1411.5640
2  1459.37  1467.94  1458.99  1466.47  1391.23415  1422.8525
1412.2296
3  1466.47  1466.47  1456.62  1461.89  1391.51600  1423.4127
1412.7910
4  1461.89  1461.89  1451.64  1457.15  1391.78730  1423.9431
1413.6718

      MA_200    Stoch_K    Stoch_D  ...     RSI_MA        ADX
MACD  \
0  1424.9335  52.699795  35.738707  ...  53.309777  13.206005
6.184123
1  1427.5495  82.404153  55.412018  ...  53.554508  14.232887
8.194474
2  1430.4090  96.274507  77.126151  ...  54.056273  15.299410
10.242532
3  1432.8065  93.391723  90.690128  ...  54.590883  16.086106
11.365056
4  1434.7605  91.259726  93.641985  ...  54.764475  16.387387
11.736890

   Close_VIX  Close_NASDAQ  Close_DAX  Close_GOLD  Close_CPI
Close_NFP  \
0      14.68    2746.46997    7778.78   1686.8860      229.6
243000
1      14.56    2732.26001    7756.44   1664.0250      229.6
243000
2      13.83    2724.48999    7776.37   1655.7705      229.6
243000
```

```
3      13.79    2724.21997    7732.66   1646.9935      229.6
243000
4      13.62    2718.71997    7695.83   1659.2565      229.6
243000

   Close_T_chomage
0              8.0
1              8.0
2              8.0
3              8.0
4              8.0

[5 rows x 21 columns]
```

The `Date` column was removed from the dataset using the `drop` method, as it was no longer required for analysis.

```
# Lire le dataset
data_inter=pd.read_excel("C:/Users/omar
alaoui/Desktop/CSI4900/variables independantes/data_final pour
interpolation.xlsx")
data_inter.head()

        Date     Open     High      Low    Close       MA_20
MA_50  \
0 2013-01-02  1426.19  1462.43  1426.19  1462.42  1390.67455
1421.6443
1 2013-01-03  1462.42  1465.47  1455.53  1459.37  1390.95055
1422.2158
2 2013-01-04  1459.37  1467.94  1458.99  1466.47  1391.23415
1422.8525
3 2013-01-07  1466.47  1466.47  1456.62  1461.89  1391.51600
1423.4127
4 2013-01-08  1461.89  1461.89  1451.64  1457.15  1391.78730
1423.9431

      MA_100     MA_200    Stoch_K  ...     RSI_MA        ADX
MACD  \
0  1411.5234  1424.9335  52.699795  ...  53.309777  13.206005
6.184123
1  1411.5640  1427.5495  82.404153  ...  53.554508  14.232887
8.194474
2  1412.2296  1430.4090  96.274507  ...  54.056273  15.299410
10.242532
3  1412.7910  1432.8065  93.391723  ...  54.590883  16.086106
11.365056
4  1413.6718  1434.7605  91.259726  ...  54.764475  16.387387
11.736890

   Close_VIX  Close_NASDAQ  Close_DAX  Close_GOLD  Close_CPI
```

```
Close_NFP  \
0      14.68    2746.46997    7778.78   1686.8860       229.6
243000
1      14.56    2732.26001    7756.44   1664.0250       229.6
243000
2      13.83    2724.48999    7776.37   1655.7705       229.6
243000
3      13.79    2724.21997    7732.66   1646.9935       229.6
243000
4      13.62    2718.71997    7695.83   1659.2565       229.6
243000

   Close_T_chomage
0              8.0
1              8.0
2              8.0
3              8.0
4              8.0

[5 rows x 22 columns]
```

```
# Suppression des colonnes
data_inter = data_inter.drop(columns='Date')
# Affichage des premières lignes du DataFrame pour vérifier
data_inter.head()
```

```
      Open     High      Low    Close       MA_20      MA_50
MA_100  \
0  1426.19  1462.43  1426.19  1462.42  1390.67455  1421.6443
1411.5234
1  1462.42  1465.47  1455.53  1459.37  1390.95055  1422.2158
1411.5640
2  1459.37  1467.94  1458.99  1466.47  1391.23415  1422.8525
1412.2296
3  1466.47  1466.47  1456.62  1461.89  1391.51600  1423.4127
1412.7910
4  1461.89  1461.89  1451.64  1457.15  1391.78730  1423.9431
1413.6718

      MA_200    Stoch_K    Stoch_D  ...     RSI_MA        ADX
MACD  \
0  1424.9335  52.699795  35.738707  ...  53.309777  13.206005
6.184123
1  1427.5495  82.404153  55.412018  ...  53.554508  14.232887
8.194474
2  1430.4090  96.274507  77.126151  ...  54.056273  15.299410
10.242532
3  1432.8065  93.391723  90.690128  ...  54.590883  16.086106
11.365056
4  1434.7605  91.259726  93.641985  ...  54.764475  16.387387
```

```
11.736890

   Close_VIX  Close_NASDAQ  Close_DAX  Close_GOLD  Close_CPI
Close_NFP  \
0      14.68    2746.46997    7778.78   1686.8860      229.6
243000
1      14.56    2732.26001    7756.44   1664.0250      229.6
243000
2      13.83    2724.48999    7776.37   1655.7705      229.6
243000
3      13.79    2724.21997    7732.66   1646.9935      229.6
243000
4      13.62    2718.71997    7695.83   1659.2565      229.6
243000

   Close_T_chomage
0              8.0
1              8.0
2              8.0
3              8.0
4              8.0

[5 rows x 21 columns]
```

SECTION II

PART A: Data Preprocessing

Data preprocessing is a critical step in any data analysis or machine learning project. It ensures that the data is clean, consistent, and ready for meaningful analysis. Without proper preprocessing, the quality of results and the performance of predictive models can be significantly compromised. The key steps in data preprocessing include **handling missing values**, **outlier detection and correction**, and **data transformation**.

Properly preprocessed data leads to:

- More accurate and meaningful insights.
- Improved model performance.
- A stronger foundation for decision-making.

Investing time in preprocessing ensures that the data reflects its true patterns and relationships, enabling successful analysis and model development.

---

1) Handling Missing Values

Missing values are common in real-world datasets and, if not addressed properly, can lead to inaccurate results or biased models.

- **Why It Matters:**
  - Missing values can reduce the accuracy of analysis and model performance.
  - They can lead to errors during computations, such as invalid averages or model failures.
- **Techniques to Handle Missing Values:**
  - **Deletion**: Removing rows or columns with missing values (used when the missing data is minimal).
  - **Interpolation**: Estimating the missing values based on surrounding data (e.g., cubic spline interpolation).
  - **Imputation**: Replacing missing values with statistical measures (mean, median, or mode).

By filling in missing values effectively, the integrity and completeness of the dataset are preserved.

Due to the large size of the dataset, there may be many missing data points in the collected data. This is a common problem in statistics. Generally, the solutions to this problem are either deleting the missing data or using interpolation. The delete method does not require extra computation but can introduce large errors by removing too much data. For this reason, this project has chosen interpolation to fill in the missing data.
The method used here is **piecewise cubic spline interpolation**.

This method constructs a function (f) defined over an interval ([a, b]) and a set of nodes such that:
[a = x_0 < x_1 < ... < x_n = b]

Then, a **cubic spline interpolant** (S) for (f) satisfies the following conditions:

Explanation of Conditions:

1. **(S_n(x)) is a cubic spline interpolant on the subinterval ([x_j, x_{j+1}]), and (S_n(x_j)) means the value of the attribute in day (j):**
   - This means that on each subinterval ([x_j, x_{j+1}]), there is a cubic polynomial that fits the data. The function (S_n(x_j)) gives the value of the attribute at day (j).
2. **(S_n(x_j) = f(x_j)) and (S_n(x_{j+1}) = f(x_{j+1})):**
   - The spline exactly passes through the known data points. At points (x_j) and (x_{j+1}), the spline (S_n) equals the actual function values (f(x_j)) and (f(x_{j+1})), ensuring the curve passes through the real data points.
3. **(S_{n+1}(x_{j+1}) = S_n(x_{j+1})):**
   - This ensures **continuity** of the spline. The value of the spline at (x_{j+1}) must be the same whether calculated from the left ((S_n)) or the right ((S_{n+1})). This prevents any jumps or discontinuities in the curve.

4. **(S_{n+1}'(x_{j+1}) = S_n'(x_{j+1})):**
   - This condition ensures **continuity of the first derivative** (or slope) of the spline at (x_{j+1}). The slope of the curve at (x_{j+1}) is the same when approached from the left or right, maintaining a smooth curve at the join points.
5. **(S_{n+1}''(x_{j+1}) = S_n''(x_{j+1})):**
   - This ensures **continuity of the second derivative** (or curvature) of the spline at (x_{j+1}). A continuous second derivative guarantees the curve changes smoothly across join points, avoiding sharp changes in shape.
6. **One of the following boundary conditions is satisfied:**
   - (S''(x_0) = S''(x_n) = 0): Known as the **natural spline condition**, this forces the curve to be flat at the two ends (no curvature).

   - (S'(x_0) = f'(x_0)): This condition ensures the slope of the spline at the start matches the slope of the function (f), useful when the derivative at the beginning is known.

Flexibility of the Spline Method:

This spline interpolation method involves **four constants**, giving it sufficient flexibility to ensure that the resulting interpolant is not only continuously differentiable (no sharp changes) but also has a continuous second derivative (ensuring smooth curvature changes across its entire length).

This flexibility makes **cubic spline interpolation** highly effective for filling in missing data, as it guarantees smooth and realistic interpolated values, avoiding sudden jumps or irregularities. [14]

```
# Function for cubic spline interpolation
def fill_missing_with_cubic_spline(x, y):
    # Ensure x and y are Numpy arrays
    x = np.array(x)
    y = np.array(y)

    # Mask valid (non-missing) values
    valid_mask = ~np.isnan(y)  # Identify non-NaN values
    x_valid = x[valid_mask]
    y_valid = y[valid_mask]

    # Create a cubic spline interpolator
    spline = CubicSpline(x_valid, y_valid, bc_type='natural')  #
Natural spline condition

    # Interpolate to fill missing values
    y_interpolated = spline(x)

    return y_interpolated, spline
```

In this study, the collected dataset does not contain any missing data points. However, to illustrate the effectiveness of the proposed method, an additional dataset was artificially generated by randomly removing values.

```
# Dataset with missing values
data_inter=pd.read_excel("C:/Users/omar
alaoui/Desktop/CSI4900/variables independantes/data_final pour
interpolation.xlsx")
data_inter.head()

data_inter = data_inter.drop(columns='Date')

data_inter.head()
```

The `fill_missing_with_cubic_spline` function performs cubic spline interpolation to fill in missing values (`NaN`) in a dataset. It works as follows:

Inputs:

- `x`: An array of independent variable values (e.g., time or indices).
- `y`: An array of dependent variable values (e.g., observations) that may contain missing values (`NaN`).

Steps:

1. **Convert Inputs to Numpy Arrays**
   The input arrays `x` and `y` are converted to Numpy arrays for compatibility and computational efficiency.

2. **Identify Valid (Non-Missing) Data**
   - The function creates a mask to identify where `y` does **not** contain missing values (`NaN`).
   - The valid values of `x` and `y` are extracted.
3. **Create the Cubic Spline Interpolator**
   - A **cubic spline interpolator** is created using the valid data points.
   - The condition `bc_type='natural'` ensures a **natural spline**, where the second derivative at the endpoints is set to zero (flat curvature).
4. **Interpolate to Fill Missing Values**
   - The spline function is applied to the full `x` array, generating interpolated values for all points, including those that were missing.

Outputs:

- `y_interpolated`: An array where missing values have been filled using cubic spline interpolation.
- `spline`: The cubic spline interpolator, which can be used for further analysis or plotting.

Replacing Missing Values with Cubic Spline Interpolation and Accuracy Calculation

The process iterates through each column of the dataset and performs the following:

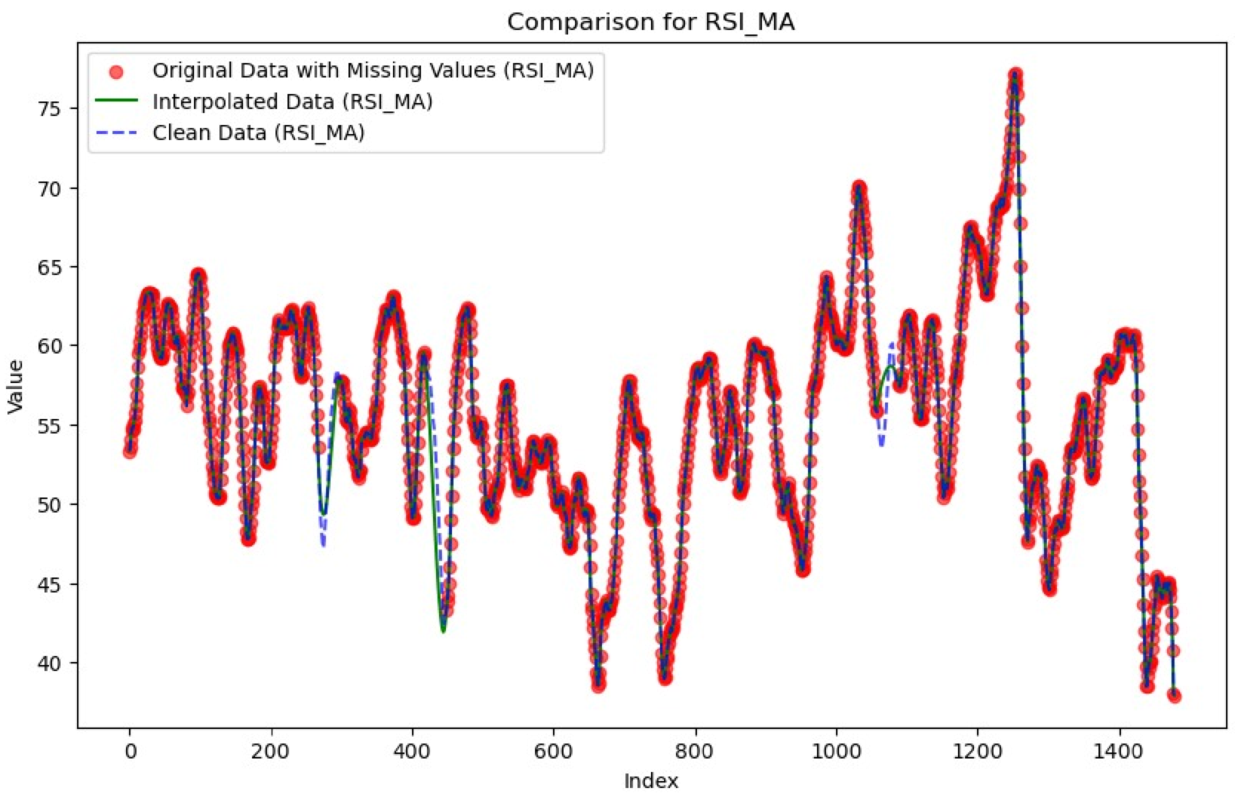
1. **Identify Columns with Missing Values:**
   - For columns containing `NaN` values, interpolation is applied.
2. **Cubic Spline Interpolation:**
   - Missing values are filled using **cubic spline interpolation** with natural boundary conditions to ensure smooth and realistic results.
3. **Accuracy Calculation Using Statistical Measures:**
   - Interpolated values are compared with real values from a clean dataset.
   - To evaluate the method's performance:
     - The **absolute errors** between interpolated and real values are calculated.
     - The **accuracy** is measured as the percentage of interpolated values within a ±2 margin of error.
     - Additional **statistical values** (e.g., total missing values, correct predictions, and error magnitude) are included to measure and demonstrate the accuracy of the method.
4. **Visualization:**
   - **Comparison Plot:** Displays original data (with missing values), interpolated data, and clean data for validation.
   - **Error Analysis Plot:** Visualizes the errors, including the threshold for the margin of error.
5. **Columns Without Missing Values:**
   - A message is displayed indicating that no interpolation is required.

By including statistical values and visualizations, this approach not only fills missing data smoothly but also quantifies and validates the method's accuracy.

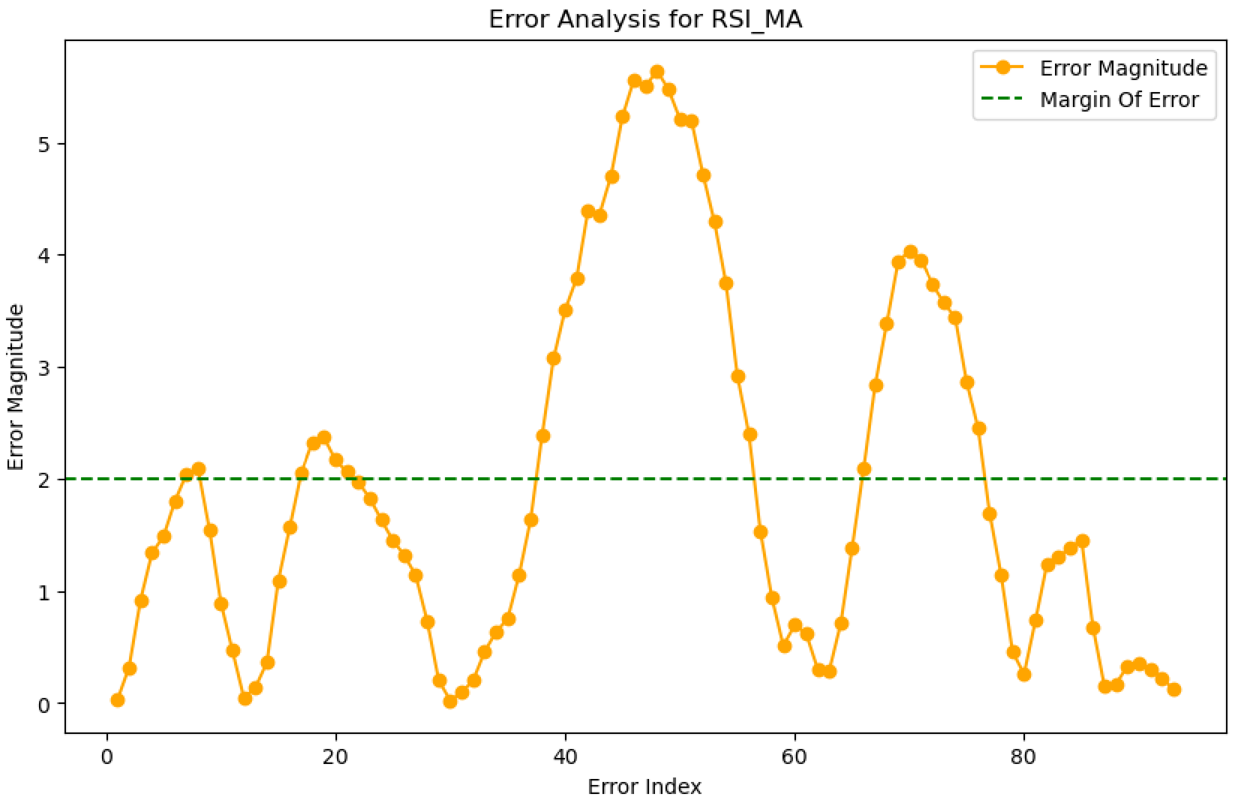
```
# Replace missing values in the dataset using cubic spline
interpolation and calculate accuracy
for column in data_inter.columns:
    if data_inter[column].isnull().any():  # Check if the column has
missing values
        print(f"Interpolating missing values for column: {column}")

        # Save the original column with NaNs for visualization
        original_data_with_nans = data_inter[column].copy()

        # Perform interpolation
        x_indices = np.arange(len(data_inter))  # Use row indices as x
        y_filled, spline_model =
fill_missing_with_cubic_spline(x_indices, data_inter[column])
```

```
        data_inter[column] = y_filled  # Replace the column with
interpolated values

        # Count total missing values
        total_missing = original_data_with_nans.isna().sum()

        # Compare interpolated values with real values
        real_values = data[column][original_data_with_nans.isna()]  #
Real values from clean data
        interpolated_values = y_filled[original_data_with_nans.isna()]
# Interpolated values
        errors = interpolated_values - real_values  # Calculate the
errors
        correct_predictions = np.sum(np.abs(errors) <= 2)  # Within ±2
margin
        accuracy = (correct_predictions / total_missing) * 100  #
Accuracy percentage

        # Print statistics
        print(f"Column: {column}")
        print(f"Total Missing Values: {total_missing}")
        print(f"Correct Predictions: {correct_predictions}")
        print(f"Accuracy: {accuracy:.2f}%\n")

        # Visualization: Plot both the data with missing values and
clean data
        plt.figure(figsize=(10, 6))

        # Plot original data (with missing values, shown as points)
        plt.scatter(
            np.arange(len(original_data_with_nans))
[~original_data_with_nans.isna()],
            original_data_with_nans.dropna(),
            label=f'Original Data with Missing Values ({column})',
            color='red',
            alpha=0.6
        )

        # Plot interpolated data as a continuous line
        plt.plot(data_inter[column], label=f'Interpolated Data
({column})', color='green')

        # Plot the clean data from the dataframe without missing
values
        plt.plot(data[column], label=f'Clean Data ({column})',
color='blue', linestyle='--', alpha=0.7)

        # Add titles and legend
        plt.title(f"Comparison for {column}")
        plt.legend()
        plt.xlabel("Index")
        plt.ylabel("Value")
        plt.show()

        # Visualization: Plot errors
        plt.figure(figsize=(10, 6))
        error_indices = np.arange(1, len(errors) + 1)  # Number errors
from 1 to total missing values
        plt.plot(error_indices, np.abs(errors), marker='o',
linestyle='-', color='orange', label='Error Magnitude')
        plt.axhline(2, color='green', linestyle='--', label='Margin Of
Error')  # ±2 margin threshold
        plt.title(f"Error Analysis for {column}")
        plt.xlabel("Error Index")
        plt.ylabel("Error Magnitude")
        plt.legend()
        plt.show()
    else:
        print(f"{column} doesn't contain missing values!")
```

```
Open doesn't contain missing values!
High doesn't contain missing values!
Low doesn't contain missing values!
Close doesn't contain missing values!
MA_20 doesn't contain missing values!
MA_50 doesn't contain missing values!
MA_100 doesn't contain missing values!
MA_200 doesn't contain missing values!
Stoch_K doesn't contain missing values!
Stoch_D doesn't contain missing values!
RSI_28 doesn't contain missing values!
Interpolating missing values for column: RSI_MA
Column: RSI_MA
Total Missing Values: 93
Correct Predictions: 56
Accuracy: 60.22%
```



Comparison for RSI_MA
Original Data with Missing Values (RSI_MA)
Interpolated Data (RSI_MA)
Clean Data (RSI_MA)
Value
Index



Error Analysis for RSI_MA
Error Magnitude
Margin Of Error
Error Magnitude
Error Index

```
ADX doesn't contain missing values!
MACD doesn't contain missing values!
Close_VIX doesn't contain missing values!
Close_NASDAQ doesn't contain missing values!
Close_DAX doesn't contain missing values!
Close_GOLD doesn't contain missing values!
Close_CPI doesn't contain missing values!
Close_NFP doesn't contain missing values!
Close_T_chomage doesn't contain missing values!
```

Missing Data and Interpolation Process

The column `RSI_MA` initially contained **93 missing values**.

Accuracy of the Interpolation

To evaluate the effectiveness of the interpolation:

- Out of **93 missing values**, **56 values** were correctly predicted within a margin of error of ±2.
- This corresponds to an **accuracy of 60.22%**.

Visual Analysis

1. **Comparison of Original, Interpolated, and Clean Data:**
   - The first plot compares:
     - The **original data** with missing values (red points).
     - The **interpolated data** (green line).
     - The **clean data** without missing values (blue dashed line).
   - **Observation:**
     The interpolated values closely follow the clean data, indicating that the cubic spline interpolation effectively captures the overall trend of the dataset.
2. **Error Analysis:**
   - The second plot presents the **error magnitude** of the interpolated values compared to the clean data:
     - Errors within the margin of ±2 are acceptable.
     - The **green dashed line** represents this margin of error.
     - Larger errors, represented as orange points above the threshold, highlight areas where the interpolation deviates from the clean data.
   - **Observation:**
     While a majority of the errors fall within the acceptable range, a few regions with larger variability in the data result in deviations beyond the threshold.

Conclusion

The cubic spline interpolation method demonstrates **moderate accuracy** with an overall performance of **60.22%** for the column `RSI_MA`. The method successfully preserves the general trend of the data, but deviations occur in areas with higher volatility. This highlights the method's effectiveness while also suggesting that further refinement may be required for datasets with rapid fluctuations or noise.

```
# Create High_tomorrow by shifting the 'High' column by one step:
data['High_tomorrow'] = data['High'].shift(-1)
data = data.dropna(subset=['High_tomorrow'])  # Remove the last rows
with missing values

data.head()
```

```
      Open     High      Low    Close       MA_20       MA_50
MA_100  \
0  1426.19  1462.43  1426.19  1462.42  1390.67455  1421.6443
1411.5234
1  1462.42  1465.47  1455.53  1459.37  1390.95055  1422.2158
1411.5640
2  1459.37  1467.94  1458.99  1466.47  1391.23415  1422.8525
1412.2296
3  1466.47  1466.47  1456.62  1461.89  1391.51600  1423.4127
1412.7910
4  1461.89  1461.89  1451.64  1457.15  1391.78730  1423.9431
1413.6718

      MA_200    Stoch_K    Stoch_D  ...        ADX       MACD
Close_VIX  \
0  1424.9335  52.699795  35.738707  ...  13.206005   6.184123
14.68
1  1427.5495  82.404153  55.412018  ...  14.232887   8.194474
14.56
2  1430.4090  96.274507  77.126151  ...  15.299410  10.242532
13.83
3  1432.8065  93.391723  90.690128  ...  16.086106  11.365056
13.79
4  1434.7605  91.259726  93.641985  ...  16.387387  11.736890
13.62

   Close_NASDAQ  Close_DAX  Close_GOLD  Close_CPI  Close_NFP
Close_T_chomage  \
0    2746.46997    7778.78   1686.8860      229.6     243000
8.0
1    2732.26001    7756.44   1664.0250      229.6     243000
8.0
2    2724.48999    7776.37   1655.7705      229.6     243000
8.0
3    2724.21997    7732.66   1646.9935      229.6     243000
```

```
8.0
4    2718.71997    7695.83   1659.2565      229.6      243000
8.0

   High_tomorrow
0        1465.47
1        1467.94
2        1466.47
3        1461.89
4        1464.73

[5 rows x 22 columns]
```

Creating the `High_tomorrow` Feature

After addressing the missing values in the dataset, we proceeded to prepare the data for analysis by generating a new feature: `High_tomorrow`. This feature represents the **"High" value of the next day**, which is essential for forecasting and predictive modeling.

Steps Performed:

1. **Shifting the `High` Column:**
   - The `High` column was shifted upwards by one step using the `shift(-1)` function.

   - This operation creates a new column, `High_tomorrow`, where each row now contains the **"High" value of the next day**.
2. **Handling Missing Values:**
   - Shifting the column introduces `NaN` values at the last row(s) since there are no subsequent values to populate them.

   - To maintain data integrity, these rows were removed using the `dropna` function, focusing specifically on the `High_tomorrow` column.

Purpose:

By aligning the data in this manner, we ensure that the target variable (`High_tomorrow`) is correctly defined relative to the predictors.

```python
# Create a list with the column names
column_name=['Open', 'High', 'Low', 'Close', 'MA_20', 'MA_50',
'MA_100', 'MA_200', 'Stoch_K', 'Stoch_D', 'RSI_28', 'RSI_MA', 'ADX',
'MACD', 'Close_VIX', 'Close_NASDAQ', 'Close_DAX', 'Close_GOLD',
'Close_CPI', 'Close_NFP', 'Close_T_chomage']
```

2 Handling Outliers

Outliers are data points that deviate significantly from the overall pattern of the data. They can distort statistical measures, such as the mean, and negatively impact model performance.

- **Why It Matters:**
  - Outliers can skew results, making models less accurate or robust.
  - They may mislead conclusions in statistical analysis or visualization.
- **Techniques to Handle Outliers:**
  - **Detection**: Using methods like the interquartile range (IQR) or Z-scores to identify outliers.
  - **Correction**: Replacing extreme values using techniques such as:
    - **Clipping**: Adjusting outliers to the upper or lower bounds.
    - **Transformation**: Applying logarithmic or other transformations to reduce the impact of outliers.

Effectively managing outliers ensures that the analysis focuses on the true trends in the data.

Quartile Method for Detecting and Handling Outliers

In this project, the dataset consists of 21 features spanning a period of five years. Due to the nature of the data, some outliers are present. Detecting and addressing these outliers is a crucial preprocessing step before performing any data analysis, as outliers can distort results. To achieve this, the **quartile method** and the **interquartile range (IQR)** are used to identify and manage these outliers.

What Causes Outliers?

Outliers may occur due to:

- **Extreme values**: Data points significantly higher or lower than the rest of the dataset.

- **Errors**: Mistakes in data collection or inaccuracies caused by experimental limitations.

Detecting Outliers

The project defines an interval to detect outliers. Any data point that falls outside this interval is classified as an outlier. A widely-used statistical approach for defining this interval is the **quartile method**.

Quartiles:

- **First Quartile (Q1)**: The middle value of the first half of the ranked data.

- **Second Quartile (Q2)**: The median of the entire ranked dataset.

- **Third Quartile (Q3)**: The middle value of the second half of the ranked data.

Interquartile Range (IQR):

The **IQR** is the difference between the third quartile (Q3) and the first quartile (Q1):
IQR = Q3 - Q1

Using the IQR, the outlier detection interval is defined as:

- **Upper boundary**: Q3 + 1.5 × IQR

- **Lower boundary**: Q1 – 1.5 × IQR

Any data points that fall **outside this range** are considered outliers.

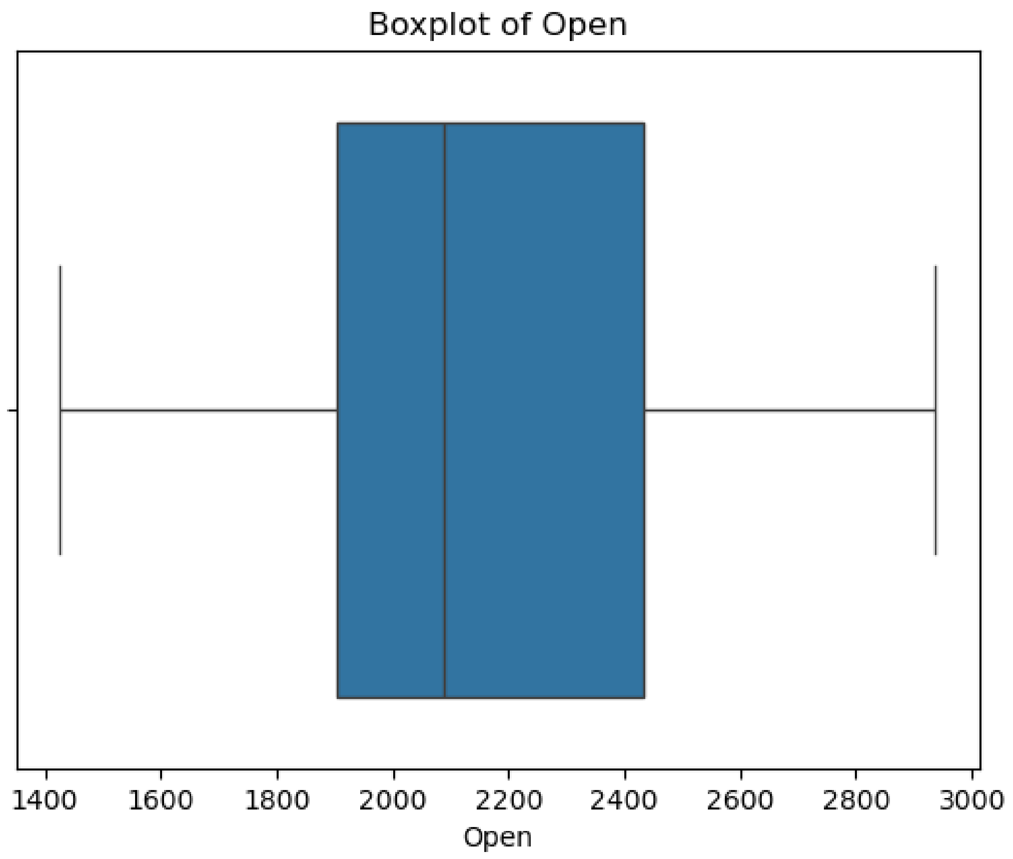
In this project, if more than 90 outliers are detected, they are handled rather than removed. Deleting outliers might result in the loss of valuable information, which could affect the analysis. Instead, the **clip method** is applied to manage them.

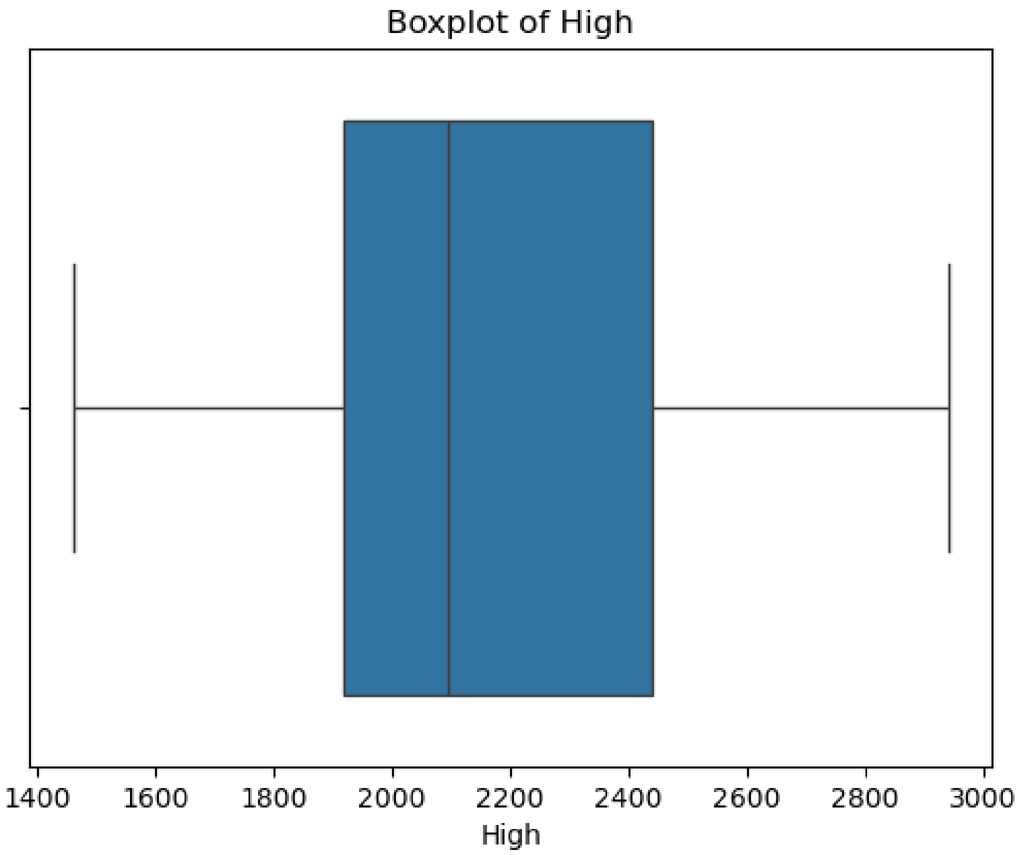
Clip Method:

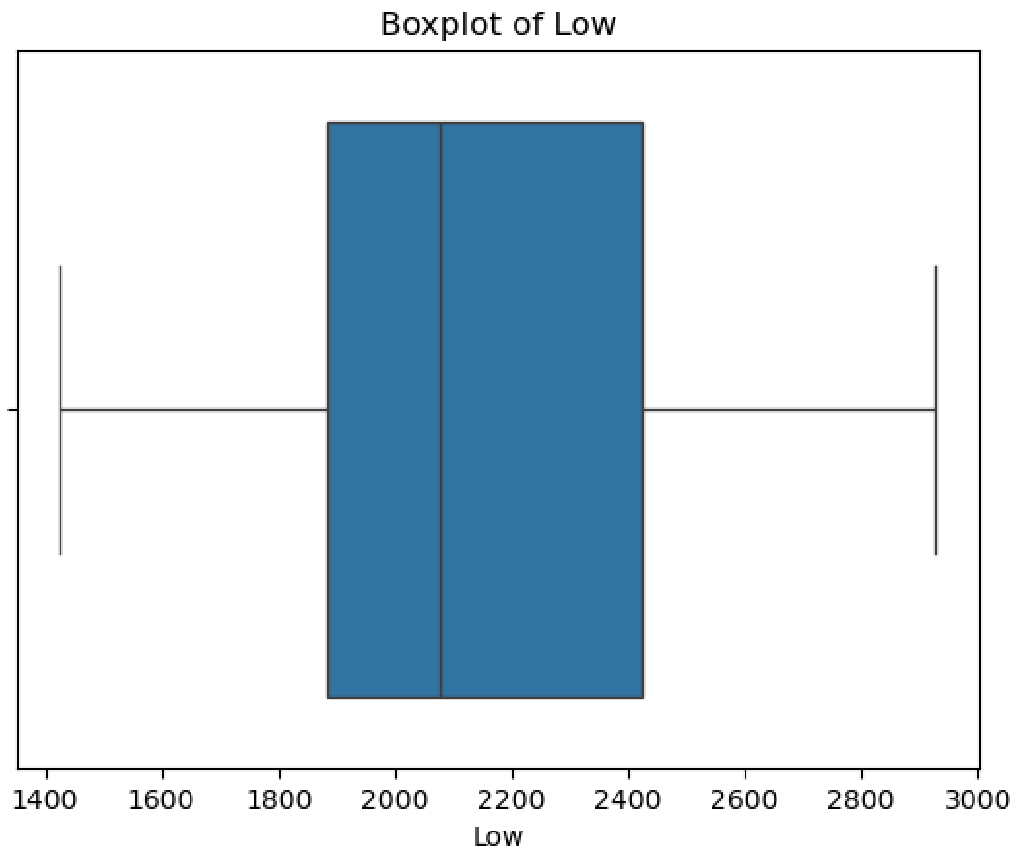
- Outliers smaller than the lower boundary are replaced by the **lower boundary**.

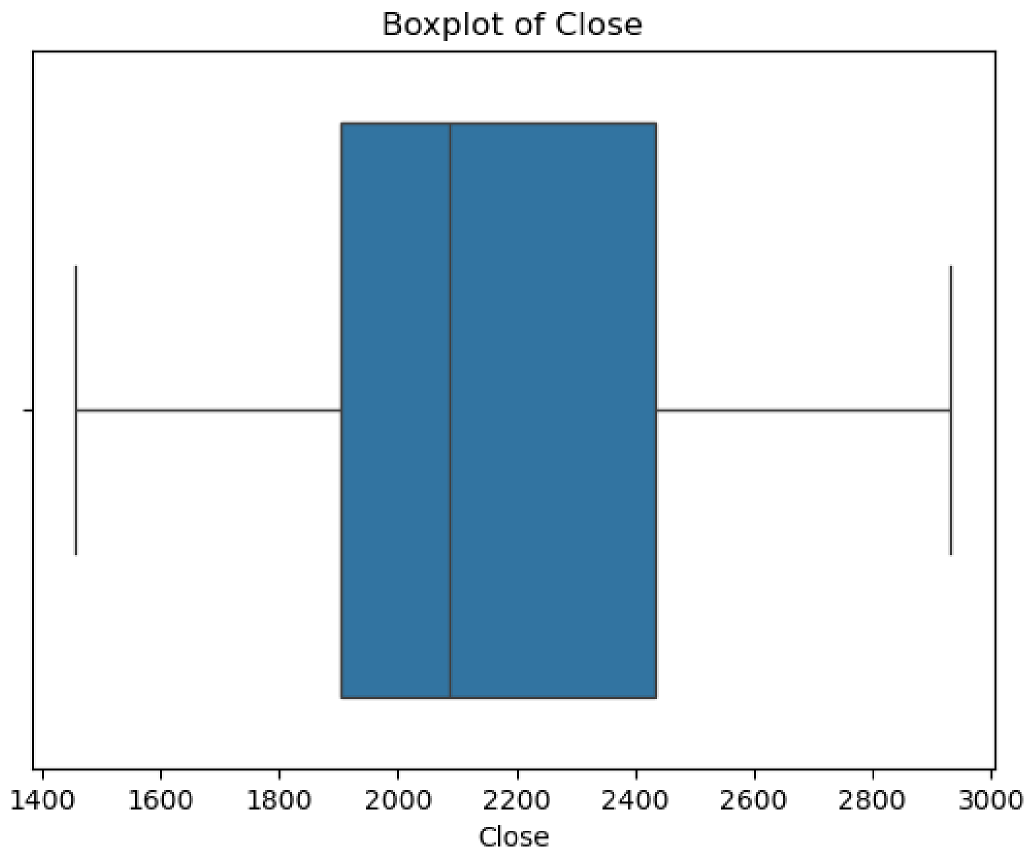
- Outliers larger than the upper boundary are replaced by the **upper boundary**. [15].

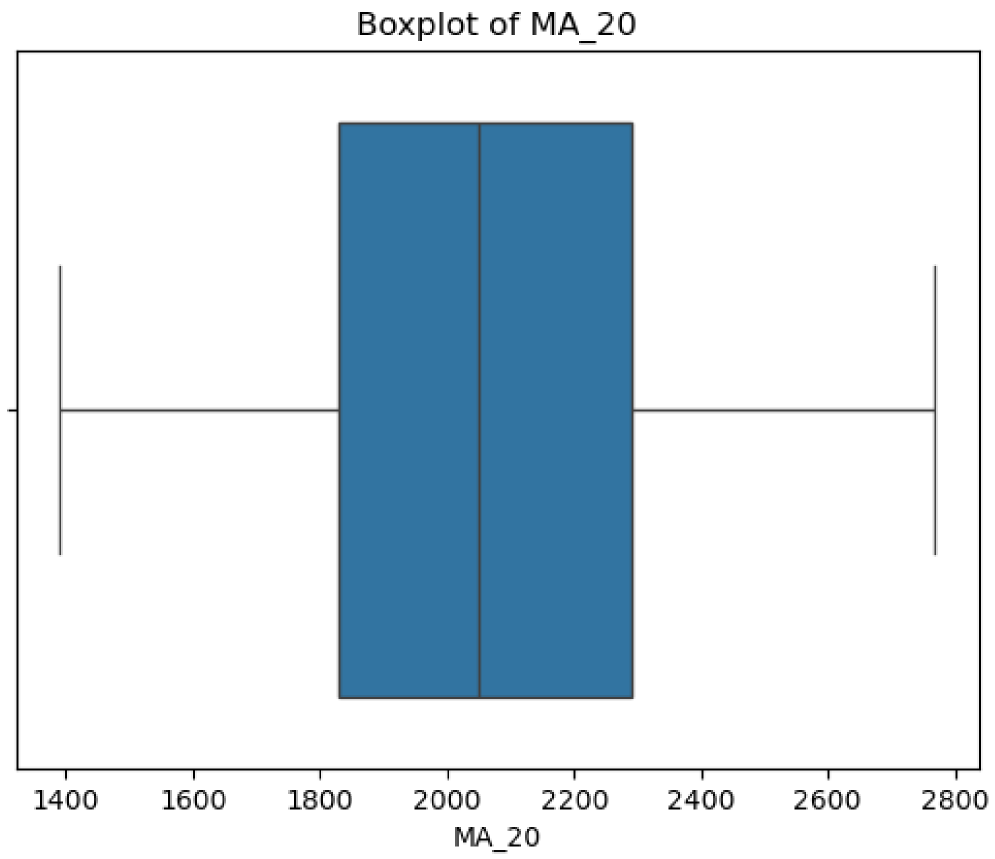
```python
# Visualiser les outliers dans toutes les colonnes

for i in column_name:
    sns.boxplot(x=data[i])
    plt.title(f'Boxplot of {i}')
    plt.show()
```

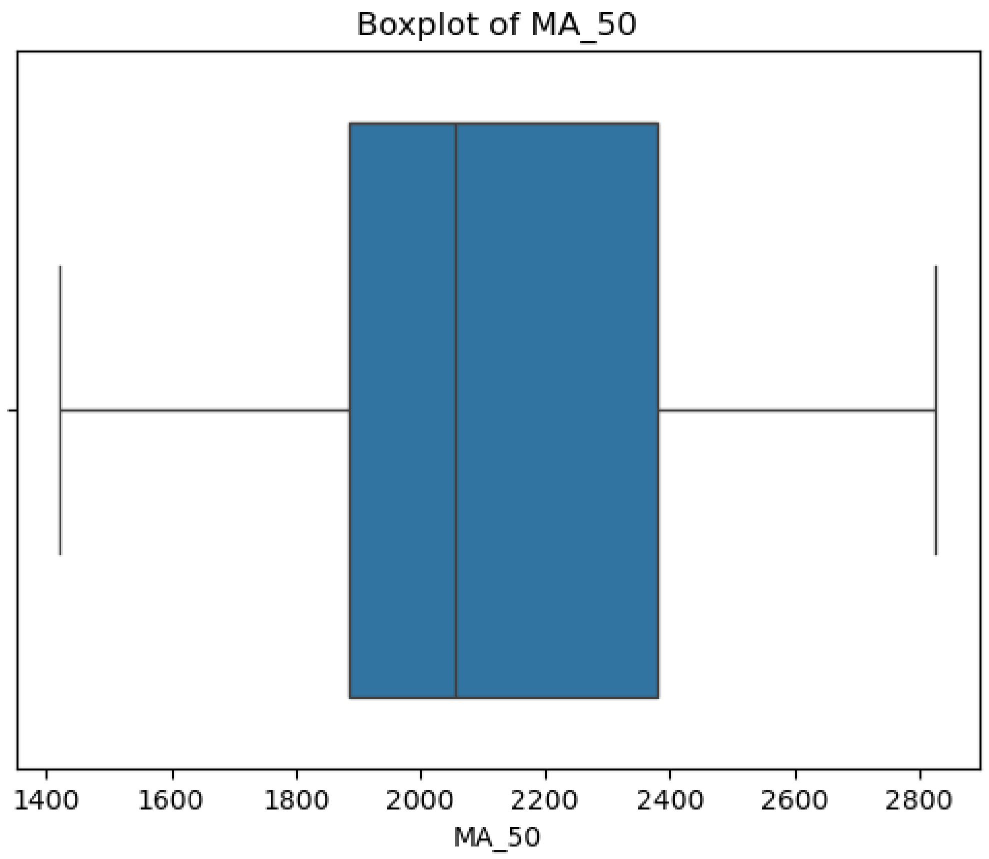

Boxplot of Open
1400
1600
1800
2000
2200
2400
2600
2800
3000
Open

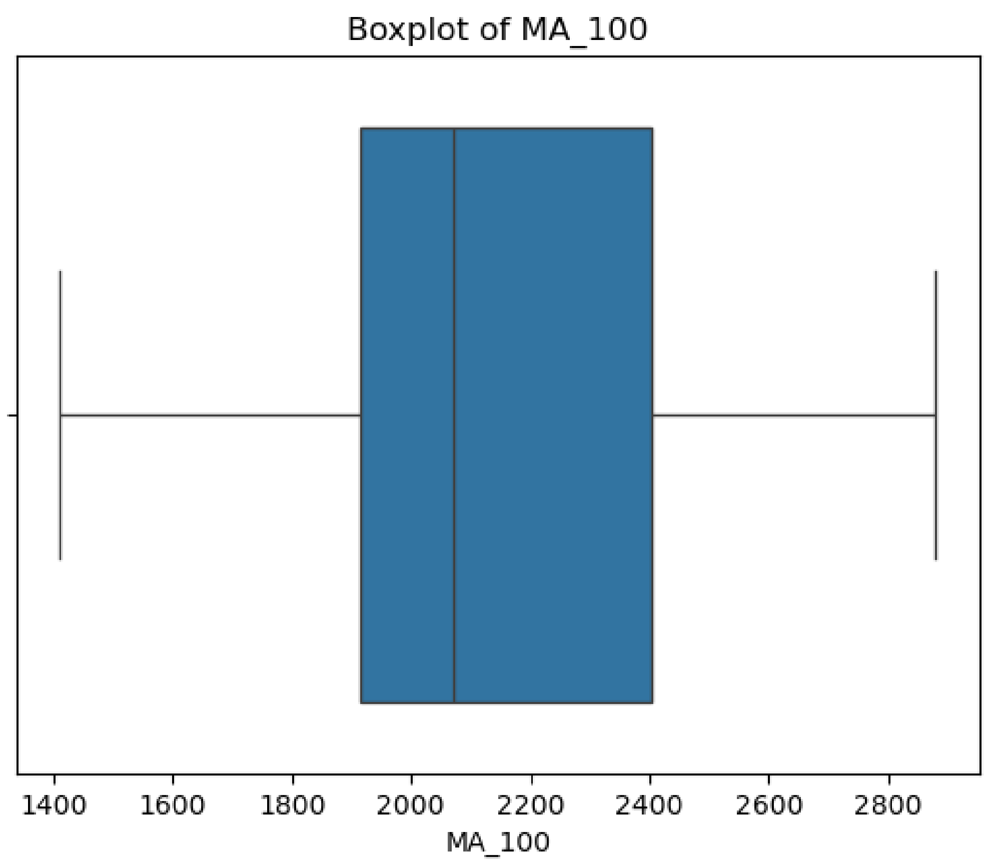

Boxplot of High
1400
1600
1800
2000
2200
2400
2600
2800
3000
High

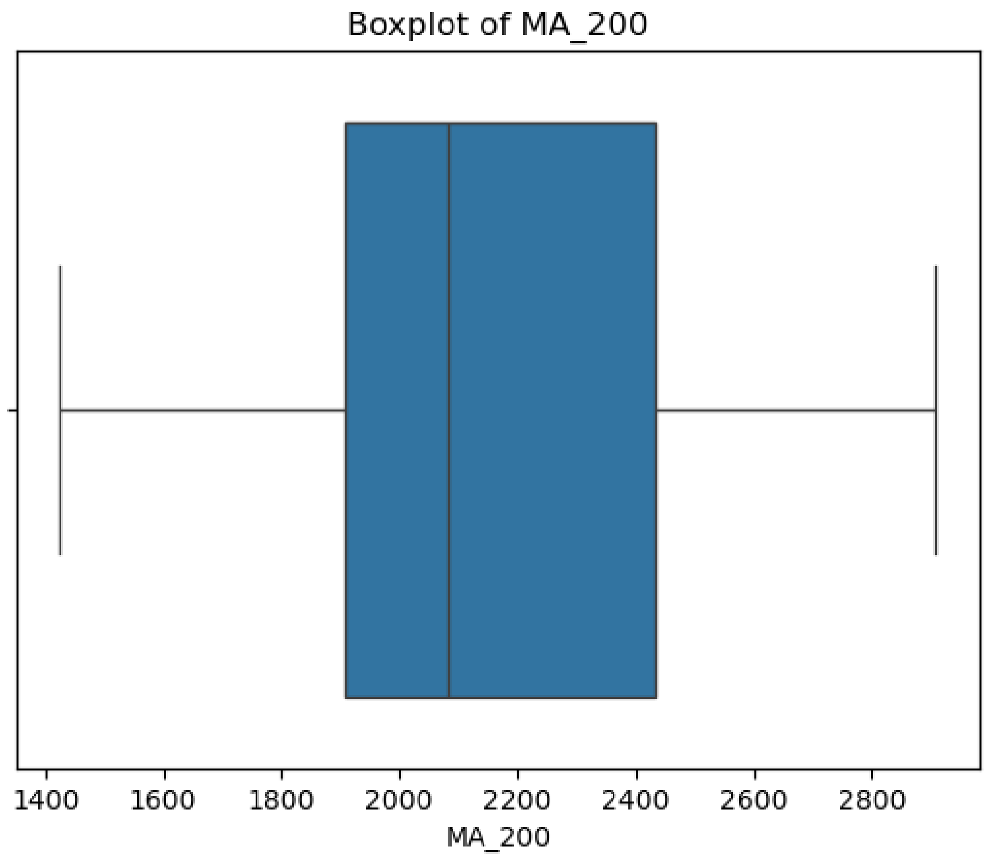

Boxplot of Low
1400
1600
1800
2000
2200
2400
2600
2800
3000
Low

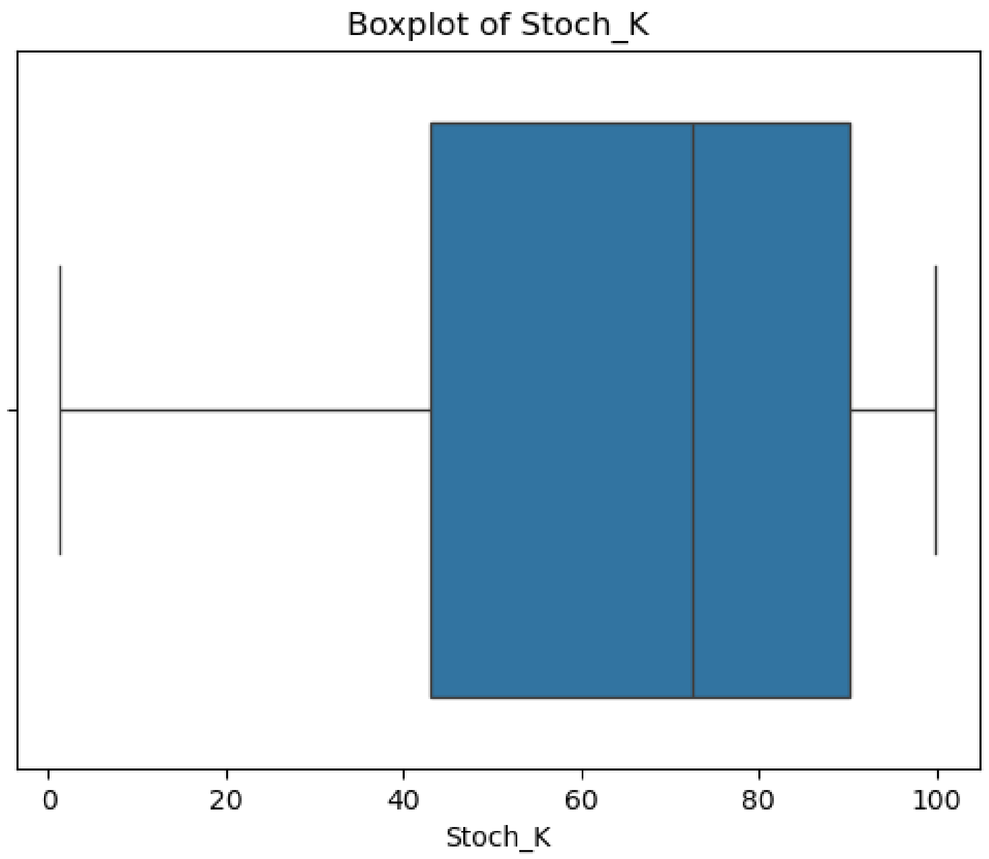

Boxplot of Close
1400
1600
1800
2000
2200
2400
2600
2800
3000
Close

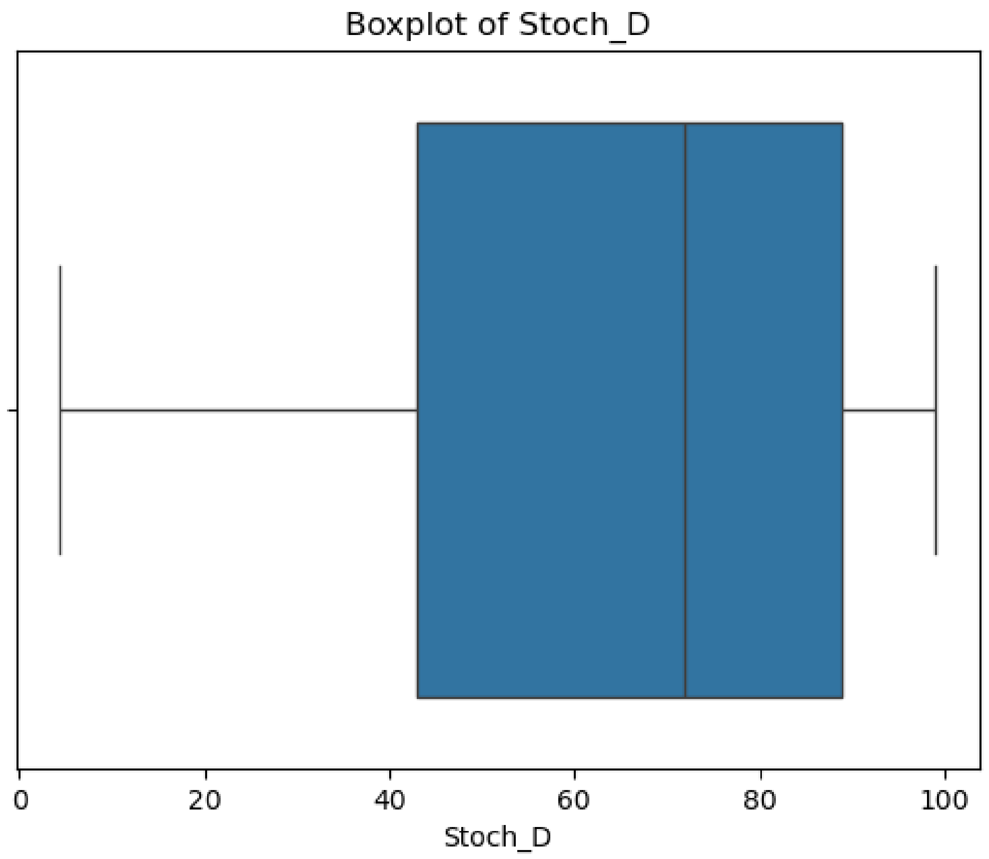

Boxplot of MA_20
1400
1600
1800
2000
2200
2400
2600
2800
MA_20

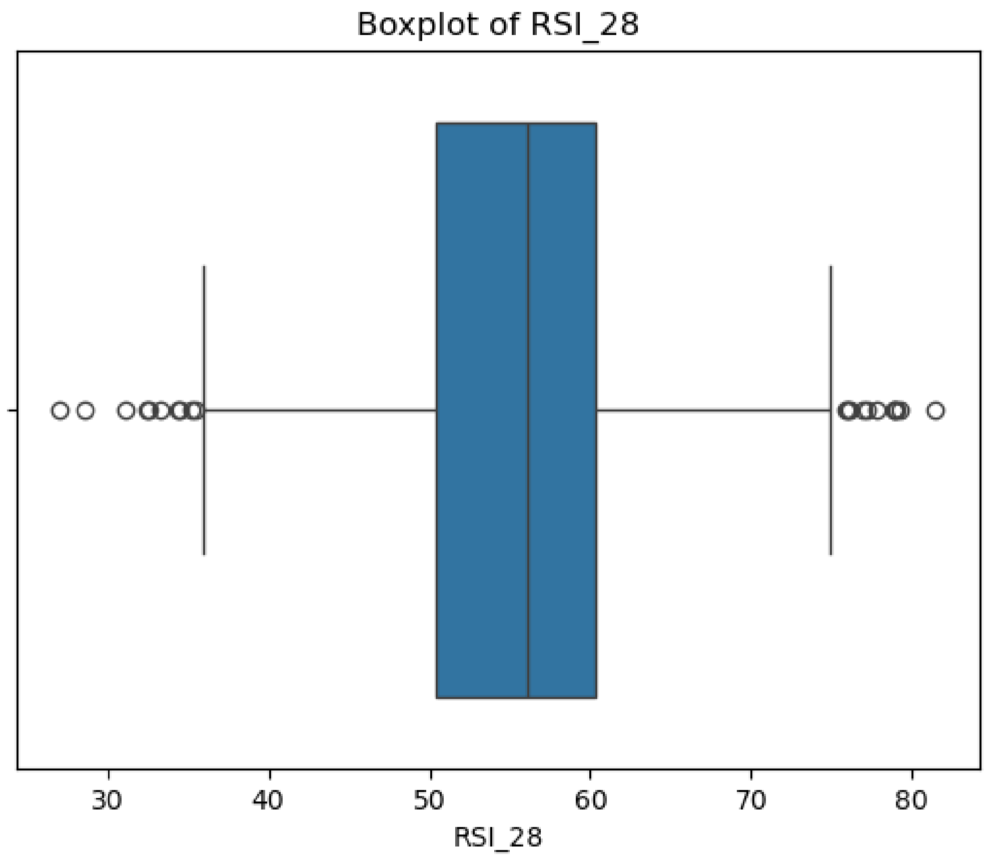

Boxplot of MA_50
1400
1600
1800
2000
2200
2400
2600
2800
MA_50

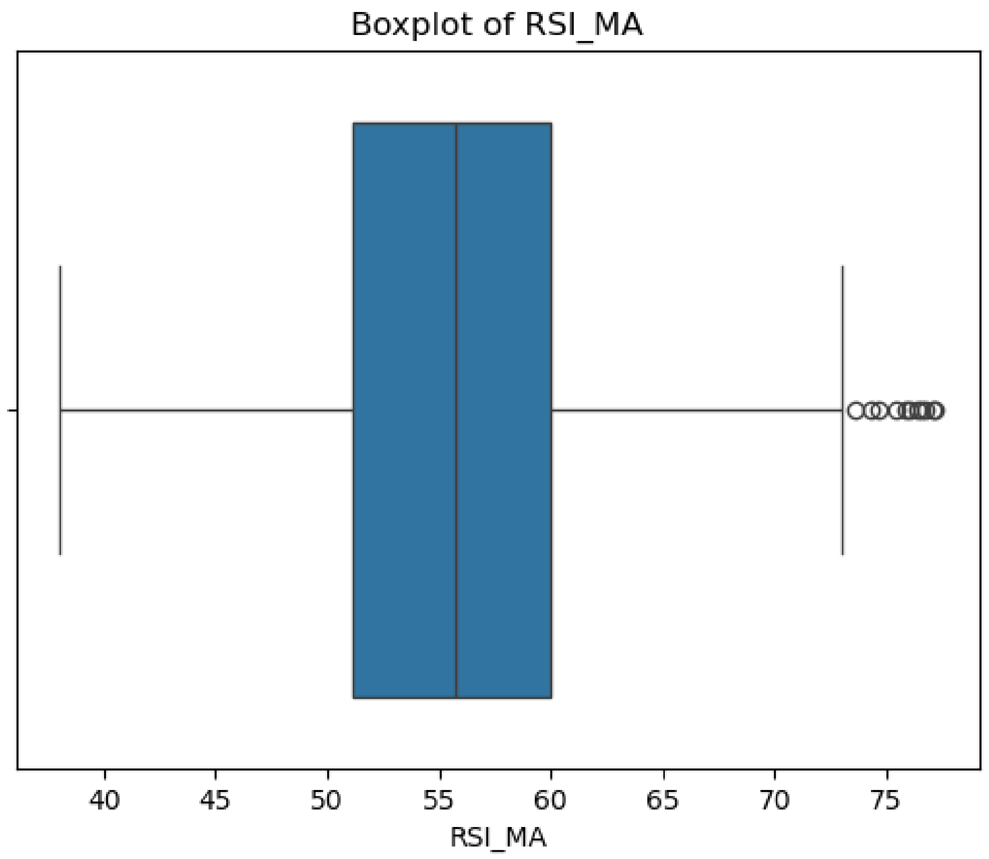

Boxplot of MA_100
1400
1600
1800
2000
2200
2400
2600
2800
MA_100

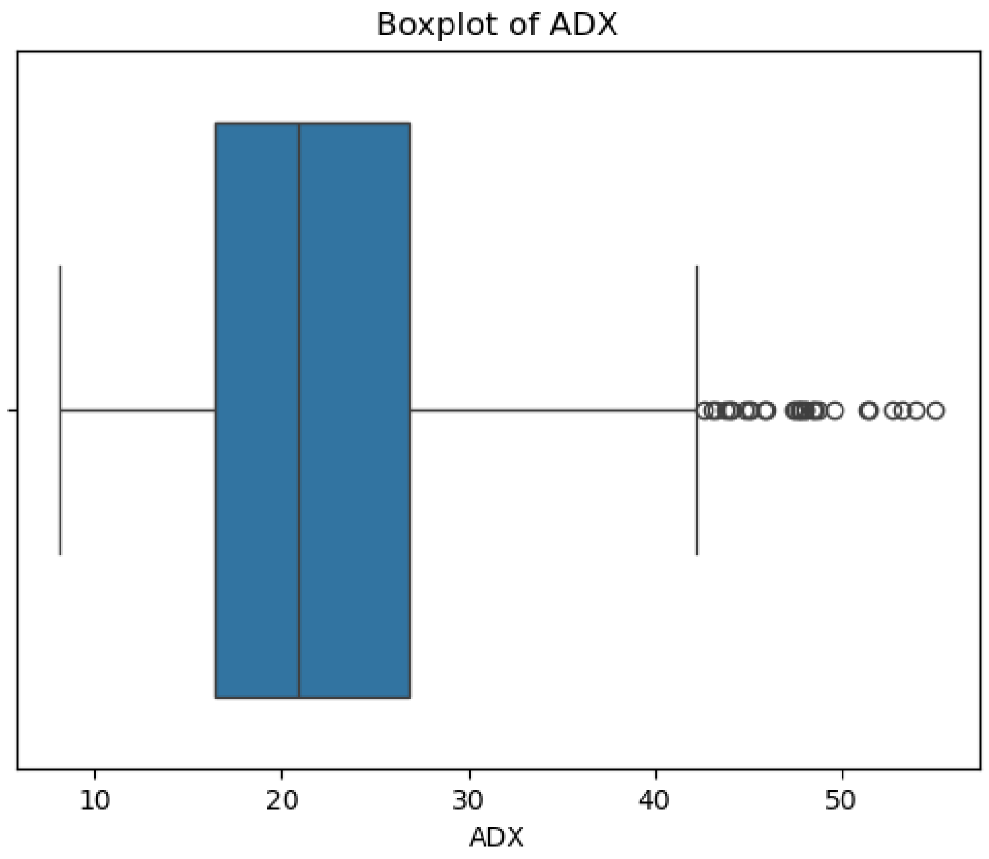

Boxplot of MA_200
1400
1600
1800
2000
2200
2400
2600
2800
MA_200

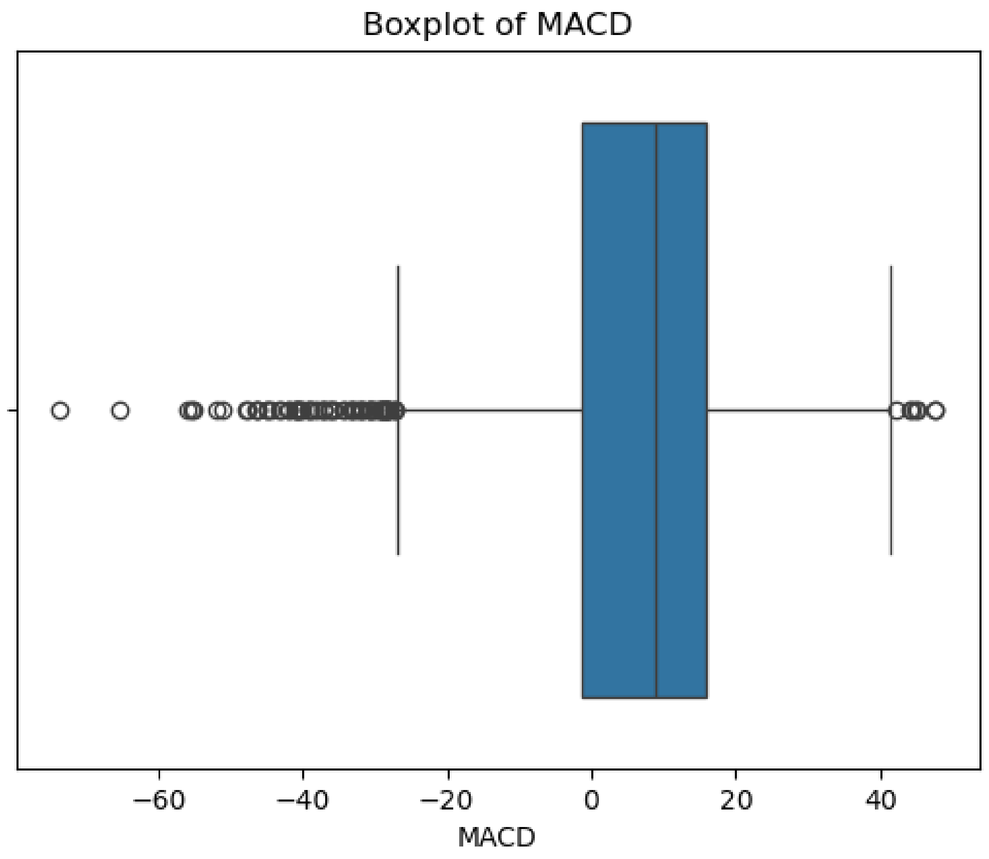

Boxplot of Stoch_K
0
20
40
60
80
100
Stoch_K

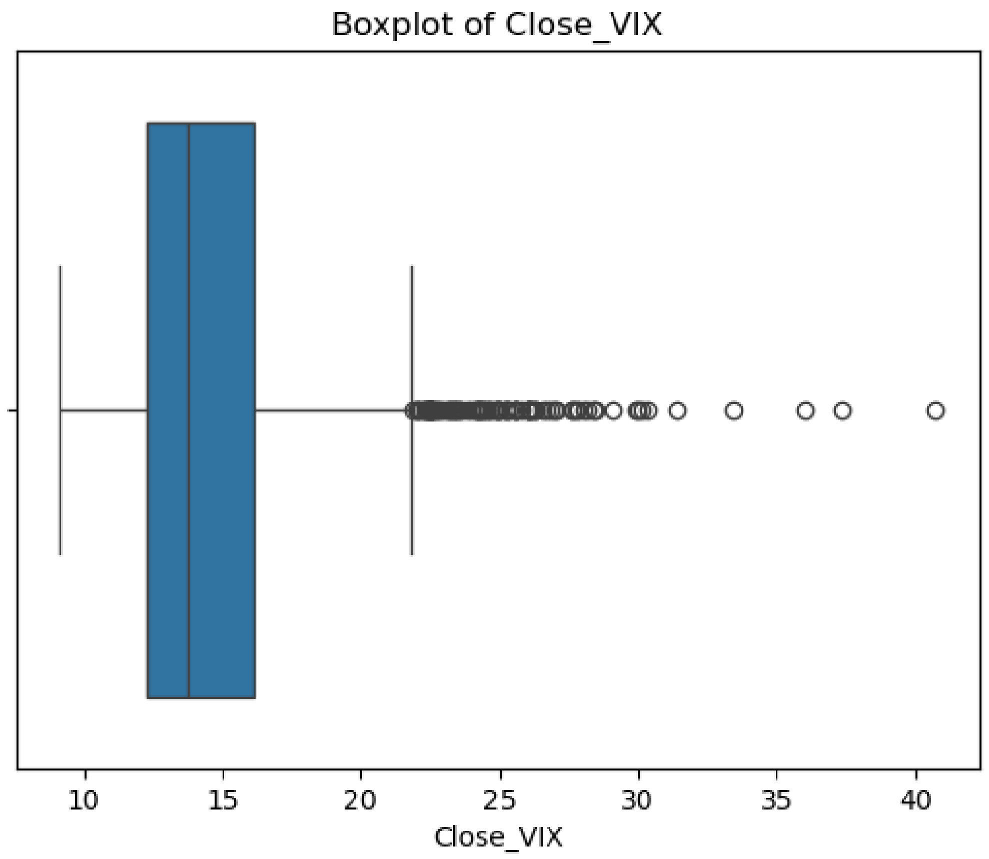

Boxplot of Stoch_D
0
20
40
60
80
100
Stoch_D

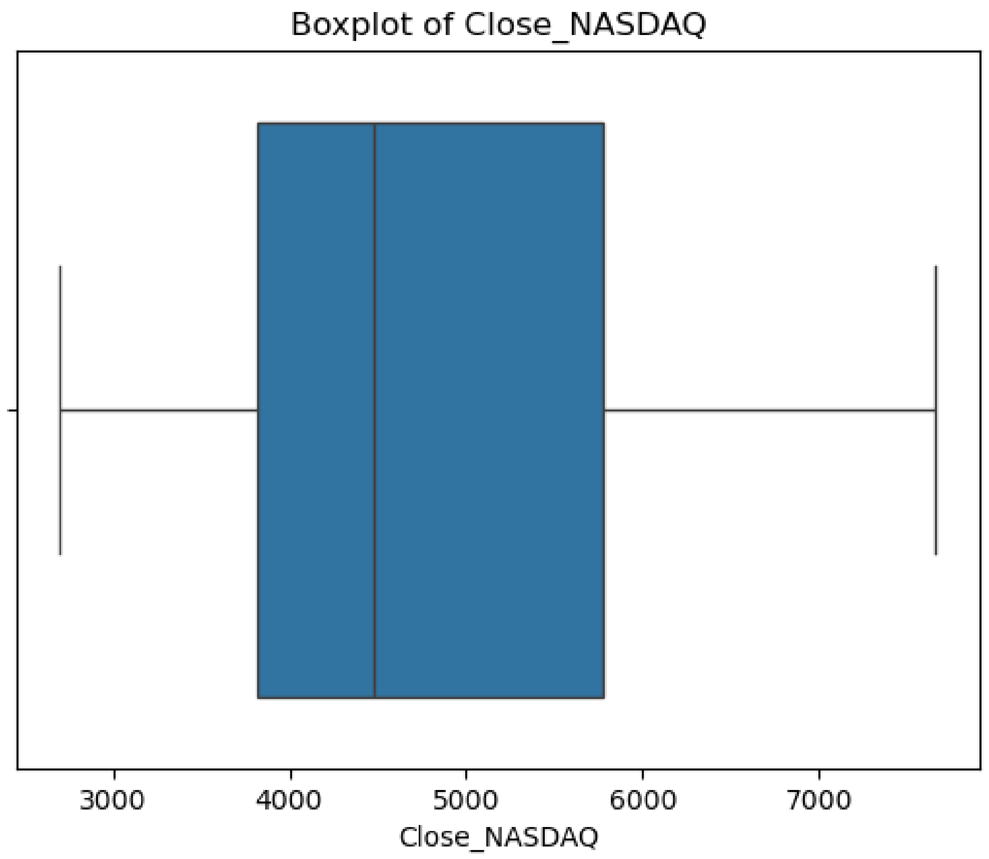

Boxplot of RSI_28
30
40
50
60
70
80
RSI_28

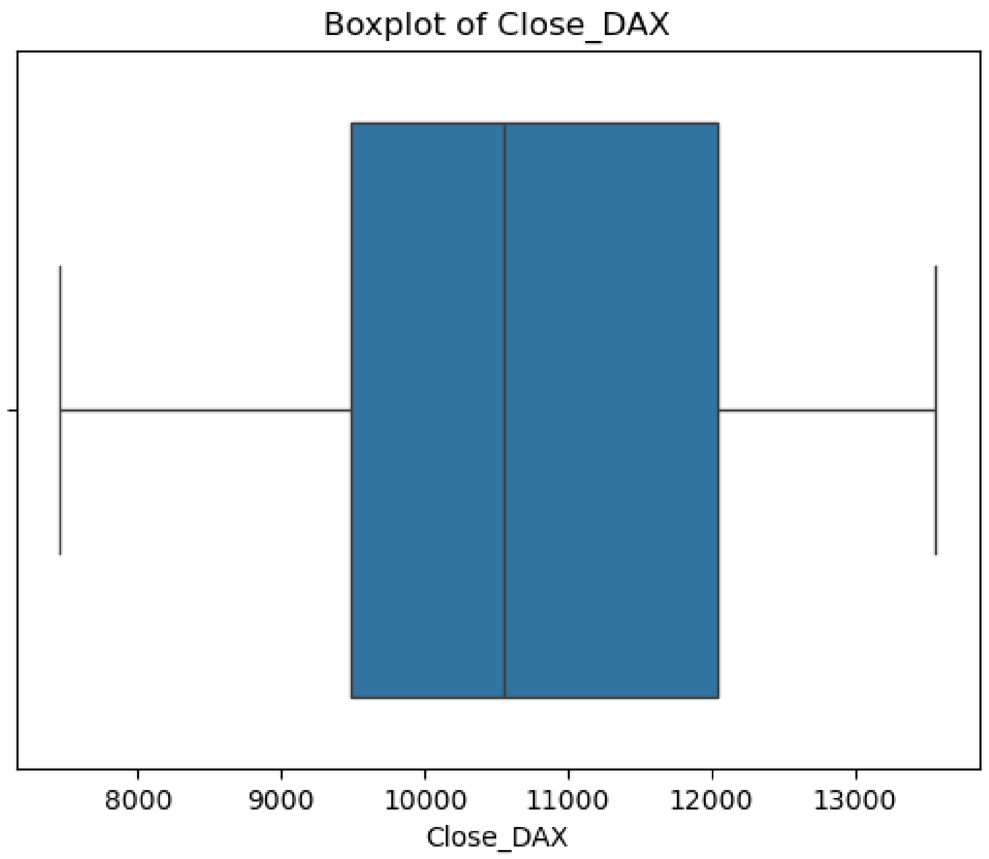

Boxplot of RSI_MA
40
45
50
55
60
65
70
75
RSI_MA

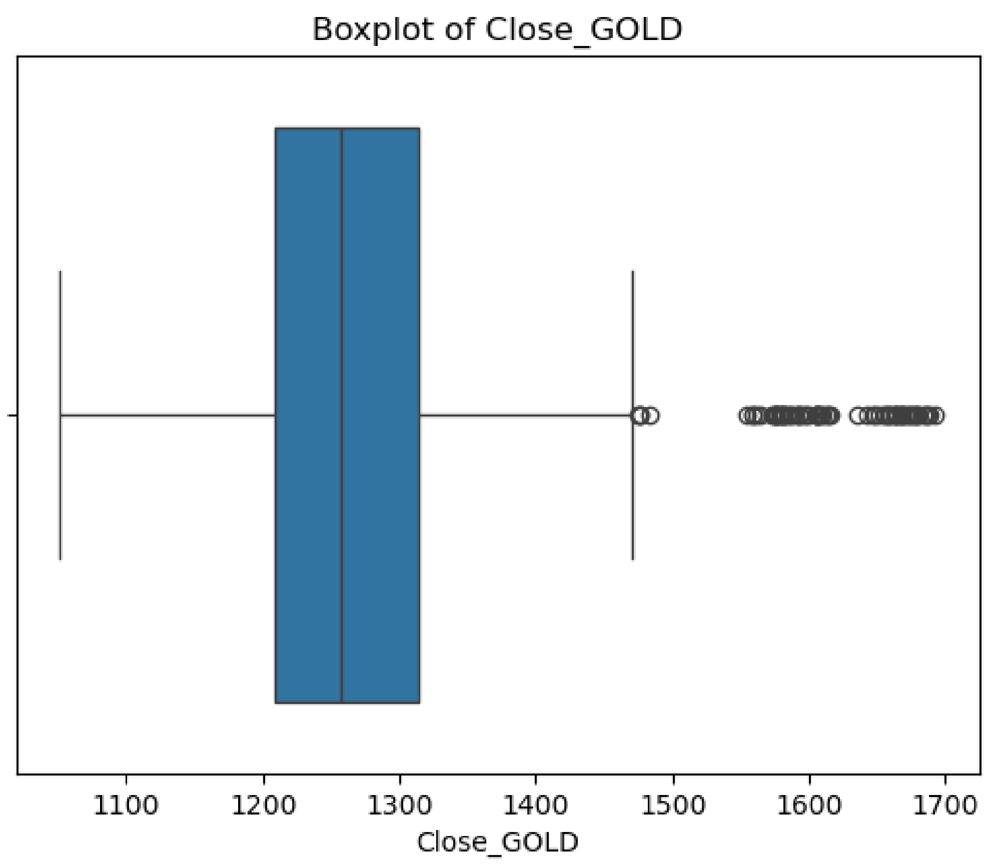

Boxplot of ADX
10
20
30
40
50
ADX


Boxplot of MACD
−60
−40
−20
0
20
40
MACD

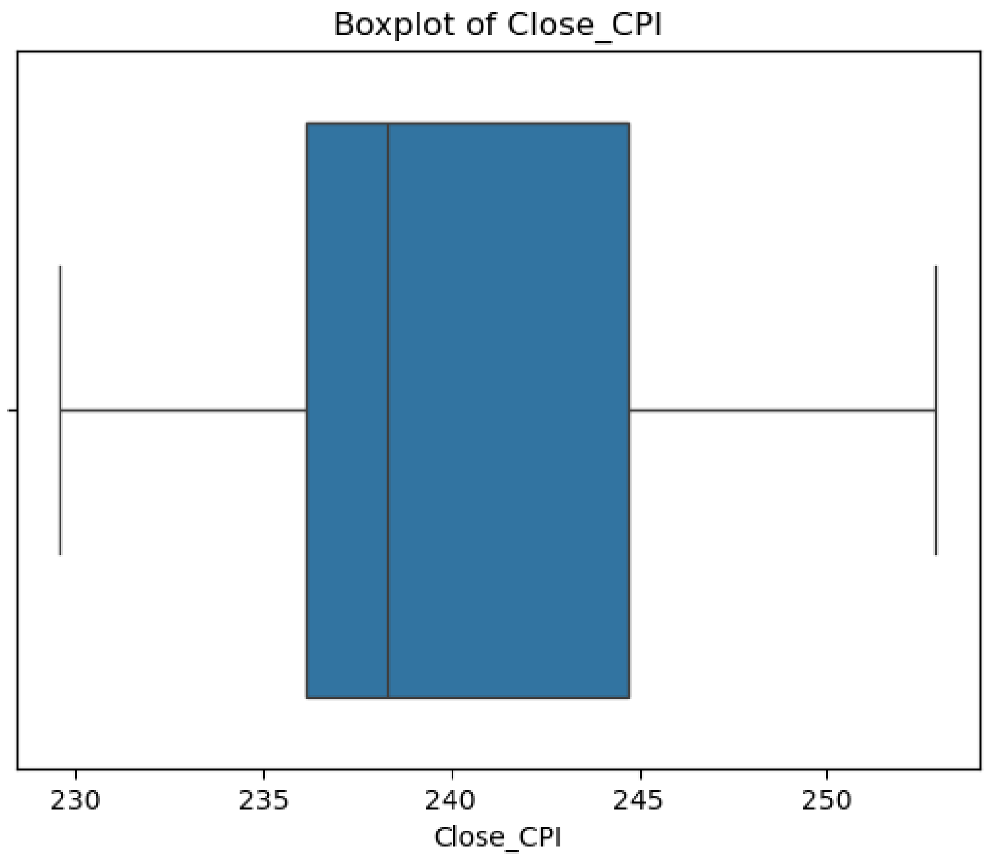

Boxplot of Close_VIX
10
15
20
25
30
35
40
Close_VIX

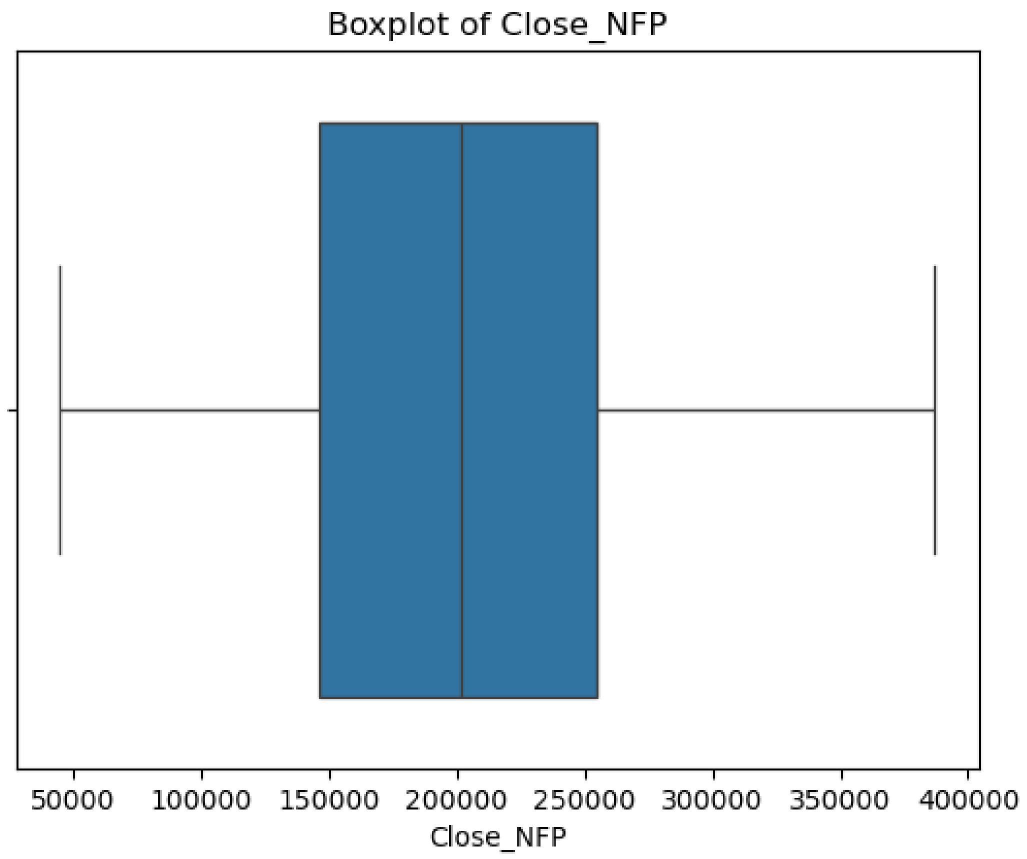

Boxplot of Close_NASDAQ
3000
4000
5000
6000
7000
Close_NASDAQ

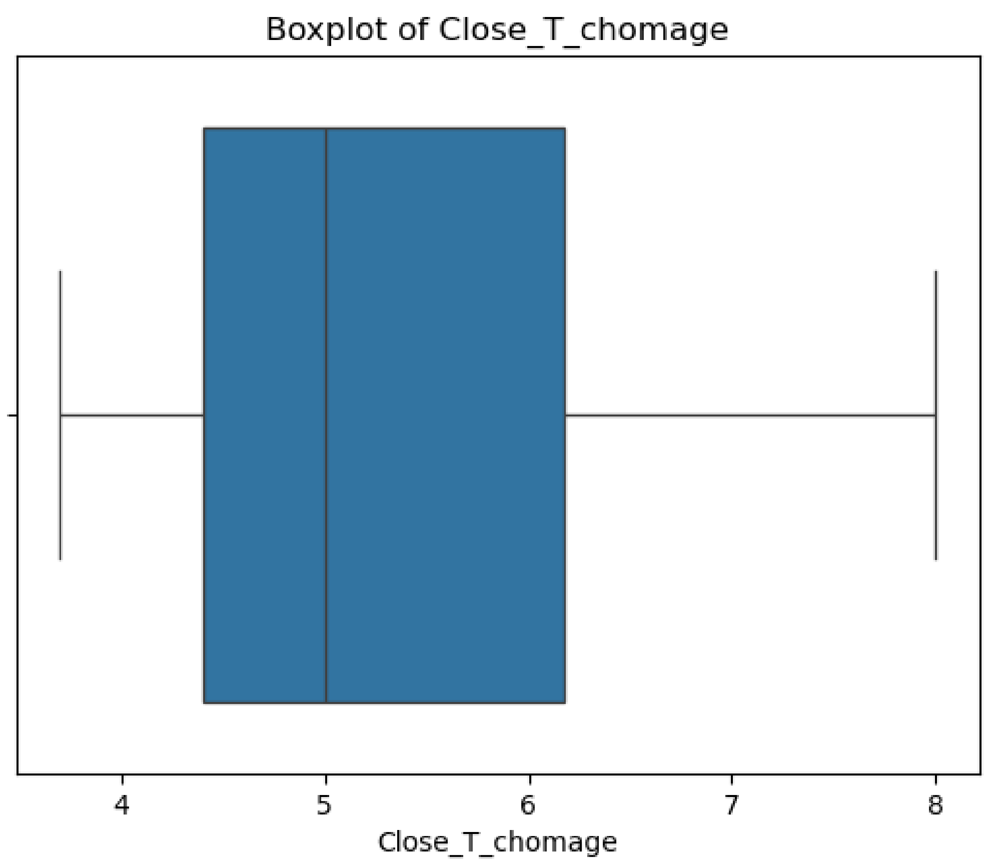

Boxplot of Close_DAX
8000
9000
10000
11000
12000
13000
Close_DAX



Boxplot of Close_GOLD
1100
1200
1300
1400
1500
1600
1700
Close_GOLD


Boxplot of Close_CPI
230
235
240
245
250
Close_CPI


Boxplot of Close_NFP
50000
100000
150000
200000
250000
300000
350000
400000
Close_NFP

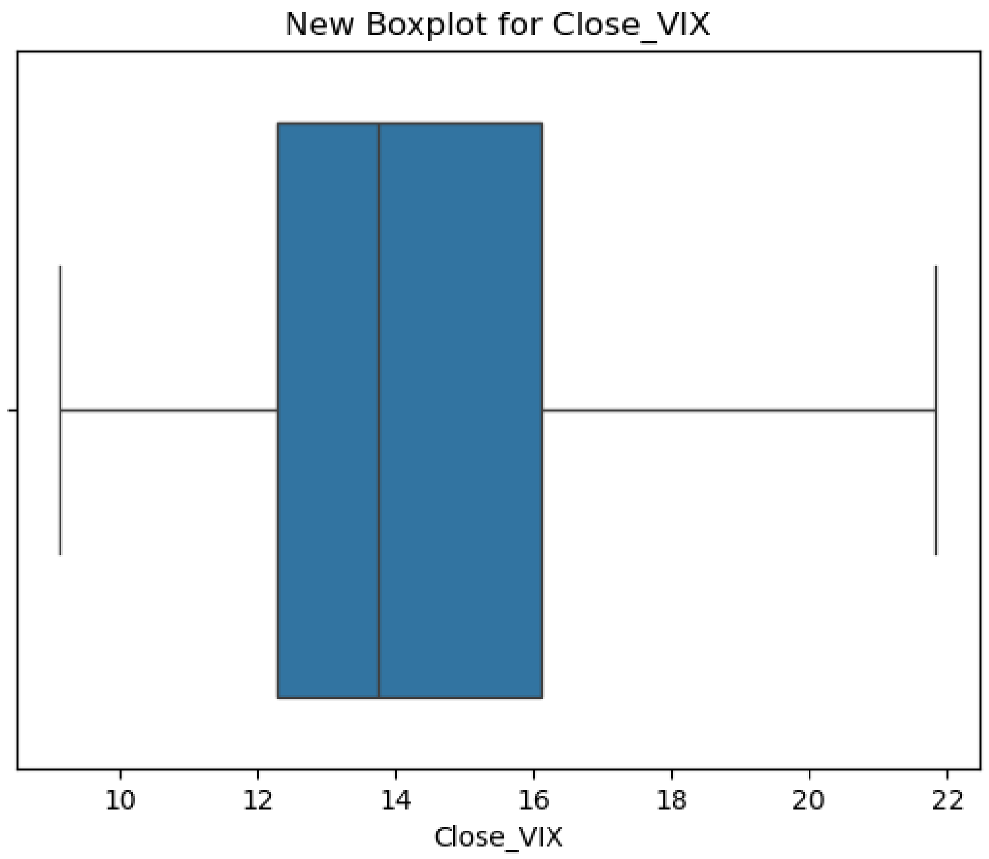


Visualizing Outliers Using Boxplots

To determine the presence of outliers in the dataset, we utilized **boxplots** for each feature. Boxplots are an effective graphical representation for identifying outliers, as they highlight data points that fall outside the typical range.

```
# Calculate the number of outliers in each column and apply
corrections
for i in column_name:
    # Calculate Q1 (25th percentile) and Q3 (75th percentile)
    Q1 = data[i].quantile(0.25)
    Q3 = data[i].quantile(0.75)

    # Calculate the IQR (interquartile range)
    IQR = Q3 - Q1

    # Define the bounds to detect outliers
    lower_bound = Q1 - 1.5 * IQR
    upper_bound = Q3 + 1.5 * IQR

    # Identify the outliers
    outliers = data[(data[i] < lower_bound) | (data[i] > upper_bound)]
    print(f"Number of outliers in column {i}: {len(outliers)}")
    # If the number of outliers is greater than or equal to 90, apply
the clip method to replace outliers
    if len(outliers) >= 90:
        # Apply the clip method to replace values below the lower
bound and above the upper bound
        data[i] = data[i].clip(lower=lower_bound, upper=upper_bound)
        print(f"Outliers in column {i} have been replaced by the lower
and upper bounds.")

        # Visualize the new Boxplot
        sns.boxplot(x=data[i])
        plt.title(f'New Boxplot for {i}')
        plt.show()
```

```
Number of outliers in column Open: 0
Number of outliers in column High: 0
Number of outliers in column Low: 0
Number of outliers in column Close: 0
Number of outliers in column MA_20: 0
Number of outliers in column MA_50: 0
Number of outliers in column MA_100: 0
Number of outliers in column MA_200: 0
Number of outliers in column Stoch_K: 0
Number of outliers in column Stoch_D: 0
Number of outliers in column RSI_28: 21
Number of outliers in column RSI_MA: 11
Number of outliers in column ADX: 28
Number of outliers in column MACD: 79
Number of outliers in column Close_VIX: 92
Outliers in column Close_VIX have been replaced by the lower and upper
bounds.
```



```
Number of outliers in column Close_NASDAQ: 0
Number of outliers in column Close_DAX: 0
Number of outliers in column Close_GOLD: 72
Number of outliers in column Close_CPI: 0
Number of outliers in column Close_NFP: 0
Number of outliers in column Close_T_chomage: 0
```

Outlier Detection and Correction Process

The code systematically detects and corrects outliers in each column of the dataset using the **interquartile range (IQR)** method. The main steps are as follows:

1. **Calculation of Quartiles and IQR:**
   - For each column, the first quartile (**Q1**) and third quartile (**Q3**) are calculated.
   - The **interquartile range (IQR)**, which represents the spread of the middle 50% of the data, is computed as:
     IQR = Q3 − Q1
2. **Defining Outlier Bounds:**
   - The lower and upper bounds for detecting outliers are defined as:
     - **Lower Bound**: Q1 − 1.5 × IQR

        - **Upper Bound**: Q3 + 1.5 × IQR

3. **Identifying Outliers:**
    - Data points that fall below the lower bound or above the upper bound are classified as outliers.
    - The total number of outliers in each column is printed.
4. **Correcting Outliers:**
    - If a column contains **90 or more outliers**, the **clip method** is applied to replace extreme values:
        - Values below the lower bound are set to the lower bound.
        - Values above the upper bound are set to the upper bound.
5. **Visualizing Corrected Data:**
    - After correction, a **boxplot** is generated to verify that the outliers have been adjusted and the column's distribution remains reasonable.

After applying the **IQR-based outlier detection method** to all columns in the dataset, the following results were obtained:

- Columns such as **Open, High, Low, Close, MA_20, MA_50, MA_100, MA_200**, and others showed **no outliers**.
- The columns with detected outliers include:
    - **RSI_28**: 21 outliers
    - **RSI_MA**: 11 outliers
    - **ADX**: 28 outliers
    - **MACD**: 79 outliers
    - **Close_VIX**: 92 outliers
    - **Close_GOLD**: 72 outliers

Correction Applied

For the column **Close_VIX**, where 92 outliers were identified, the **clip method** was applied.

Visualization of Corrected Data

The updated **boxplot** for the column `Close_VIX` (shown above) confirms that the outliers have been successfully handled:

- The whiskers now reflect the adjusted range of values.
- No extreme outliers are visible beyond the whiskers.

3) Data Transformation

Data transformation involves converting data into a suitable format or scale for analysis and modeling. This step improves the performance and interpretability of models.

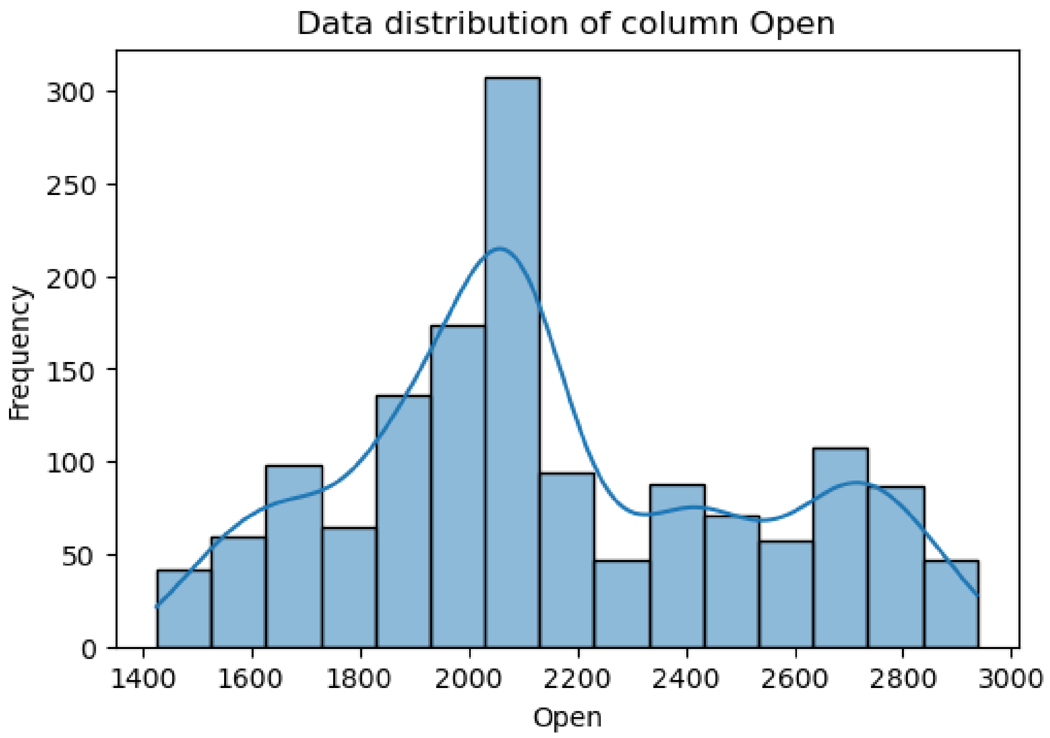
- **Why It Matters:**

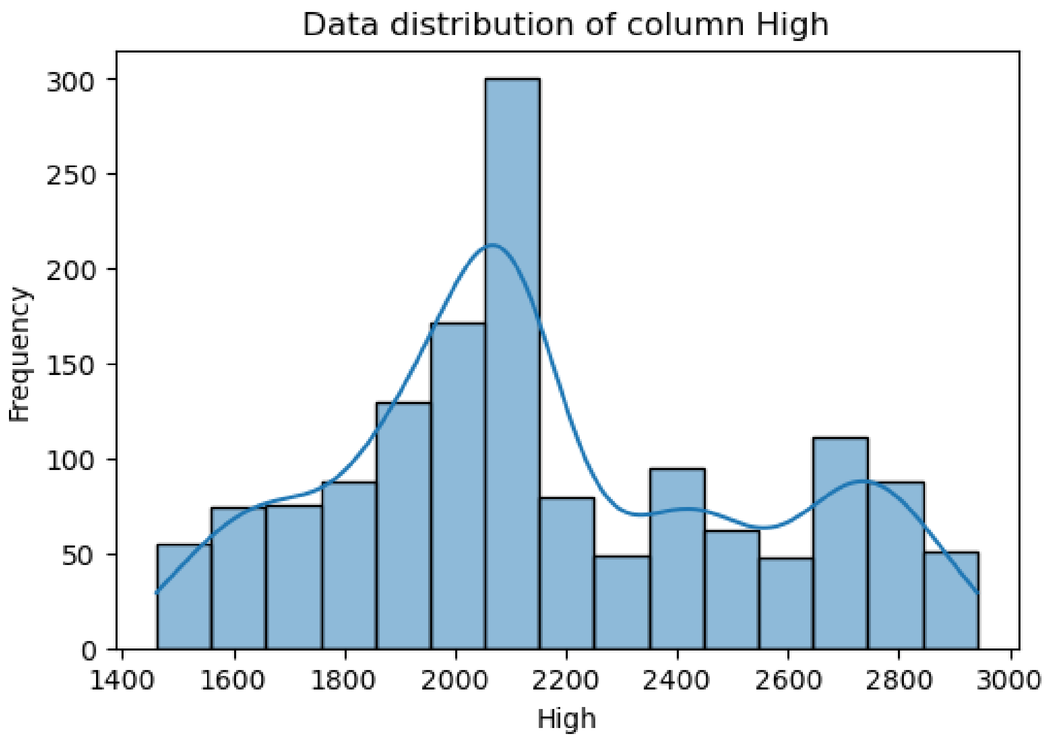
    - It ensures that all features are on a comparable scale, especially for distance-based algorithms.
    - Transformation can help normalize data distributions, making patterns easier to identify.
    - Satisfies key assumptions for regression and statistical analysis:
        - **Linearity**: Ensures relationships between variables are more linear.
        - **Homoscedasticity**: Stabilizes variance across predictor variables.
        - **Independence of Variables**: Reduces irregular patterns or outliers that may interfere with assumptions.
        - **Normality of Residuals**: Improves data distribution, contributing to normally distributed residuals.
- **Common Techniques:**
    - **Square Root Transformation**:
        - Reduces skewness for moderately skewed data by taking the square root of values.
        - It compresses large values while maintaining the relative order of the data.
    - **Log Transformation**:
        - Reduces skewness in highly skewed data by applying the natural logarithm (log).
        - This method is particularly effective for data with a wide range of values, as it compresses large values more than small ones.
    - **Box-Cox Transformation**:
        - A more general power transformation that can stabilize variance and make data more normally distributed.
        - It is defined as:

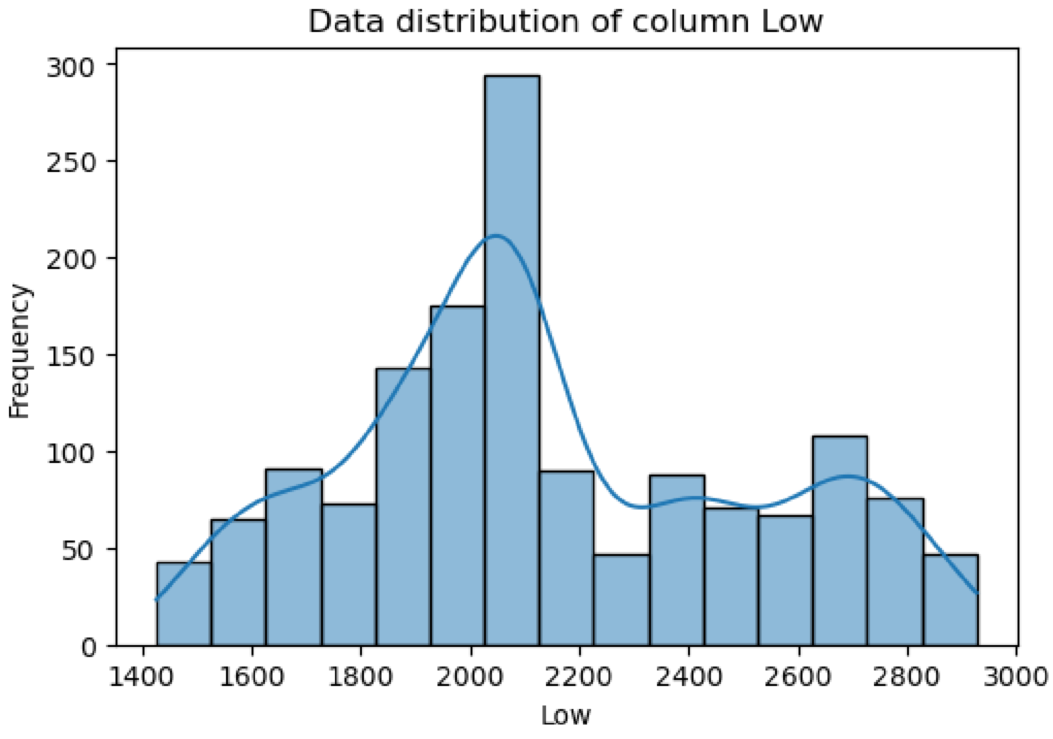
$$y(\lambda)=\left\{\begin{array}{ll}\dfrac{y^{\lambda}-1}{\lambda} & \text{if }\lambda\neq 0\\ \text{¿}\log(y) & \text{if }\lambda=0\end{array}\right)$$

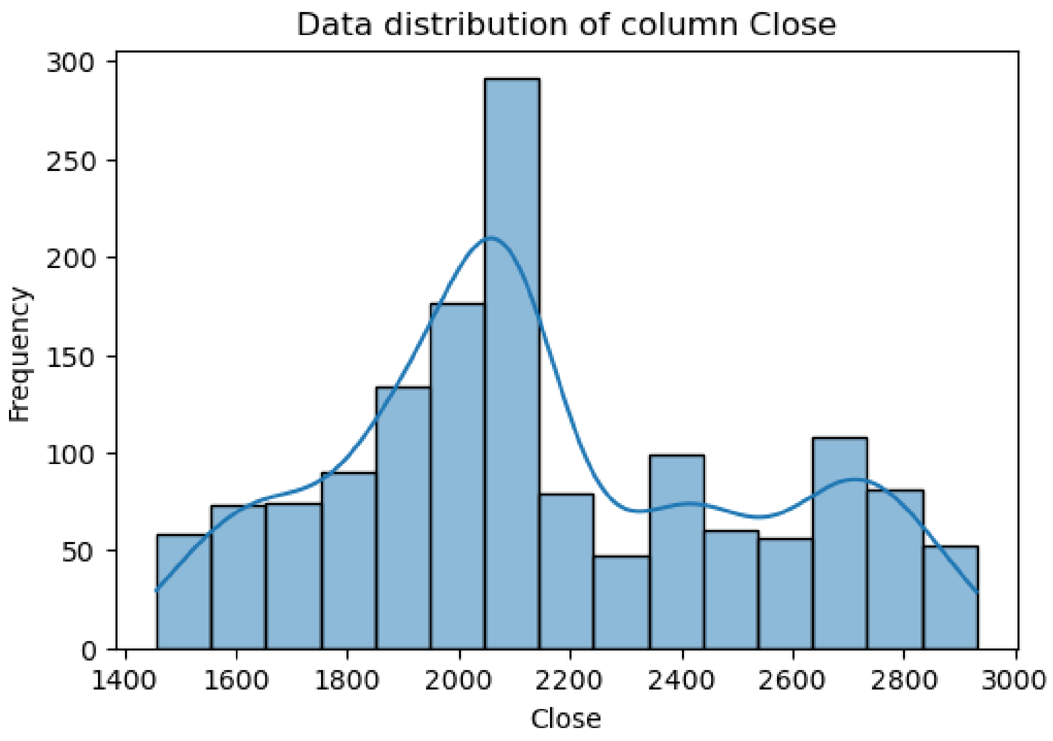
- The Box-Cox method finds the optimal value of (\lambda) to transform the data, ensuring minimal skewness.

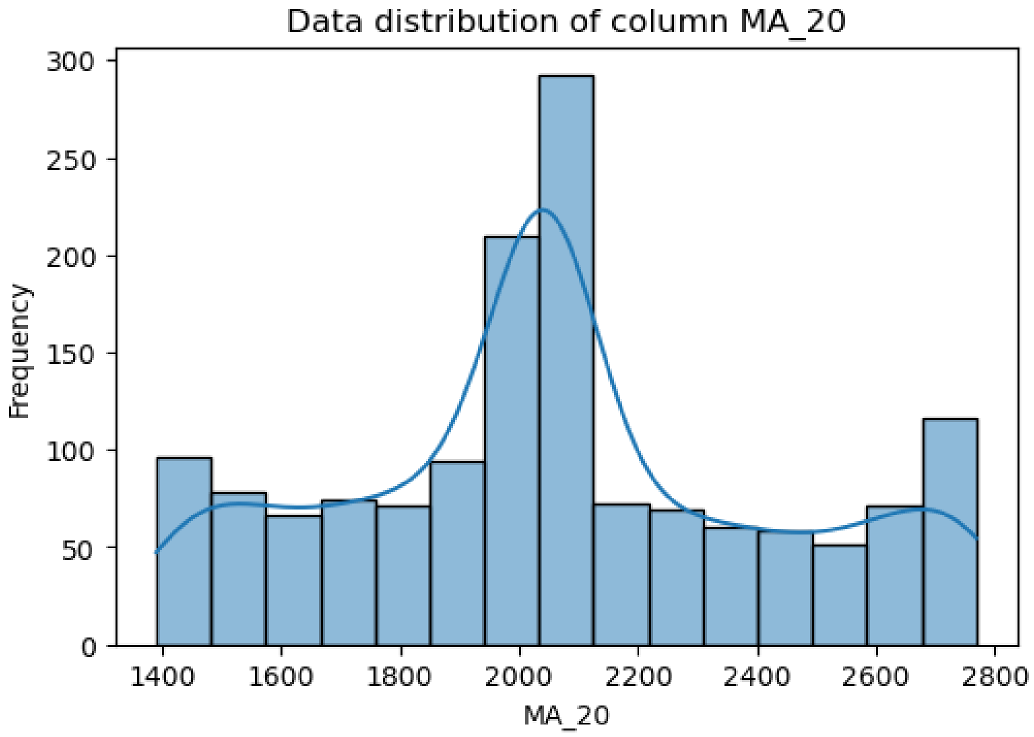
```
# Visualize the distribution of data in each column to see if they
follow a normal distribution
# If not, apply a transformation.
for i in column_name:
    # 1. Visualization before the transformation (Histograms before
standardization using Seaborn)
    plt.figure(figsize=(6, 4))
    sns.histplot(data[i], bins=15, kde=True, edgecolor='black')
    plt.title(f'Data distribution of column {i}')
    plt.xlabel(i)
    plt.ylabel('Frequency')
    plt.show()
```

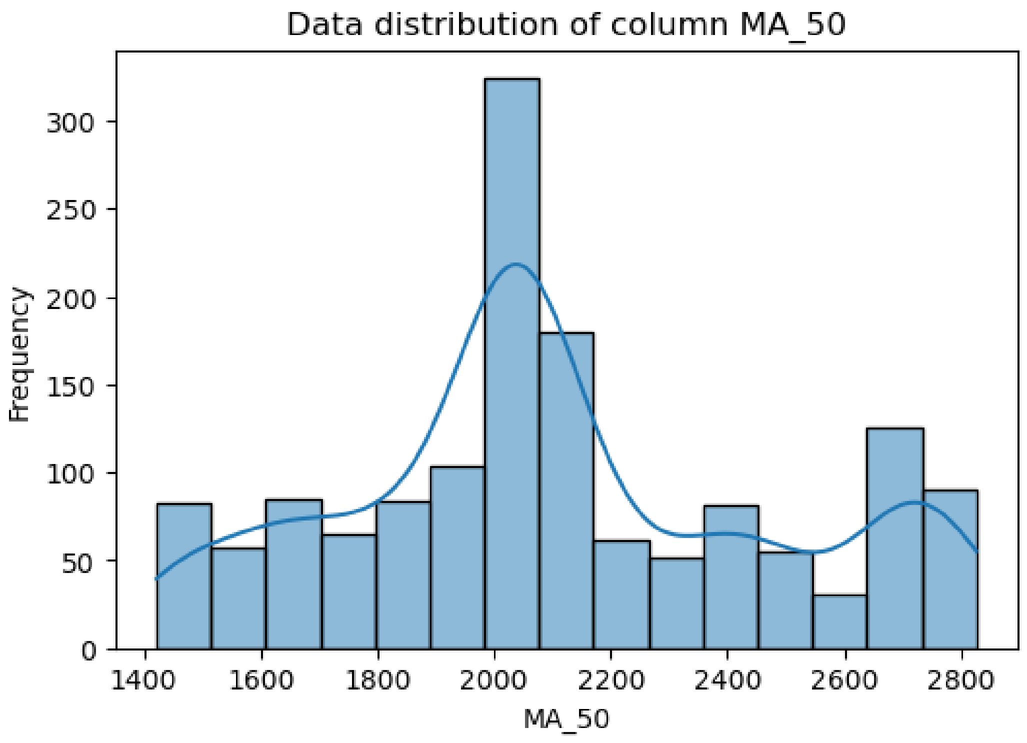


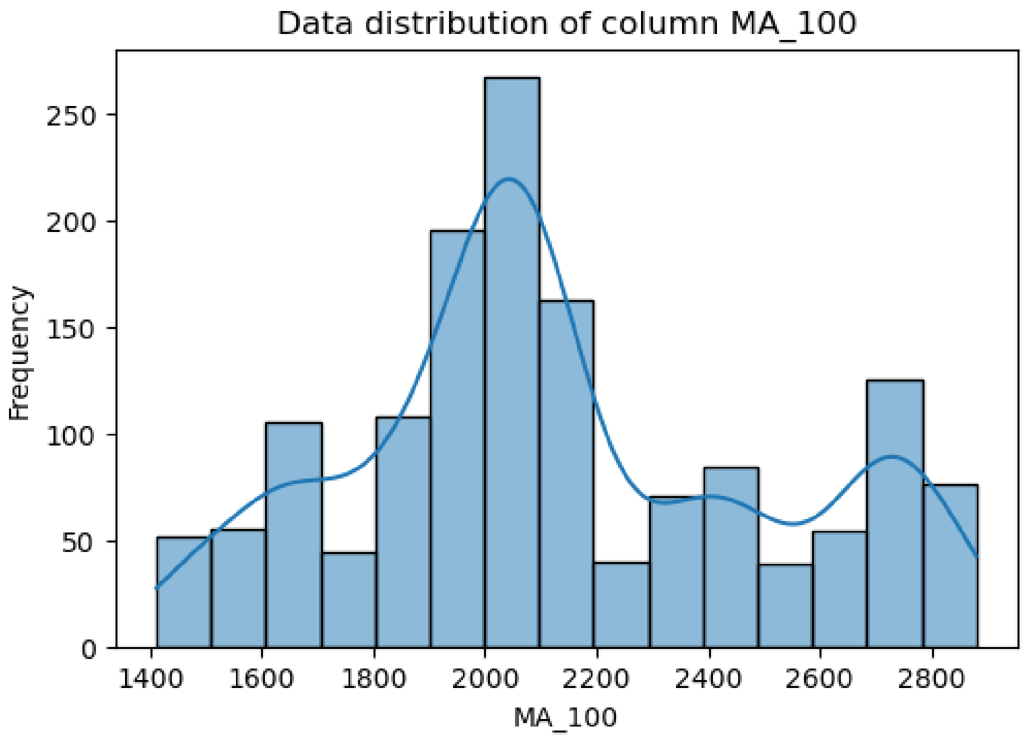

Data distribution of column High
Frequency
0
50
100
150
200
250
300
1400
1600
1800
2000
2200
2400
2600
2800
3000
High

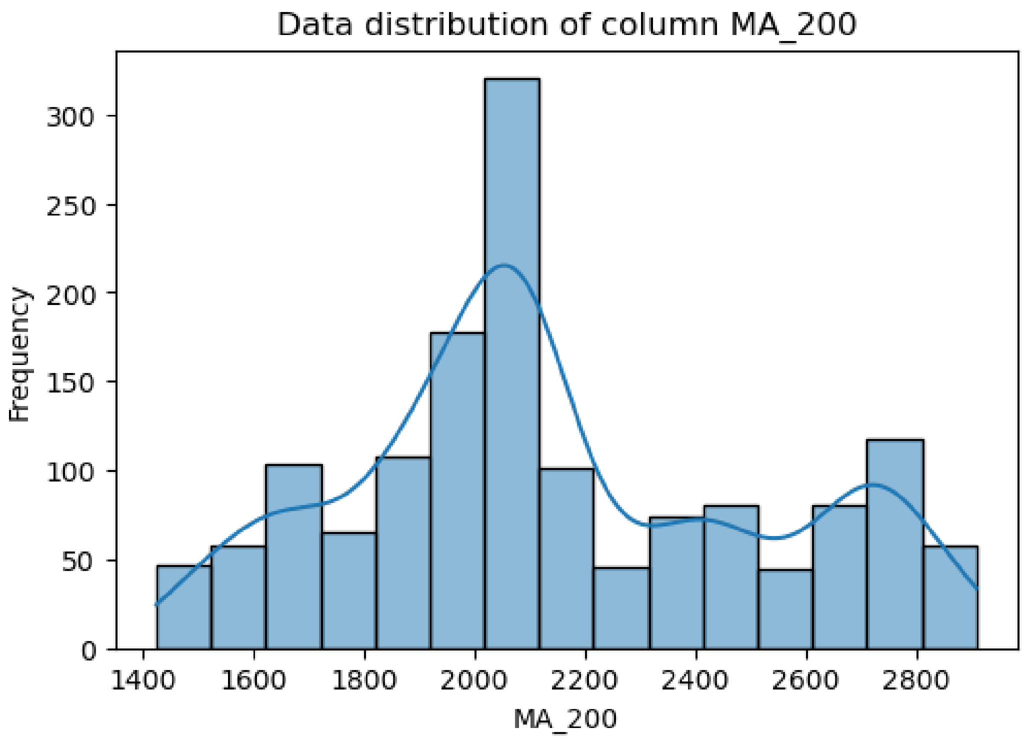

Data distribution of column Low
Frequency
0
50
100
150
200
250
300
1400
1600
1800
2000
2200
2400
2600
2800
3000
Low

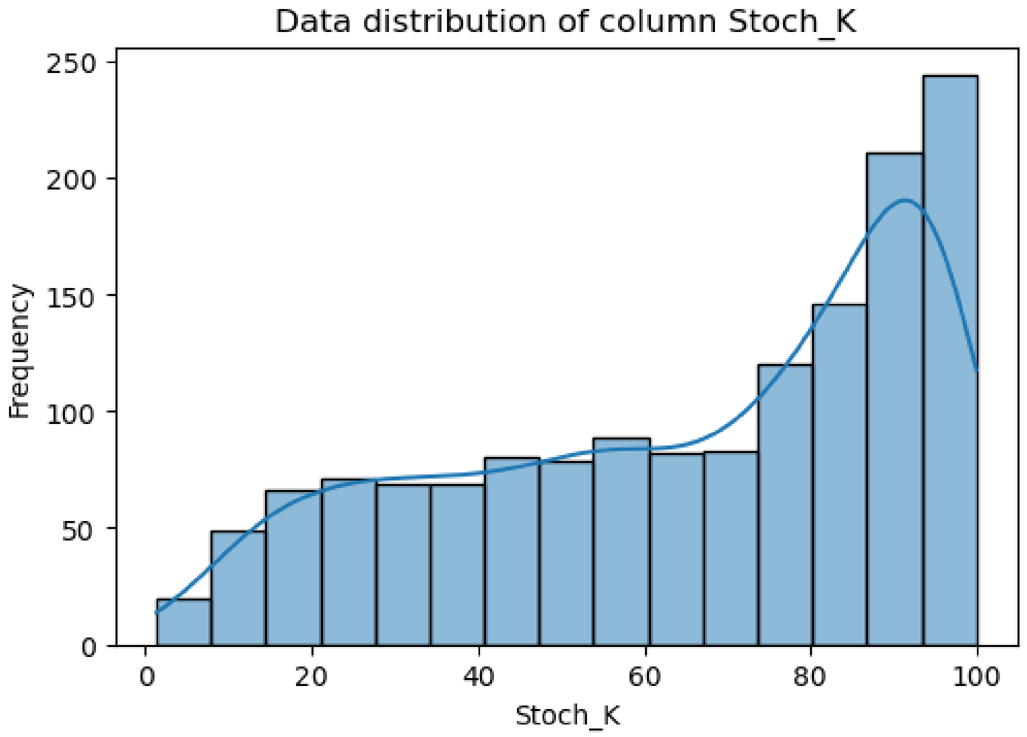

Data distribution of column Close
Frequency
Close
0
50
100
150
200
250
300
1400
1600
1800
2000
2200
2400
2600
2800
3000

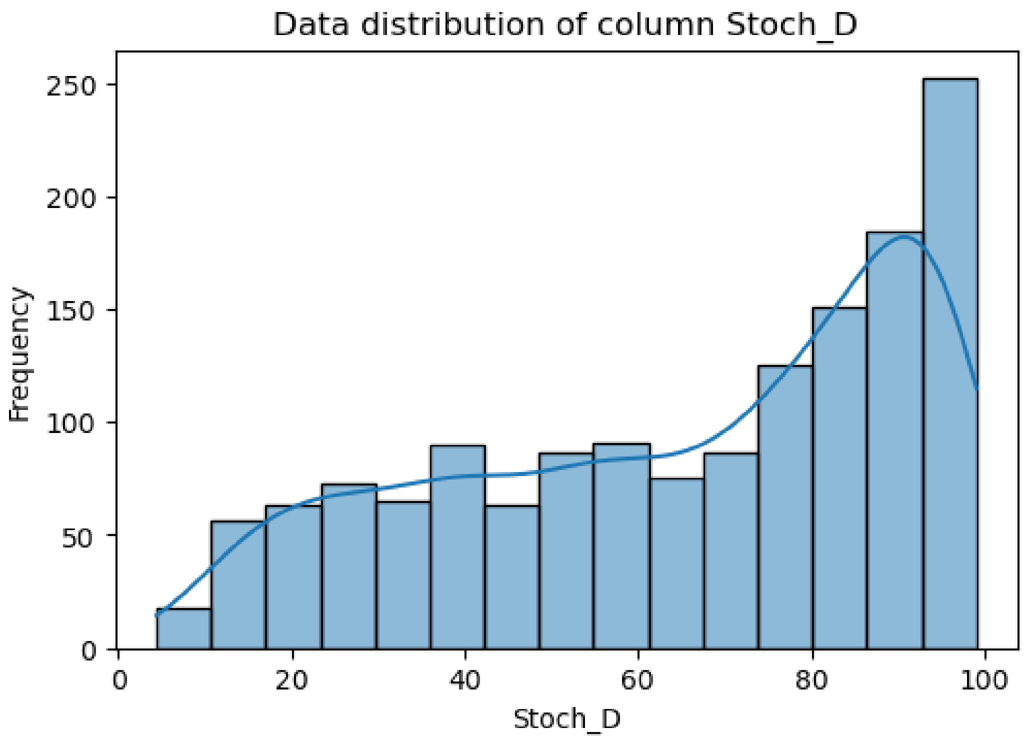

Data distribution of column MA_20
Frequency
MA_20
0
50
100
150
200
250
300
1400
1600
1800
2000
2200
2400
2600
2800

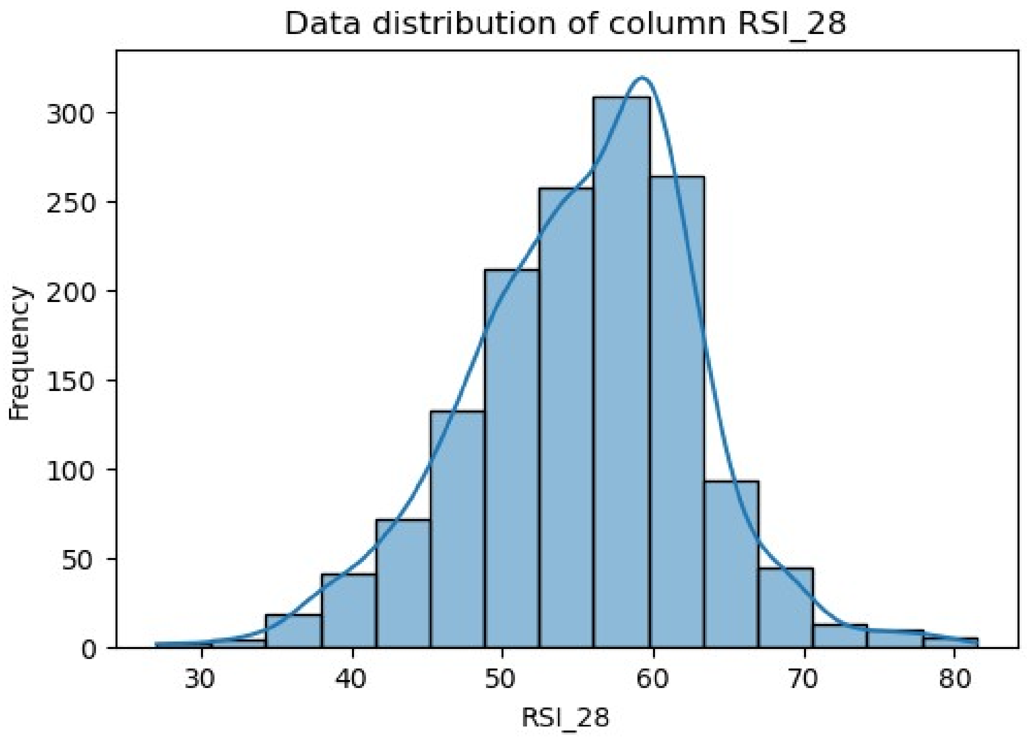

Data distribution of column MA_50
Frequency
MA_50
0
50
100
150
200
250
300
1400
1600
1800
2000
2200
2400
2600
2800

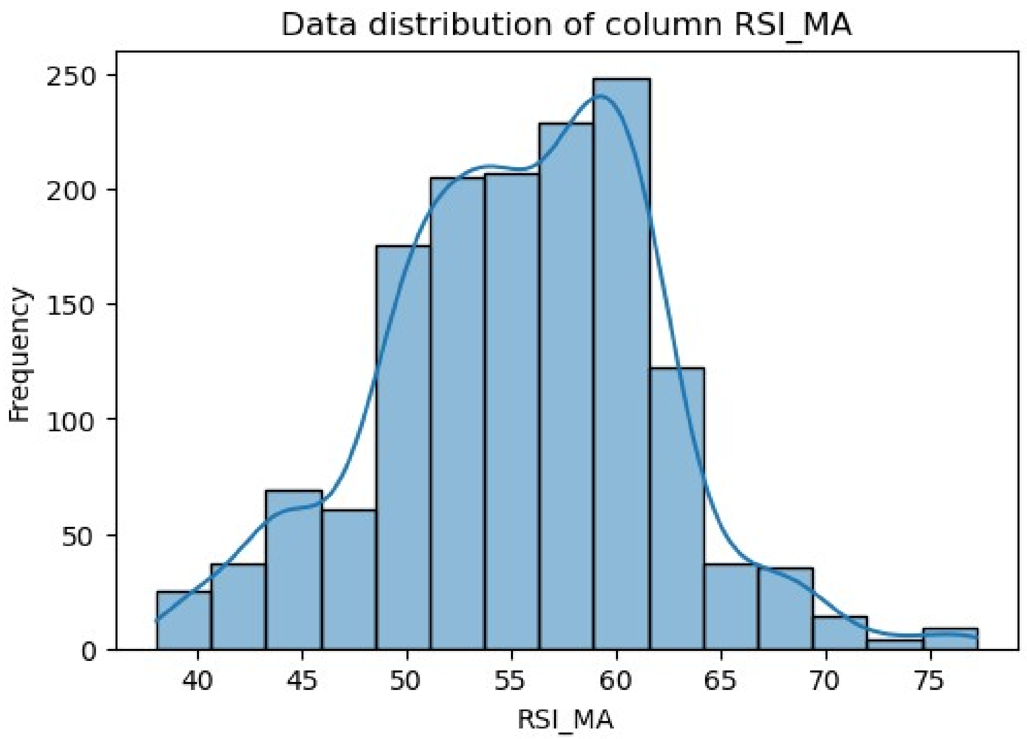

Data distribution of column MA_100
Frequency
MA_100
0
50
100
150
200
250
1400
1600
1800
2000
2200
2400
2600
2800

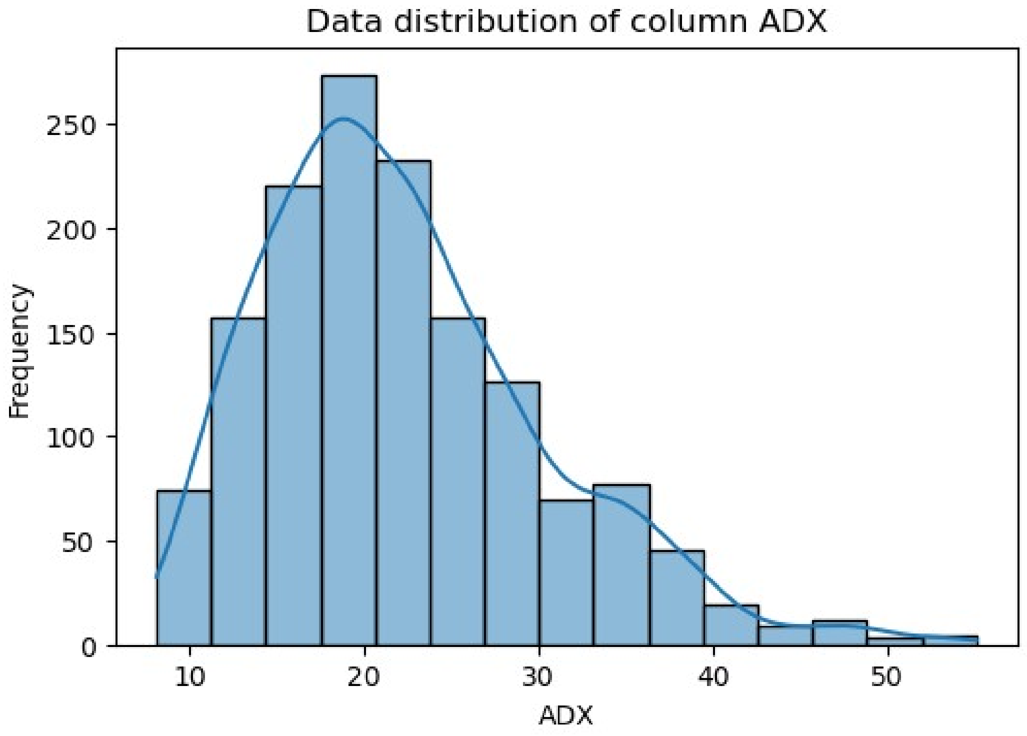

Data distribution of column MA_200
Frequency
MA_200
0
50
100
150
200
250
300
1400
1600
1800
2000
2200
2400
2600
2800

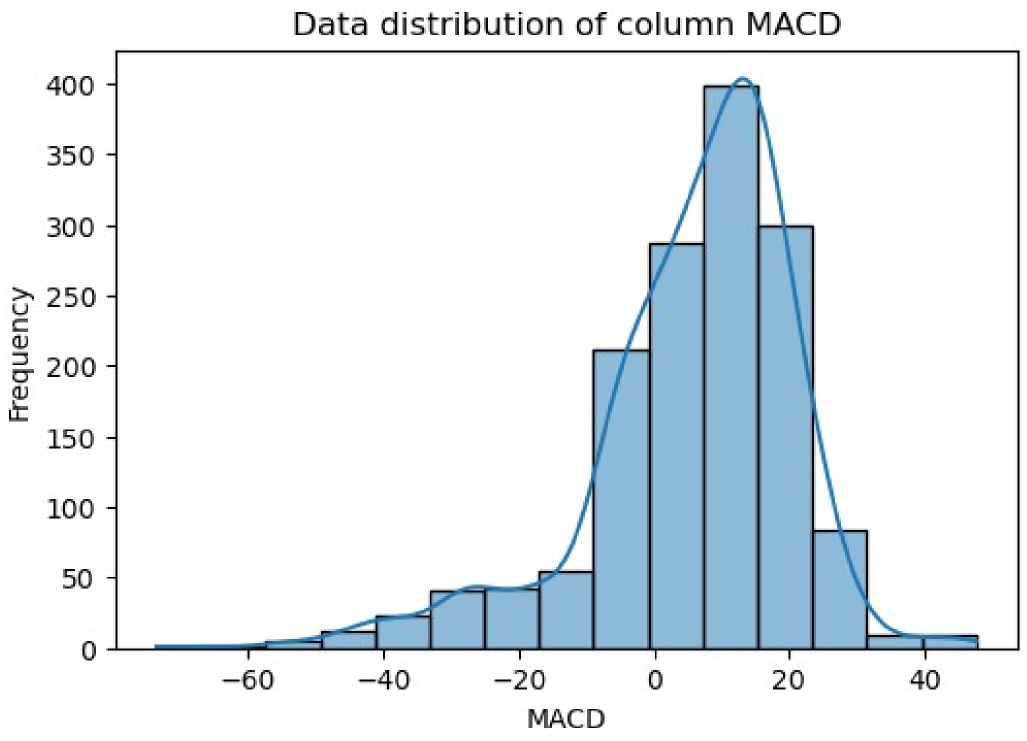

Data distribution of column Stoch_K
Frequency
Stoch_K
0
50
100
150
200
250
0
20
40
60
80
100

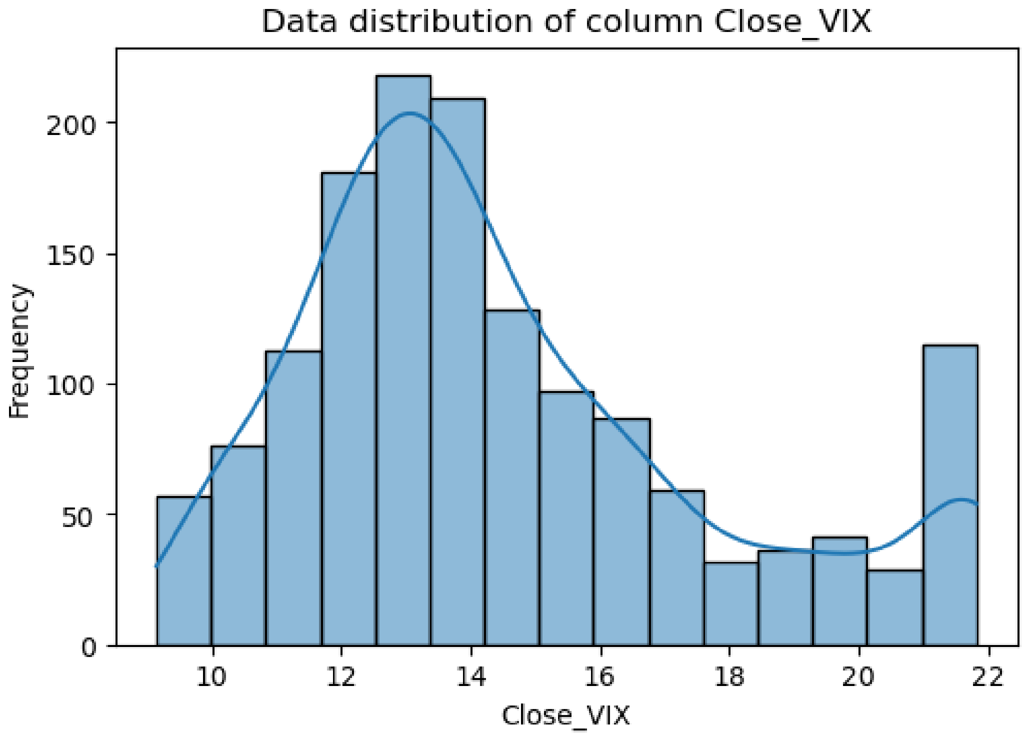

Data distribution of column Stoch_D
Frequency
Stoch_D
0
50
100
150
200
250
0
20
40
60
80
100

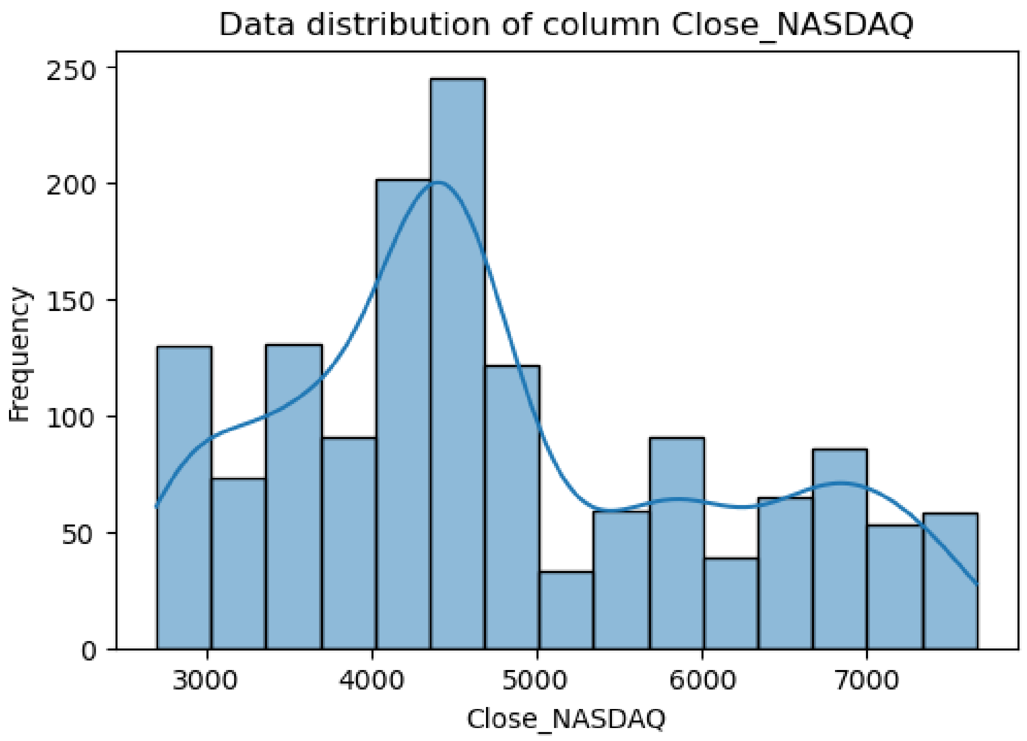

Data distribution of column RSI_28
Frequency
RSI_28
0
50
100
150
200
250
300
30
40
50
60
70
80

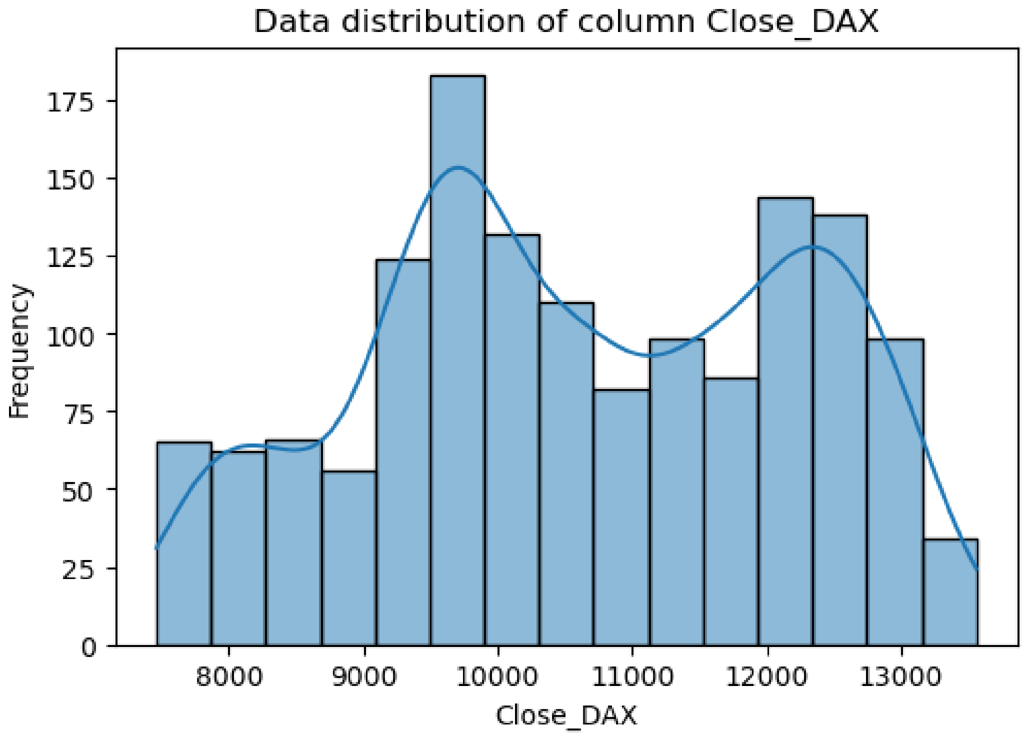

Data distribution of column RSI_MA
Frequency
RSI_MA
0
50
100
150
200
250
40
45
50
55
60
65
70
75

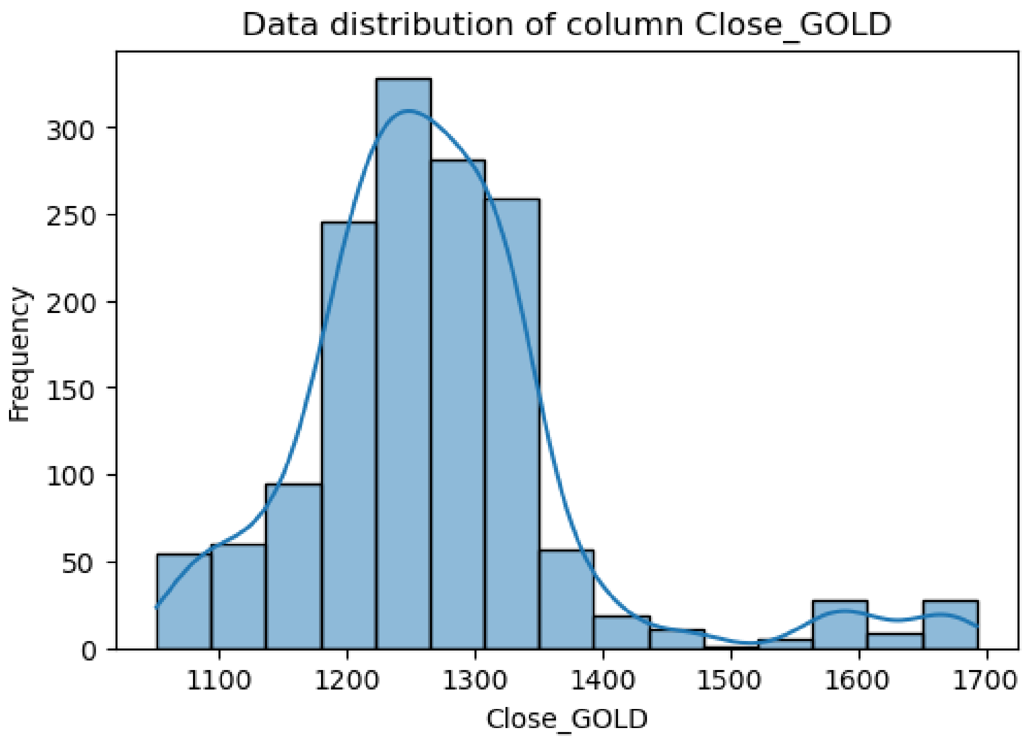

Data distribution of column ADX
Frequency
ADX
0
50
100
150
200
250
10
20
30
40
50

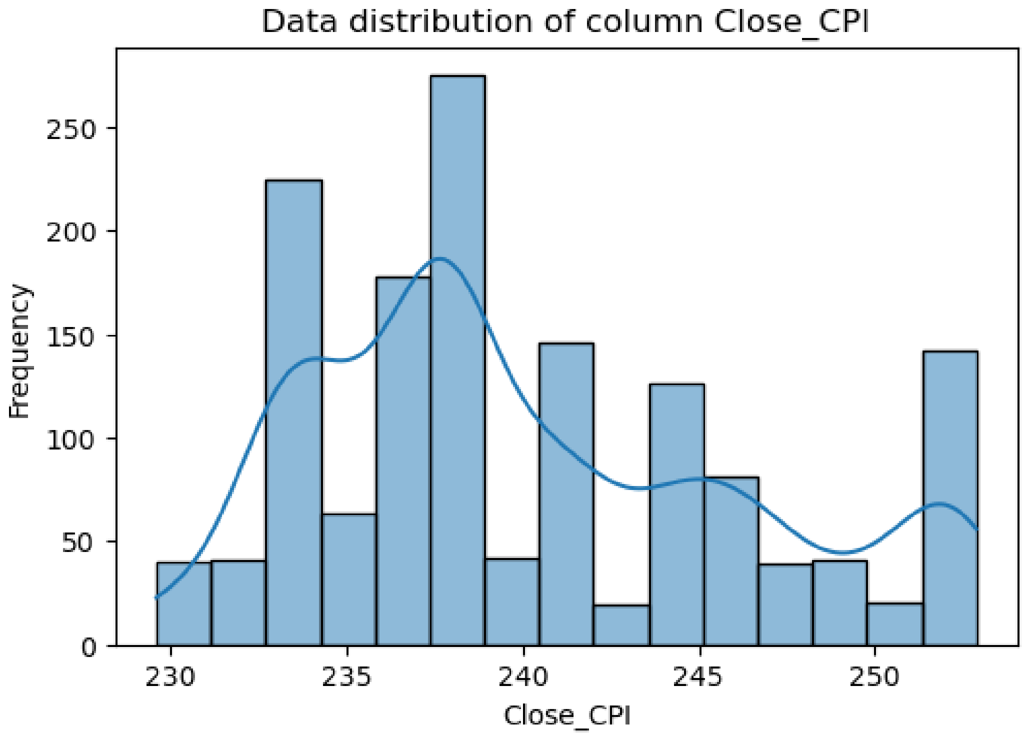

Data distribution of column MACD
Frequency
MACD
0
50
100
150
200
250
300
350
400
−60
−40
−20
0
20
40

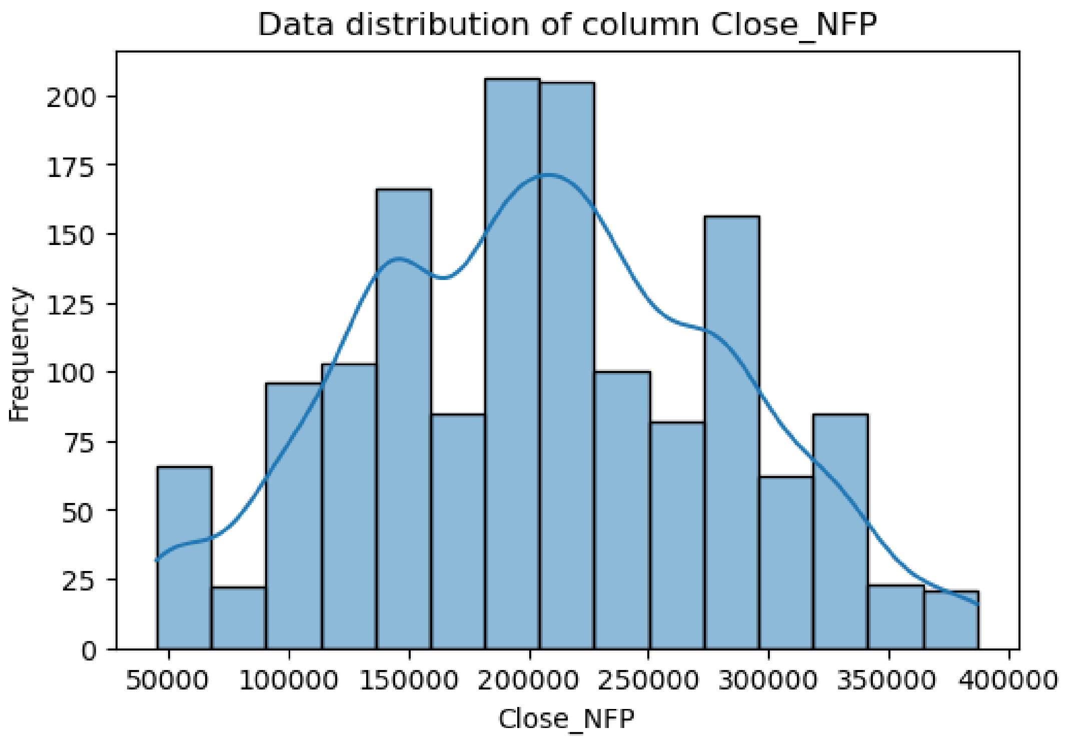

Data distribution of column Close_VIX
Frequency
Close_VIX
0
50
100
150
200
10
12
14
16
18
20
22

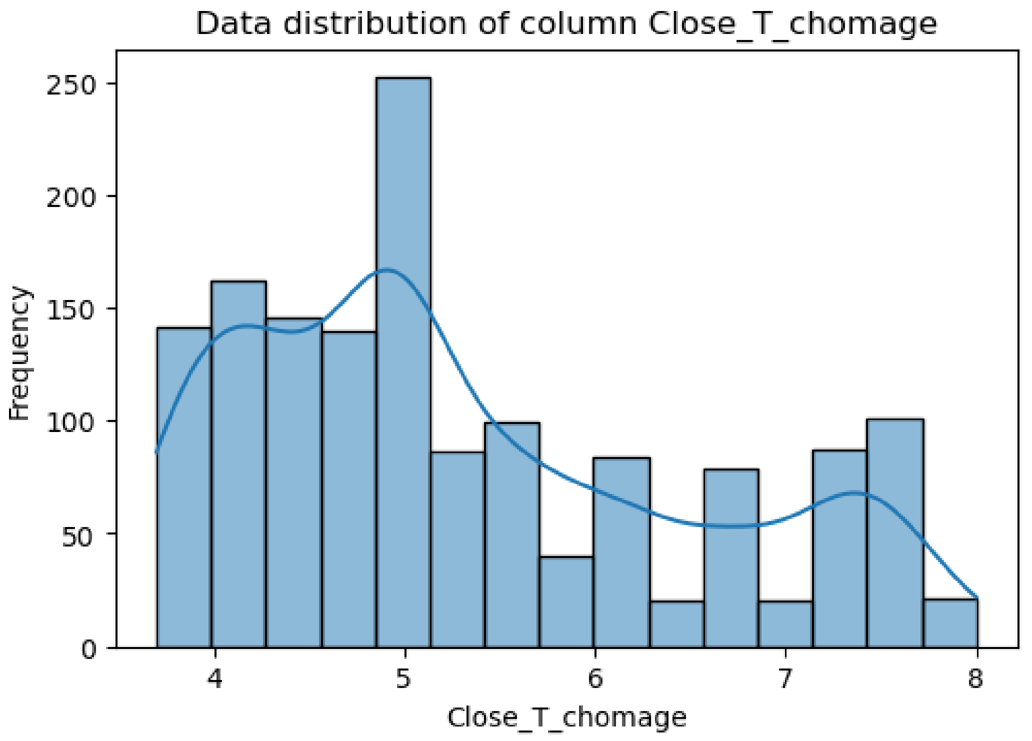

Data distribution of column Close_NASDAQ
Frequency
0
50
100
150
200
250
3000
4000
5000
6000
7000
Close_NASDAQ


Data distribution of column Close_DAX
Frequency
0
25
50
75
100
125
150
175
8000
9000
10000
11000
12000
13000
Close_DAX


Data distribution of column Close_GOLD
Frequency
Close_GOLD
0
50
100
150
200
250
300
1100
1200
1300
1400
1500
1600
1700


Data distribution of column Close_CPI
Frequency
Close_CPI
0
50
100
150
200
250
230
235
240
245
250





In this step, we visualized the distribution of data for each column to assess whether the data follows a **normal distribution**. This is a crucial step before applying transformations or standardization, as many statistical and machine learning methods assume normally distributed data.

- **Process:**
  - For each column in the dataset:
    - A **histogram** was plotted using Seaborn to observe the data's distribution.
    - The `kde=True` option added a smooth curve to better visualize the underlying pattern.
- **Purpose:**
  - By examining the histograms, we identified columns where the data deviated from normality (e.g., skewed or non-symmetric distributions).
  - For such columns, transformations (like log or standardization) can be applied in subsequent steps to bring the data closer to a normal distribution, improving the effectiveness of analysis and modeling.

1. Columns with Near-Normal Distributions

- **Open, High, Low, Close, MA_20, MA_50, MA_100, MA_200**
  These columns display relatively **symmetrical distributions** centered around their means. While there is a slight skewness in some features, the data is generally well-behaved, suggesting minimal transformation may be required.
- **RSI_28 and RSI_MA**
  These features exhibit **near-normal distributions** with minor right skewness. The data appears clean, and transformations may not be necessary.

---

2. Columns with Right-Skewed Distributions

- **ADX, Close_VIX, Close_GOLD, Close_T_chomage**
  These columns show a **right-skewed distribution** with longer tails extending toward higher values. This skewness indicates the presence of extreme values on the higher end, which could affect analysis or modeling if not addressed.
- **MACD**
  The `MACD` column displays an irregular distribution, with a noticeable spread of negative values. The presence of negative values highlights the need for preprocessing steps to ensure compatibility with certain transformations, such as Box-Cox.

---

3. Columns with Multimodal or Irregular Distributions

- **Stoch_K and Stoch_D**
  These features exhibit skewed distributions, with most data concentrated toward the upper range (near 100). This uneven spread may require transformations to bring balance and improve symmetry.
- **Close_NASDAQ and Close_DAX**
  Both columns show **bimodal patterns**, indicating the presence of multiple peaks or

clusters. These patterns suggest that the data may represent more than one underlying distribution or trend.

- **Close_CPI**
  This column displays an irregular distribution with multiple peaks. The presence of such irregularities may require smoothing transformations to make the data more uniform.

- **Close_NFP**
  The distribution shows a slight right skew, but the data is relatively well spread without extreme peaks or outliers.

```
# List of columns for square root transformation, excluding RSI_28 and
RSI_MA
sqrt_transform_columns = [
    'Open', 'High', 'Low', 'Close',
    'MA_20', 'MA_50', 'MA_100', 'MA_200',
    'Stoch_K', 'Stoch_D',
    'ADX', 'Close_VIX', 'Close_NASDAQ', 'Close_DAX',
    'Close_GOLD', 'Close_CPI', 'Close_NFP', 'Close_T_chomage'
]

# Apply square root transformation to each column in the list
for col in sqrt_transform_columns:
    # np.sqrt can be used directly if the values are strictly positive
    data[col] = np.sqrt(data[col])

# Quick check
data[sqrt_transform_columns].head()

        Open       High        Low      Close      MA_20      MA_50  \
0  37.764931  38.241731  37.764931  38.241600  37.291749  37.704699
1  38.241600  38.281458  38.151409  38.201702  37.295449  37.712277
2  38.201702  38.313705  38.196728  38.294517  37.299251  37.720717
3  38.294517  38.294517  38.165691  38.234670  37.303029  37.728142
4  38.234670  38.234670  38.100394  38.172634  37.306666  37.735171

      MA_100     MA_200   Stoch_K   Stoch_D       ADX  Close_VIX  \
0  37.570246  37.748291  7.259462  5.978186  3.634007   3.831449
1  37.570787  37.782926  9.077673  7.443925  3.772650   3.815757
2  37.579643  37.820748  9.811957  8.782150  3.911446   3.718871
3  37.587112  37.852431  9.663939  9.523136  4.010749   3.713489
4  37.598827  37.878233  9.552996  9.676879  4.048134   3.690528

   Close_NASDAQ  Close_DAX  Close_GOLD  Close_CPI   Close_NFP
Close_T_chomage
0     52.406774  88.197392   41.071718  15.152558  492.950302
2.828427
1     52.271025  88.070653   40.792463  15.152558  492.950302
2.828427
```

```
2     52.196647  88.183729   40.691160  15.152558  492.950302
2.828427
3     52.194061  87.935545   40.583168  15.152558  492.950302
2.828427
4     52.141346  87.725880   40.733972  15.152558  492.950302
2.828427

# Replace only negative values of MACD with 1
data.loc[data['MACD'] < 0, 'MACD'] = 1

# Apply the Box-Cox transformation
data['MACD'], lambda_macd = boxcox(data['MACD'])
```

1. Square Root Transformation

The **square root transformation** was applied to columns with strictly positive values to reduce skewness and stabilize variance. This method is particularly effective for moderately skewed data.

- **Columns Transformed:**
  `Open, High, Low, Close, MA_20, MA_50, MA_100, MA_200, Stoch_K, Stoch_D, ADX, Close_VIX, Close_NASDAQ, Close_DAX, Close_GOLD, Close_CPI, Close_NFP, Close_T_chomage`

- **Rationale:**
  Square root transformation compresses larger values more than smaller ones, helping to normalize distributions and improve the balance of the data.

---

2. Handling Negative Values in `MACD`

The `MACD` column contained **negative values**, which are incompatible with the Box-Cox transformation, as it requires strictly positive inputs. To address this:

- Negative values in `MACD` were replaced with **1**, ensuring the column meets the Box-Cox requirements.

Before building and evaluating models, the dataset was prepared by separating the **dependent** and **independent** variables:

```
# Define dependent and independent variables
X = data.drop(columns=['High_tomorrow'])
Y = data['High_tomorrow']
```

- **Independent Variables (X):**
  All features in the dataset except the target variable (`High_tomorrow`) were used as predictors.

- **Dependent Variable (Y):**
  The target variable is `High_tomorrow`, which represents the **"High" value of the next day** and is the variable we aim to predict.

```
# Split the data into training and testing sets
X_train, X_test, Y_train, Y_test = train_test_split(X, Y,
test_size=0.2, random_state=42)
```

Splitting the dataset into **training** and **testing** sets is a critical step in building and evaluating machine learning models. This process ensures that the model learns effectively while providing a reliable measure of its performance on unseen data.

The **training set** is used to train the model, allowing it to learn patterns and relationships between the independent variables and the target variable. By exposing the model to this subset of the data, it can optimize its parameters to minimize error during predictions.

The **testing set**, on the other hand, acts as a validation tool to evaluate the model's generalization ability. It simulates new, unseen data to assess how well the model performs beyond the training phase. Without a proper split, there is a high risk of overfitting, where the model performs exceptionally well on the training set but fails to predict accurately on new data.

In this project, an **80/20 split** was applied, where 80% of the data was allocated for training and 20% for testing. The **random state** was set to 42 to ensure reproducibility of results. This split strikes a balance between providing enough data for the model to learn effectively and retaining sufficient data for testing its performance.

By splitting the data appropriately, we ensure that the model is not only capable of fitting the training data but also generalizing to unseen inputs, resulting in more robust and reliable predictions.

Importance and Effect of Data Transformation After Data Splitting

After splitting the data into training and testing sets, applying appropriate **data transformations** such as standardization and normalization is crucial to ensure that the features are on comparable scales. Many machine learning algorithms are sensitive to the scale and distribution of input features, and untransformed data can lead to poor model performance or biased predictions.

Why Data Transformation is Important

1. **Improves Model Performance**:
   Models like gradient-based methods (e.g., linear regression, logistic regression, and neural networks) perform better when features are scaled. It helps optimize the convergence speed and ensures accurate predictions.

2. **Ensures Features Are Comparable**:
   In real-world datasets, features often have different ranges and units. For example, stock prices might range in thousands, while technical indicators like `RSI` and

`Stoch_K` have values between 0 and 100. Scaling ensures that no single feature dominates the learning process due to its magnitude.

3. **Reduces Bias in Distance-Based Algorithms**:
   Algorithms like k-Nearest Neighbors (k-NN) or clustering methods calculate distances between data points. If features are not scaled, those with larger ranges can disproportionately influence the results.

4. **Maintains Generalization to New Data**:
   Transformations are applied to the training data **first**, and the same transformation is applied to the testing data. This ensures consistency and avoids information leakage, preserving the model's ability to generalize effectively.

Effects of Transformations

1. **Standardization (Z-Score Scaling)**:
   Standardization transforms features to have a **mean of 0** and a **standard deviation of 1**. It centers the data around zero and ensures a uniform scale for features with different ranges or units.

   - **Effect**: Features like `Open`, `High`, `ADX`, and `MACD`, which originally had varying magnitudes, are now comparable. This transformation is particularly beneficial for algorithms like linear regression and principal component analysis (PCA).

2. **Normalization (Min-Max Scaling)**:
   Normalization scales features to a specific range, typically between **0 and 1**. It preserves the relationships within the data but compresses the values into a defined range.

   - **Effect**: Features like `Stoch_K`, `Stoch_D`, and `RSI`, which are already on smaller scales, are brought into a uniform range, ensuring consistency and stability.

```python
# Initialize scalers
standard_scaler = StandardScaler()
min_max_scaler = MinMaxScaler()

# Lists of columns for each transformation
standardization_columns = [
    'Open', 'High', 'Low', 'Close',
    'MA_20', 'MA_50', 'MA_100', 'MA_200',
    'ADX', 'MACD', 'Close_VIX',
    'Close_NASDAQ', 'Close_DAX', 'Close_GOLD',
    'Close_CPI', 'Close_NFP'
]
normalization_columns = [
    'Stoch_K', 'Stoch_D', 'RSI_28',
    'RSI_MA', 'Close_T_chomage'
]

# Apply standardization to the selected columns of the training set
X_train[standardization_columns] =
standard_scaler.fit_transform(X_train[standardization_columns])
# Apply the same transformation to the test set
X_test[standardization_columns] =
standard_scaler.transform(X_test[standardization_columns])

# Apply normalization to the selected columns of the training set
X_train[normalization_columns] =
min_max_scaler.fit_transform(X_train[normalization_columns])
# Apply the same transformation to the test set
X_test[normalization_columns] =
min_max_scaler.transform(X_test[normalization_columns])
```

Two scaling techniques were applied to prepare the data for analysis: **standardization** (StandardScaler) and **normalization** (MinMaxScaler), applied selectively based on the statistical distributions of the features.

---

1. Standardization

Objective: Transform features to have a **mean of 0** and **standard deviation of 1**. This technique is ideal for continuous features with approximately Gaussian distributions.

Columns standardized: `Open`, `High`, `Low`, `Close`, `MA_20`, `MA_50`, `MA_100`, `MA_200`, `ADX`, `MACD`,`Close_VIX`, `Close_NASDAQ`, `Close_DAX`, `Close_GOLD`,`Close_CPI`, `Close_NFP`

Based on the visualizations of the distribution plots:

- The price-related columns (`Open`, `High`, `Low`, `Close`) and moving averages (`MA_20`, `MA_50`, `MA_100`, `MA_200`) display **symmetrical distributions** that resemble a bell shape, but their values are spread across different ranges.

- The `MACD` indicator is already centered around zero, but its values vary significantly, requiring scaling for consistency.

- Standardization brings all these features to a common scale with a mean of 0 and a standard deviation of 1, ensuring that differences in magnitude do not impact models sensitive to feature scales.

---

2. Normalization

Objective: Scale features to a **range of 0 to 1**, suitable for bounded or skewed data.

Columns normalized:

- **Stochastic indicators:** `Stoch_K`, `Stoch_D`
- **Relative Strength Index**: `RSI_28`, `RSI_MA`
- **Unemployment rate:** `Close_T_chomage`

Based on the visualizations of the distribution plots:

- `Stoch_K` and `Stoch_D` range between **0 and 100**, with skewed distributions toward higher values.
- `RSI_28` and `RSI_MA` are bounded between ~30 and 80, following a restricted bell shape.
- `Close_T_chomage` has a limited range (~4 to 8) but displays uneven distribution.
- Normalization is appropriate for these features as it preserves their bounded nature and aligns their scales to the [0, 1] range.

PART B: Model Selection

I) Variable Selection Methods

In this project, we implemented **variable selection** techniques to identify the most significant predictors for the final model. The methods used include **Forward Selection**, **Backward Selection**, and **Stepwise Selection**:

```
def forward_selection(X_train, Y_train, variables_available):
    """Performs forward selection and returns the selected 
variables."""
    variables_included = []
    variables_eliminated = []

    while variables_available:
        pvalues_candidate = pd.Series(index=variables_available, 
dtype=float)
        for var in variables_available:
            X_candidate = sm.add_constant(X_train[variables_included + 
[var]])
            model = sm.OLS(Y_train, X_candidate).fit()
            pvalues_candidate[var] = model.pvalues[var]

        min_pvalue = pvalues_candidate.min()
        if min_pvalue < 0.05:
            best_var = pvalues_candidate.idxmin()
            variables_included.append(best_var)
            variables_available.remove(best_var)
            print(f"Added variable '{best_var}' with p-value 
{min_pvalue:.4f}")

            # Backward elimination within forward step
            while True:
                X_temp = sm.add_constant(X_train[variables_included])
                model = sm.OLS(Y_train, X_temp).fit()
                pvalues = model.pvalues.iloc[1:]  # Exclude intercept
                max_pvalue = pvalues.max()
                if max_pvalue >= 0.05:
                    worst_var = pvalues.idxmax()
                    variables_included.remove(worst_var)
                    variables_eliminated.append(worst_var)
                    print(f"Removed variable '{worst_var}' with p-
value {max_pvalue:.4f}")
                else:
                    break
        else:
            print("No additional significant variable found.")
            break

    return variables_included
```

1) Forward Selection

Method:
Forward selection starts with an **empty model** and progressively adds variables based on their statistical significance.

Process:
Variables are added **one at a time** based on their p-values.

Key Steps:

- At each step, the variable with the **lowest p-value** (< 0.05) is added to the model.
- The model is re-evaluated after each addition to check for improvements.

The process stops when **no significant variables** (p-value < 0.05) can be added to the model.

The provided function `forward_selection` implements the **forward selection** process while incorporating an **optional backward elimination step** within each iteration.

1. **Initialization**:
   - `variables_included`: List of variables currently in the model.
   - `variables_eliminated`: Tracks variables removed during backward elimination.

   - `variables_available`: Remaining predictors available for inclusion.
2. **Forward Step**:
   - For each candidate variable in `variables_available`, the function computes p-values by adding the variable to the model.
   - The variable with the **minimum p-value** is selected if it satisfies the threshold (< 0.05).
3. **Backward Elimination (Within Forward Step)**:
   - Once a new variable is added, the function re-evaluates all predictors in the model.
   - If any predictor's p-value becomes **≥ 0.05**, it is removed from `variables_included`.
4. **Stopping Rule**:
   - The forward step stops when no remaining variable meets the p-value threshold (< 0.05).
   - The backward step ends when all remaining variables are significant.

Output

The function returns the list `variables_included`, which contains the final set of significant predictors.

Advantages

- This hybrid approach ensures that the model does not retain insignificant variables added earlier.
- The **dynamic backward elimination** enhances model reliability by continuously re-evaluating predictors.

```
def backward_elimination(X_train, Y_train, variables):
    """Performs backward elimination and returns the selected 
variables."""
    variables_included = variables.copy()
    while len(variables_included) > 0:
        X_temp = sm.add_constant(X_train[variables_included])
        model = sm.OLS(Y_train, X_temp).fit()
        pvalues = model.pvalues.iloc[1:]  # Exclude intercept

        max_pvalue = pvalues.max()
        worst_var = pvalues.idxmax()

        if max_pvalue > 0.05:
            variables_included.remove(worst_var)
            print(f"Removed variable '{worst_var}' with p-value 
{max_pvalue:.4f}")
        else:
            print("All remaining variables are significant.")
            break
    return variables_included
```

2) Backward Selection

Method:
Backward selection starts with a **full model** that includes all predictors and progressively removes insignificant variables.

Process:
Variables are removed iteratively based on their p-values.

Key Steps:

- At each step, the variable with the **highest p-value** (> 0.05) is removed from the model.

- The model is re-evaluated after each removal.

The process stops when all **remaining variables** have p-values **≤ 0.05**.

The provided function `backward_elimination` performs the **backward elimination** process to iteratively remove insignificant predictors from the model.

1. **Initialization**:
   - `variables_included`: A list containing all predictors initially considered for the model.
2. **Iteration**:
   - The function fits an OLS regression model using the current set of predictors.

   - The p-values of all variables (excluding the intercept) are evaluated.
3. **Elimination Step**:
   - The predictor with the **highest p-value** (> 0.05) is removed from `variables_included`.

   - This step repeats until all remaining variables are significant (p-value ≤ 0.05).
4. **Stopping Rule**:
   - The process terminates when no predictor exceeds the p-value threshold.

Output

The function returns `variables_included`, the list of significant predictors that remain after the elimination process.

Advantages

- The algorithm ensures that only statistically significant variables are retained in the final model.
- It automates the variable selection process, simplifying the identification of key predictors.

```
def bidirectional_selection(X_train, Y_train, variables):
    """Performs bidirectional selection and returns the selected 
variables."""
    random.seed(42)
    variables_included = random.sample(variables, 10)
    changed = True

    while changed:
        changed = False
        # Backward elimination
        X_temp = sm.add_constant(X_train[variables_included])
        model = sm.OLS(Y_train, X_temp).fit()
        pvalues = model.pvalues.iloc[1:]  # Exclude intercept
        max_pvalue = pvalues.max()
        worst_var = pvalues.idxmax()

        if max_pvalue > 0.05:
            variables_included.remove(worst_var)
            print(f"Removed variable '{worst_var}' with p-value 
{max_pvalue:.4f}")
            changed = True
            continue

        # Forward selection
        variables_available = [var for var in variables if var not in 
variables_included]
        pvalues_candidate = pd.Series(index=variables_available, 
dtype=float)
        for var in variables_available:
            X_candidate = sm.add_constant(X_train[variables_included + 
[var]])
            model = sm.OLS(Y_train, X_candidate).fit()
            pvalues_candidate[var] = model.pvalues[var]

        min_pvalue = pvalues_candidate.min()
        if min_pvalue < 0.05:
            best_var = pvalues_candidate.idxmin()
            variables_included.append(best_var)
            print(f"Added variable '{best_var}' with p-value 
{min_pvalue:.4f}")
            changed = True
        else:
            print("No additional significant variable found.")
    return variables_included
```

3) Stepwise Selection

Method:
Stepwise selection combines the **forward** and **backward selection** methods, allowing both the addition and removal of variables.

Process:

- The process begins with a set of **10 randomly selected variables**.
- Variables are either **added** or **removed** based on their significance.

Key Steps:

- **Adding Variables**: A variable is added if it has the **lowest p-value** (< 0.05).
- **Removing Variables**: A variable is removed if it has the **highest p-value** (> 0.05).
- The model alternates between these two steps until no further changes can improve the model.

The process stops when **no additional variables** can be added or removed.

The `bidirectional_selection` function combines **forward selection** and **backward elimination** to iteratively refine a regression model.

1. **Initialization**:
   - A **random subset** of 10 predictors is selected to start the process.
2. **Backward Elimination Step**:
   - Fit an OLS model with the current predictors.
   - Identify the predictor with the **highest p-value** (> 0.05) and remove it.
   - If a variable is removed, the process restarts.
3. **Forward Selection Step**:
   - Evaluate remaining unused predictors by adding them one at a time to the model.
   - Identify the predictor with the **lowest p-value** (< 0.05) and add it.
   - If a variable is added, the process restarts.
4. **Stopping Rule**:
   - The process stops when no variables can be added or removed.

Output

- Returns `variables_included`, the final list of selected predictors.

Advantages

- Combines the strengths of forward selection and backward elimination.
- More flexible and robust than purely forward or backward methods.
- Ensures that the model retains only statistically significant predictors.

II) Ridge Regression and Cross-Validation

1) Ridge Regression: Addressing Multicollinearity with L2 Regularization

Ridge regression is an extension of linear regression that includes **L2 regularization** to handle **multicollinearity**, a situation where predictors are highly correlated. Multicollinearity can cause instability in the regression coefficients, leading to poor generalization on new data.

In Ridge regression, the loss function is modified to include a penalty term that shrinks the regression coefficients β_i towards zero. The new loss function is defined as:

Loss = RSS + $\alpha \Sigma (\beta_i^2)$

Where:

- **RSS** (Residual Sum of Squares) measures the model error.
- **α** is the **regularization strength**; a higher α shrinks coefficients (β_i) closer to zero.
- **β_i** are the regression coefficients for each predictor.

The penalty term **$\alpha \Sigma (\beta_i^2)$** ensures that large coefficients are penalized, helping to stabilize the model and reduce the impact of multicollinearity. Unlike other regularization techniques (e.g., Lasso), Ridge regression shrinks coefficients but does not eliminate any predictors.

---

2) Cross-Validation: Evaluating Model Performance

- The **Root Mean Squared Error (RMSE)** is a metric commonly used to measure the accuracy of regression models.
- **How It Works**:
  - It calculates the **difference** (residuals) between the actual values (`y_true`) and the predicted values (`y_pred`).
  - Each residual is then squared to ensure all errors are positive.

– The mean of the squared residuals is computed to determine the average squared error.

– Finally, the square root of this mean is taken to return the RMSE, which provides the error in the same units as the target variable.

- **Formula**:

$$\text{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(y_{\text{true},i} - y_{\text{pred},i}\right)^2}$$

Where:

– $y_{\text{true},i}$: Actual values.

– $y_{\text{pred},i}$: Predicted values.

– n: Number of data points.

Once Ridge regression is applied, we assess the model's performance using **10-fold cross-validation**. This process allows us to calculate the **average Root Mean Square Error (RMSE)** on the training set.

Steps of 10-Fold Cross-Validation:

1. The training data is split into **10 subsets (folds)** of approximately equal size.
2. The Ridge regression model is trained on **9 folds** and validated on the **remaining fold**.
3. This process is repeated **10 times**, with each fold serving as the validation set once.
4. After all iterations, we calculate the **RMSE** for each fold and take the average to obtain the **mean RMSE** for the training set.

The average RMSE serves as a reliable measure of the model's performance on the training data, ensuring that the evaluation is not overly dependent on any single subset of the data.

---

3) Comparing RMSE: Detecting Overfitting and Evaluating Generalization

After computing the average RMSE on the training set using cross-validation, we compare it with the RMSE obtained on the **test set** (unseen data). This comparison provides insights into the model's generalization ability:

1. **Overfitting Detection**:
   – If the RMSE on the training set is **significantly lower** than the RMSE on the test set, it indicates that the model is overfitting. This means the model performs well on training data but fails to generalize to new, unseen data.
2. **Model Performance**:
   – If the RMSE on the training set is **close to** the RMSE on the test set, the model generalizes well, demonstrating consistent performance across both datasets.

```python
def calculate_rmse(model, X, Y):
    """Cross-validation with K=10 and calculation of the mean RMSE."""
    mse = make_scorer(mean_squared_error, squared=False)  # 
squared=False returns RMSE instead of MSE
    scores = cross_val_score(model, X, Y, cv=10, scoring=mse)  # K=10
    return scores.mean()
```

The `calculate_rmse` function evaluates a model's performance using **10-fold cross-validation** and computes the **mean RMSE** (Root Mean Square Error). The steps are as follows:

1. **Custom Scoring Metric**
   The function uses the `make_scorer` method to ensure that RMSE, rather than MSE, is used for evaluation by setting `squared=False`.

2. **Cross-Validation Process**
   Using `cross_val_score`, the function performs 10-fold cross-validation, where the data is split into 10 subsets, and the RMSE is calculated for each fold.

3. **Mean RMSE**
   The function returns the **average RMSE** across all folds, providing a reliable measure of model performance during cross-validation.

```python
def evaluate_ridge_model(X_train, Y_train, X_test, Y_test, 
variables_included, alpha=1.0):
    """Fits and evaluates a Ridge model, returning the RMSE and 
average cross-validation RMSE."""
    X_train_final = X_train[variables_included]
    X_test_final = X_test[variables_included]

    ridge_model = Ridge(alpha=alpha)

    # Cross-validation RMSE
    average_rmse_cv = calculate_rmse(ridge_model, X_train_final, 
Y_train)

    # Test set RMSE
    ridge_model.fit(X_train_final, Y_train)
    Y_pred = ridge_model.predict(X_test_final)
    rmse_test = np.sqrt(mean_squared_error(Y_test, Y_pred))

    return ridge_model, average_rmse_cv, rmse_test
```

The `evaluate_ridge_model` function evaluates a Ridge regression model on both the training and test datasets. The key steps are as follows:

1. **Input Preparation**

- The training (`X_train_final`) and test (`X_test_final`) datasets are filtered to include only the selected predictors (`variables_included`).

2. **Model Initialization**
   A Ridge regression model is initialized with a regularization parameter `alpha` (default value = 1.0).

3. **Cross-Validation RMSE**
   The function uses `calculate_rmse` to perform **10-fold cross-validation** on the training data and compute the **average RMSE**, ensuring reliable evaluation.

4. **Test Set RMSE**

   - The Ridge model is fit to the training data.
   - Predictions (`Y_pred`) are made on the test data.
   - The **RMSE** is calculated on the test set using the `mean_squared_error` function with square root applied.

5. **Output**
   The function returns:

   - The trained Ridge model.
   - The average cross-validation RMSE.
   - The test set RMSE.

III) Final Model Selection

```
def run_selection_process(X_train, Y_train, X_test, Y_test,
variables):
    """Runs forward, backward, and bidirectional selection
processes."""
    minimum_RMSE = float("inf")
    final_model, final_variables = None, None

    # Forward Selection
    print("\n--- Forward Selection ---")
    variables_forward = forward_selection(X_train, Y_train,
variables.copy())
    model_forward, rmse_cv_forward, rmse_test_forward =
evaluate_ridge_model(X_train, Y_train, X_test, Y_test,
variables_forward)
    print(f"Forward Selection: Average RMSE (CV):
{rmse_cv_forward:.4f}, RMSE (Test): {rmse_test_forward:.4f}")

    # Backward Elimination
    print("\n--- Backward Elimination ---")
    variables_backward = backward_elimination(X_train, Y_train,
variables.copy())
    model_backward, rmse_cv_backward, rmse_test_backward =
evaluate_ridge_model(X_train, Y_train, X_test, Y_test,
variables_backward)
```

```
    print(f"Backward Elimination: Average RMSE (CV): 
{rmse_cv_backward:.4f}, RMSE (Test): {rmse_test_backward:.4f}")

    # Bidirectional Selection
    print("\n--- Bidirectional Selection ---")
    variables_stepwise = bidirectional_selection(X_train, Y_train, 
variables.copy())
    model_stepwise, rmse_cv_stepwise, rmse_test_stepwise = 
evaluate_ridge_model(X_train, Y_train, X_test, Y_test, 
variables_stepwise)
    print(f"Bidirectional Selection: Average RMSE (CV): 
{rmse_cv_stepwise:.4f}, RMSE (Test): {rmse_test_stepwise:.4f}")

    # Compare RMSE values
    for rmse, model, selected_vars in [(rmse_test_forward, 
model_forward, variables_forward),
                                       (rmse_test_backward, 
model_backward, variables_backward),
                                       (rmse_test_stepwise, 
model_stepwise, variables_stepwise)]:
        if rmse < minimum_RMSE:
            minimum_RMSE = rmse
            final_model = model
            final_variables = selected_vars

    print("\nFinal Model Evaluation:")
    print(f"Variables: {final_variables}")
    print(f"RMSE on Test Set: {minimum_RMSE:.4f}")
    return final_model, final_variables
```

1) Function: `run_selection_process`

This function automates the execution and comparison of three variable selection methods—**Forward Selection**, **Backward Elimination**, and **Bidirectional Selection**. For each selection method, the Ridge regression model is evaluated based on two metrics:

1. **Average RMSE from Cross-Validation** (10-fold cross-validation).
2. **RMSE on the Test Set**.

Key Steps:

1. **Forward Selection**:
   – Runs the forward selection process and evaluates the Ridge model performance using `evaluate_ridge_model`.
2. **Backward Elimination**:
   – Runs the backward elimination process and evaluates the Ridge model using the same approach.
3. **Bidirectional Selection**:
   – Performs a combined forward-backward selection process and evaluates the model performance.

4. **Model Comparison**:
   - The function compares the **RMSE on the Test Set** from all three selection methods.
   - The model with the lowest RMSE on the test set is selected as the **final model**.

Outputs:

- **Final Model**: The Ridge regression model trained on the selected variables.
- **Selected Variables**: The best-performing subset of predictors.
- **Performance Metric**: RMSE on the test set for the final model.

```
warnings.filterwarnings("ignore")
variables = X.columns.tolist()
final_model, final_variables = run_selection_process(X_train, Y_train,
X_test, Y_test, variables)
```

```
--- Forward Selection ---
Added variable 'Open' with p-value 0.0000
Added variable 'Close' with p-value 0.0000
Added variable 'Close_GOLD' with p-value 0.0000
Added variable 'Close_VIX' with p-value 0.0000
Removed variable 'Open' with p-value 0.1195
Added variable 'Close_T_chomage' with p-value 0.0000
Added variable 'Close_NASDAQ' with p-value 0.0000
Added variable 'Close_CPI' with p-value 0.0000
Added variable 'RSI_MA' with p-value 0.0000
Added variable 'Close_NFP' with p-value 0.0000
Added variable 'MA_20' with p-value 0.0005
Added variable 'ADX' with p-value 0.0012
Added variable 'MA_50' with p-value 0.0154
Added variable 'MA_100' with p-value 0.0003
Added variable 'RSI_28' with p-value 0.0332
Added variable 'High' with p-value 0.0304
Added variable 'Stoch_D' with p-value 0.0011
Removed variable 'RSI_MA' with p-value 0.1007
Added variable 'MACD' with p-value 0.0465
Removed variable 'ADX' with p-value 0.0606
No additional significant variable found.
Forward Selection: Average RMSE (CV): 13.5026, RMSE (Test): 12.9576

--- Backward Elimination ---
Removed variable 'MA_200' with p-value 0.9594
Removed variable 'Stoch_K' with p-value 0.5337
Removed variable 'Open' with p-value 0.5051
Removed variable 'MACD' with p-value 0.3362
Removed variable 'Low' with p-value 0.2885
Removed variable 'ADX' with p-value 0.0964
Removed variable 'RSI_MA' with p-value 0.0510
Removed variable 'Close_DAX' with p-value 0.0715
All remaining variables are significant.
```

```
Backward Elimination: Average RMSE (CV): 13.4879, RMSE (Test): 13.0248

--- Bidirectional Selection ---
Removed variable 'Stoch_K' with p-value 0.6872
Removed variable 'Open' with p-value 0.5672
Removed variable 'RSI_MA' with p-value 0.0968
Added variable 'Close_NASDAQ' with p-value 0.0000
Added variable 'Close_GOLD' with p-value 0.0000
Added variable 'Close_VIX' with p-value 0.0000
Removed variable 'Low' with p-value 0.0703
Added variable 'Close_NFP' with p-value 0.0000
Added variable 'Close_CPI' with p-value 0.0002
Added variable 'MA_20' with p-value 0.0039
Added variable 'Stoch_D' with p-value 0.0037
No additional significant variable found.
Bidirectional Selection: Average RMSE (CV): 13.4725, RMSE (Test):
13.0793

Final Model Evaluation:
Variables: ['Close', 'Close_GOLD', 'Close_VIX', 'Close_T_chomage',
'Close_NASDAQ', 'Close_CPI', 'Close_NFP', 'MA_20', 'MA_50', 'MA_100',
'RSI_28', 'High', 'Stoch_D', 'MACD']
RMSE on Test Set: 12.9576
```

Forward Selection

1. **Results**:
   - A total of **14 variables** were selected.
   - **Average RMSE (CV)**: 13.5026

   - **RMSE on Test Set**: 12.9576
2. **Key Observations**:
   - Forward Selection produced the lowest RMSE on the **test set**, indicating good generalization performance.
   - Several technical indicators like **'MA_20'**, **'MA_50'**, **'RSI_28'**, and **'Stoch_D'** were included, highlighting their predictive importance.

Backward Elimination

1. **Results**:
   - The process stopped when all remaining variables were statistically significant.
   - **Average RMSE (CV)**: 13.4879

   - **RMSE on Test Set**: 13.0248
2. **Key Observations**:
   - Backward Elimination retained fewer variables compared to Forward Selection, resulting in a slightly higher RMSE on the test set.

Bidirectional Selection

1. **Results**:
   - **Average RMSE (CV)**: 13.4725
   - **RMSE on Test Set**: 13.0793
2. **Key Observations**:
   - Bidirectional Selection produced an RMSE between that of Forward and Backward methods.

Final Model Evaluation

1. **Selected Variables**: The final model combines **14 predictors** identified as significant across the processes:
   - `'Close'`, `'Close_GOLD'`, `'Close_VIX'`, `'Close_T_chomage'`, `'Close_NASDAQ'`, `'Close_CPI'`, `'Close_NFP'`, `'MA_20'`, `'MA_50'`, `'MA_100'`, `'RSI_28'`, `'High'`, `'Stoch_D'`, and `'MACD'`.
2. **Performance**:
   - **RMSE on Test Set: 12.9576** (lowest across all methods).

2) Extracting and Displaying Model Coefficients

```
# Extract coefficients as a pandas series
coefficients_series = pd.Series(final_model.coef_,
index=final_variables)

# Add the intercept to the series
coefficients_series = pd.concat([pd.Series({'Intercept':
final_model.intercept_}), coefficients_series])

# Convert the coefficients to a list
coefficients_list = coefficients_series.tolist()

# Extract variable names (including the intercept)
variables_list = coefficients_series.index.tolist()

# Display the coefficients and variables
print("List of coefficients:", coefficients_list)
print("List of variables:", variables_list)

List of coefficients: [2095.665096215575, 160.61401370299794,
6.6615877405398205, 5.543410805422429, 92.60229279814492,
74.72218664721827, 9.609152166622842, -1.4251805816399714,
8.13151829094371, -6.603440858065985, 38.86744848616163,
63.414048134124435, 111.48467599178556, -12.365904787715303,
2.6654537272042047]
List of variables: ['Intercept', 'Close', 'Close_GOLD', 'Close_VIX',
'Close_T_chomage', 'Close_NASDAQ', 'Close_CPI', 'Close_NFP', 'MA_20',
'MA_50', 'MA_100', 'RSI_28', 'High', 'Stoch_D', 'MACD']
```

1. **Extract Coefficients**:
   The coefficients of the `final_model` are extracted as a `pandas` Series and assigned to the corresponding selected variables (`final_variables`).
2. **Add Intercept**:
   The intercept (constant term) of the model is included as the first element of the coefficients series.
3. **Convert to List**:
   - The coefficients series is converted to a list (`coefficients_list`) to facilitate further use or display.
   - The variable names (including the intercept) are also converted into a list (`variables_list`).
4. **Display**:
   The coefficients and their corresponding variable names are printed for interpretation.

This ensures all model parameters, including the intercept, are clearly extracted, structured, and available for analysis.

PART C: Model Assumption Checks

To ensure the validity and reliability of a regression model, it is essential to verify that key assumptions are met. The primary assumptions include **linearity**, **homoscedasticity**, **independence of variables**, and **normality of residuals**. Each is detailed below:

I) **Linearity**

- **Definition**: Linearity assumes that there is a straight-line relationship between the independent variables (predictors) and the dependent variable. This ensures that the regression model captures the relationship correctly.
- **Why it Matters**: If the relationship is not linear, the model will produce biased estimates and inaccurate predictions.
- **Check**:
  - Plot the predicted values against the residuals. A linear model should show no clear patterns in the residuals; they should be randomly scattered.

II) **Homoscedasticity**

- **Definition**: Homoscedasticity means that the residuals have constant variance across all levels of the predicted values.

- **Why it Matters**: If the variance of residuals is not constant (heteroscedasticity), predictions become unreliable because the model may not account for varying errors in the data.

- **Check**:
  - Plot the residuals against the predicted values. The spread of the residuals should remain consistent (i.e., no funnel or cone-like shapes).

  - Residual plots with patterns (e.g., increasing or decreasing spread) indicate heteroscedasticity.

III) **Independence of Variables**

- **Definition**: Residuals should be independent of one another. There should be no correlation between consecutive residuals. This is particularly important in time-series data.

- **Why it Matters**: If residuals are correlated, it violates the independence assumption, which can inflate Type I errors (false positives) and reduce model efficiency.

- **Check**:
  - Use the **Durbin-Watson test**: Values close to 2 indicate no significant autocorrelation, while values near 0 or 4 suggest positive or negative autocorrelation.

IV) **Normality of Residuals**

- **Definition**: The residuals (errors) should follow a normal distribution, centered around zero.

- **Why it Matters**: Normality of residuals is required for reliable confidence intervals and hypothesis tests. If residuals are not normally distributed, statistical inferences may become invalid.

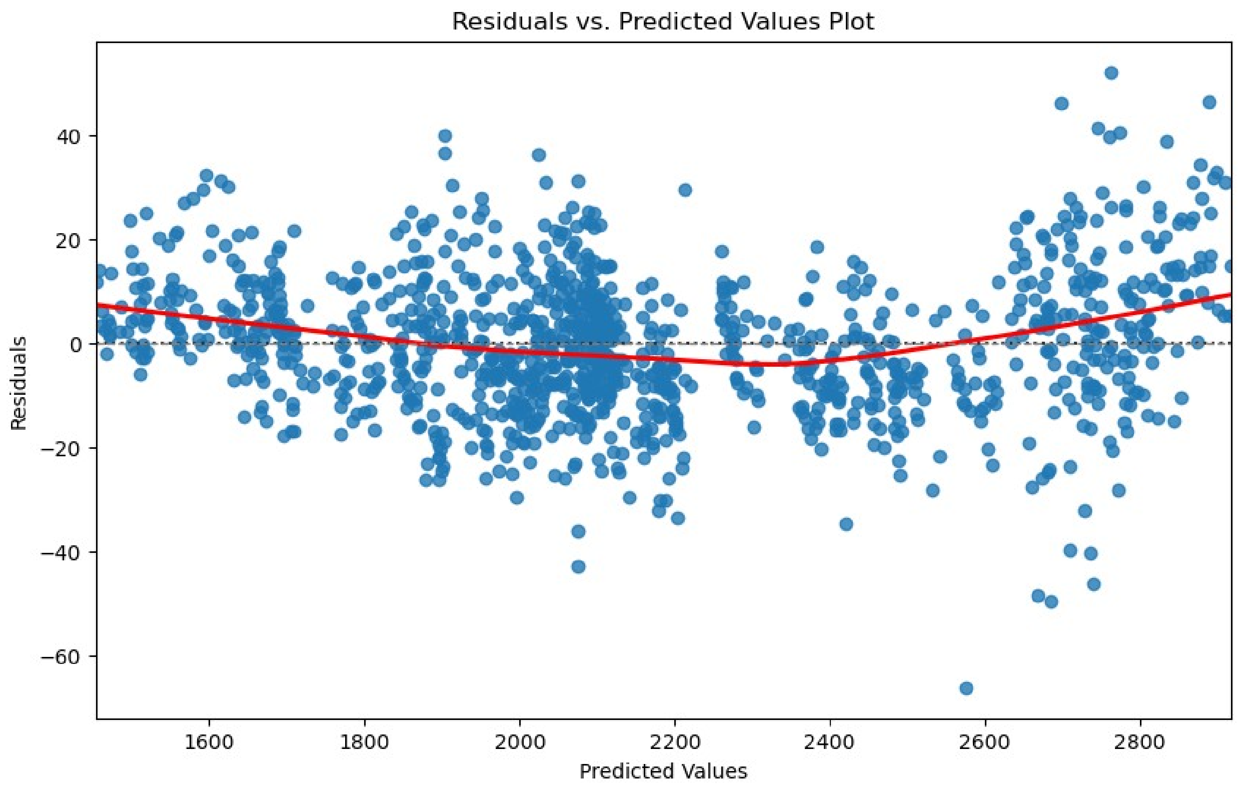
- **Check**:
  - **Q-Q Plot**: A quantile-quantile (Q-Q) plot compares the residuals to a normal distribution. If the points align closely with the diagonal line, the residuals are normally distributed.

Assumption	Method to Verify	Solution if Violated
Linearity	Residual plots	Polynomial terms or transformations
Homoscedasticity	Residual plots	Transformations, robust regression
Independence	Durbin-Watson test	Time-series models, lagged predictors
Normality of Residuals	Q-Q plot	Transformations, alternative regression

By systematically evaluating these assumptions, we ensure that the regression model produces valid and reliable results.

I) Linearity and II) Homoscedasticity Check

```python
# Exclude the intercept (if present in the list)
variables_list_no_intercept = [var for var in variables_list if var !=
'Intercept']

# Create X_train_final excluding the intercept column
X_train_final = X_train[variables_list_no_intercept]

# Calculate predicted values on the training set
Y_train_pred = final_model.predict(X_train_final)

# Calculate residuals
residuals = Y_train - Y_train_pred

# Plot the residuals against the predicted values
plt.figure(figsize=(10, 6))
sns.residplot(x=Y_train_pred, y=residuals, lowess=True,
line_kws={'color': 'red'})
plt.axhline(y=0, color='gray', linestyle='--', linewidth=1)
plt.xlabel('Predicted Values')
plt.ylabel('Residuals')
plt.title('Residuals vs. Predicted Values Plot')
plt.show()
```



1. **Exclude the Intercept**:
   - The intercept term, if present, is removed from the list of variables to ensure it is not treated as a predictor in calculations.

   - This is done using:

```
variables_list_no_intercept = [var for var in
variables_list if var != 'Intercept']
```

2. **Prepare Training Data**:
   - The training predictors (`X_train`) are filtered to include only the relevant variables (excluding the intercept).

```
X_train_final = X_train[variables_list_no_intercept]
```

3. **Predicted Values**:
   - The model calculates predicted values on the training set using the fitted model:

```
Y_train_pred = final_model.predict(X_train_final)
```

4. **Residuals Calculation**:

- Residuals (errors) are calculated as the difference between the actual (`Y_train`) and predicted values (`Y_train_pred`):

```
residuals = Y_train - Y_train_pred
```

5. **Residual Plot**:
   - A **residual plot** is generated to evaluate the assumptions:
     - **Linearity**: If the relationship is linear, residuals should be randomly scattered around zero without any clear patterns.
     - **Homoscedasticity**: Residuals should exhibit constant variance across all predicted values.
   - The `sns.residplot` function plots residuals against the predicted values. The red line (smoothed with `lowess`) helps identify any patterns:

```
sns.residplot(x=Y_train_pred, y=residuals, lowess=True,
line_kws={'color': 'red'})
```

   - Horizontal gray lines (`y=0`) indicate the baseline where residuals should ideally be centered.
   - The plot title and axis labels provide context.

Interpretation of the Plot:

1. **Linearity**:
   - The residuals (blue points) generally cluster around the horizontal axis (`y=0`), indicating a reasonably linear relationship between the predictors and the target variable.
   - Although the **red lowess line** shows a slight curvature, the deviations are minor and do not indicate a significant non-linear pattern.
2. **Homoscedasticity**:
   - The spread of residuals remains relatively consistent across the range of predicted values.
   - There is no clear pattern of increasing or decreasing variance, suggesting that the model satisfies the assumption of homoscedasticity.

By plotting residuals, we visually validate these assumptions and ensure the regression model is appropriately specified.

III) Independence of Variables

```
from statsmodels.stats.stattools import durbin_watson
```

```
# Perform the Durbin-Watson test on the residuals
dw_statistic = durbin_watson(residuals)

print(f"Durbin-Watson statistic: {dw_statistic:.4f}")

Durbin-Watson statistic: 1.9428
```

To verify the independence of residuals, the **Durbin-Watson statistic** was used. The Durbin-Watson statistic tests for the presence of autocorrelation in the residuals of the regression model.

Test Results:

- **Durbin-Watson Statistic**: 1.9428

Interpretation:
The Durbin-Watson statistic ranges from 0 to 4:

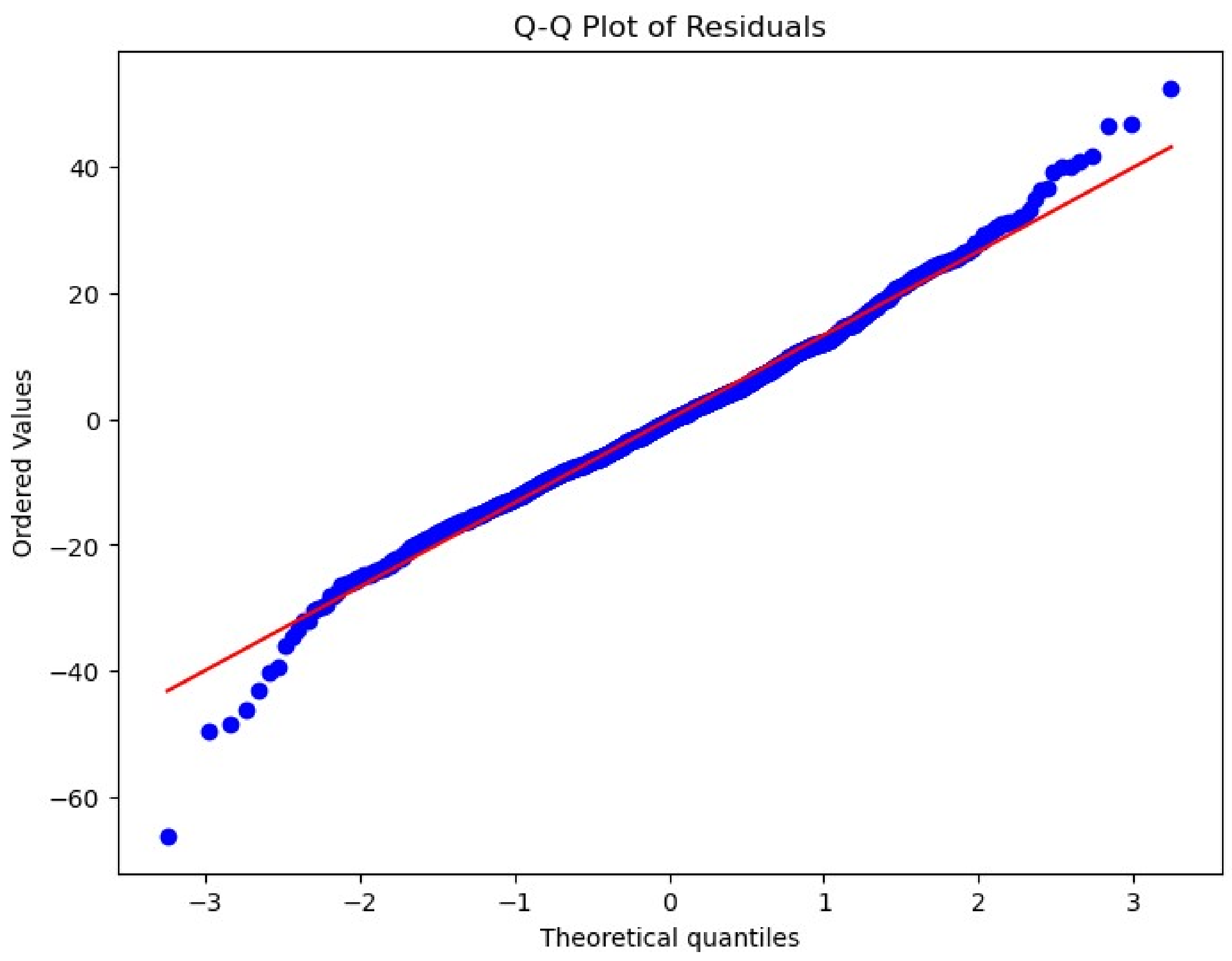
- A value close to 2 suggests no significant autocorrelation.

- Values < 2 indicate positive autocorrelation, while values > 2 indicate negative autocorrelation.

In this case, the **Durbin-Watson statistic of 1.9428** is very close to 2, indicating that **the residuals are independent** and there is no significant autocorrelation. This confirms that the independence assumption for the linear regression model is satisfied.

IV) Normality of Residuals

```
# Plot the Q-Q plot for the residuals
plt.figure(figsize=(8, 6))
stats.probplot(residuals, dist="norm", plot=plt)
plt.title('Q-Q Plot of Residuals')
plt.show()
```



To assess the assumption of **normality of residuals**, a Q-Q (Quantile-Quantile) plot is used. The Q-Q plot compares the distribution of the residuals against a normal distribution.

The Q-Q plot shows the following:

- The majority of the residual points align closely with the red 45-degree reference line, indicating that the residuals largely follow a normal distribution.
- While there are slight deviations at the lower and upper extremes (tails), these are relatively minor and do not significantly impact the overall normality assumption.
- The residuals show a strong adherence to normality across the central range, which validates the assumption of normality for the linear regression model.
- Small deviations at the extremes are common in real-world data and are not a cause for major concern in this case.

PART D: Optimization Algorithm

In predictive modeling, optimization algorithms play a **critical role** in improving the performance and efficiency of machine learning models. The optimization process **adjusts the model's**

parameters (e.g., regression coefficients) to minimize the loss function, ensuring that the model accurately captures relationships within the data.

In our case, where **Ridge Regression** is applied, the **gradient descent algorithm** is essential for optimizing the loss function by iteratively minimizing the error term. The choice of an efficient optimization algorithm becomes particularly important when dealing with **large datasets** or **complex models**, where analytical solutions (like solving for closed-form equations) may be computationally expensive or infeasible.

I) Gradient Descent: Concept and Explanation

Gradient Descent is an **iterative optimization algorithm** used to minimize a loss function by updating the model's parameters in the direction of the steepest descent (negative gradient). It is widely used in machine learning and optimization problems because of its simplicity and efficiency.

The process involves the following steps:

1. **Compute the gradient** (partial derivatives) of the loss function with respect to the model parameters.

2. **Update the parameters** in the direction opposite to the gradient, using a **learning rate** (η) to control the step size.

Gradient Descent Update Rule

$$\theta_j = \theta_j - \eta \frac{\partial J(\theta)}{\partial \theta_j}$$

Where:

- θ_j represents the **model parameters** (coefficients) to be updated.

- η is the **learning rate**, which controls the size of the step taken towards minimizing the loss.

- $J(\theta)$ is the **loss function** that the gradient descent algorithm seeks to minimize.

- $\frac{\partial J(\theta)}{\partial \theta_j}$ is the **gradient of the loss function** with respect to the parameter θ_j, determining the direction and magnitude of the update.

Types of Gradient Descent

Gradient descent comes in three main types:

1. **Batch Gradient Descent**:

   - Computes the gradient using the **entire dataset** at each iteration.
   - **Pros**: Stable convergence due to the use of all data points.
   - **Cons**: Computationally expensive for **large datasets** as it requires processing the entire dataset at once.
2. **Stochastic Gradient Descent (SGD)**:
   - Computes the gradient using a **single data point** at each iteration.
   - **Pros**: Faster updates, making it suitable for very large datasets.
   - **Cons**: High variance in updates, which may result in less stable convergence.
3. **Mini-Batch Gradient Descent** (*Our Choice*):
   - Computes the gradient using a **small batch of data points** (subset) at each iteration.
   - Combines the benefits of batch and stochastic gradient descent:
     - Faster than batch gradient descent.
     - More stable than stochastic gradient descent.
   - Efficient for training on large datasets while maintaining good performance.

The selection of **mini-batch gradient descent** as the optimization method in our project is motivated by the following considerations:

1. **Dataset Size**: Given the significant size of our dataset, it is imperative to utilize an optimization algorithm that is computationally efficient and scalable.
2. **Balance Between Speed and Stability**: Mini-batch gradient descent strikes an optimal balance by combining the advantages of both batch gradient descent and stochastic gradient descent. It ensures faster parameter updates while maintaining stability during the optimization process.
3. **Manageable Data Processing**: By processing data in manageable batches, mini-batch gradient descent reduces computational overhead while ensuring efficient memory utilization. This approach is particularly advantageous for large datasets, as it facilitates smooth convergence toward the optimal solution.

```
# Extract test data for the selected variables
X_test_final = X_test[variables_list_no_intercept].values  # Ensure
the columns are in the correct order

# prediction function
def predict(X, weights, intercept):
    return np.dot(X, weights) + intercept
```

```python
# function to calculat the RMSE
def compute_rmse(y_true, y_pred):
    return np.sqrt(np.mean((y_true - y_pred) ** 2))
```

1) Prediction Function

```python
def predict(X, weights, intercept):
    return np.dot(X, weights) + intercept
```

- **Purpose**: The `predict` function calculates the predicted values ((\hat{y})) for a given input feature matrix (X), based on the model's learned parameters.
- **How It Works**:
  - The function computes the **dot product** between the feature matrix (X) (a 2D array) and the model's weights (coefficients) represented by the `weights` parameter (a 1D array).
  - The resulting dot product represents a linear combination of input features and their corresponding coefficients.
  - Finally, the **intercept** (bias term) is added to each predicted value, ensuring that the model can fit data that does not pass through the origin.
- **Formula**:

$$\hat{y} = X \cdot w + b$$

  Where:
  - X: Feature matrix (input).
  - w: Weights (coefficients).
  - b: Intercept (bias term).
  - $\hat{y}$: Predicted values.

---

2) Function to Calculate the RMSE

```python
def compute_rmse(y_true, y_pred):
    return np.sqrt(np.mean((y_true - y_pred) ** 2))
```

- **Purpose**: The `compute_rmse` function calculates the **Root Mean Squared Error (RMSE)**, a metric commonly used to measure the accuracy of regression models.
- **How It Works**:

- The function first calculates the **difference** (residuals) between the actual values (`y_true`) and the predicted values (`y_pred`).
- Each residual is then squared to ensure all errors are positive.
- The mean of the squared residuals is computed to determine the average squared error.
- Finally, the square root of this mean is taken to return the RMSE, which provides the error in the same units as the target variable.

II) Explanation of the `mini_batch_gradient_descent` Function

```
def mini_batch_gradient_descent(X, y, initial_weights,
initial_intercept, learning_rate=0.001, epochs=1000, batch_size=32):
    n_samples, n_features = X.shape
    weights = initial_weights.copy()  # Initialize weights from final
values
    intercept = initial_intercept  # Initialize intercept from final
value

    rmse_history = []

    for epoch in range(epochs):
        # Shuffle the data for each epoch
        indices = np.random.permutation(n_samples)
        X_shuffled = X[indices]
        y_shuffled = y[indices]

        # Divide the data into mini-batches
        for i in range(0, n_samples, batch_size):
            X_batch = X_shuffled[i:i + batch_size]
            y_batch = y_shuffled[i:i + batch_size]

            # Predict current values for the mini-batch
            y_pred = predict(X_batch, weights, intercept)

            # Compute the error for the mini-batch
            error = y_pred - y_batch

            # Compute gradients for the mini-batch
            weights_gradient = (1 / len(y_batch)) * np.dot(X_batch.T,
error)
            intercept_gradient = (1 / len(y_batch)) * np.sum(error)

            # Update coefficients
            weights -= learning_rate * weights_gradient
            intercept -= learning_rate * intercept_gradient

        # Compute global RMSE for the epoch
        y_pred_epoch = predict(X, weights, intercept)
        rmse = compute_rmse(y, y_pred_epoch)
        rmse_history.append(rmse)

        # Display RMSE every 100 iterations
        if epoch % 100 == 0:
            print(f"Epoch {epoch}: RMSE = {rmse}")

    return weights, intercept, rmse_history
```

#### Key Inputs

- **X**: Input feature matrix (each row represents a data point, and each column represents a feature).
- **y**: Target variable (actual values).
- **`initial_weights`**: Initial values of the model's coefficients.
- **`initial_intercept`**: Initial value of the model's bias (intercept).
- **`learning_rate`**: Controls the step size of parameter updates (default is 0.001).
- **`epochs`**: The number of iterations (full passes over the dataset) for training the model.
- **`batch_size`**: The size of each mini-batch used to compute gradients.

Steps and Functionality

1. **Initialization**:
   - The function initializes weights and intercept to their provided starting values.
   - An empty list `rmse_history` is created to store the RMSE (Root Mean Squared Error) values at each epoch.
2. **Epoch Loop**:
   - For each epoch, the training data is **shuffled** to ensure randomness and avoid bias during mini-batch updates.
   - The shuffled data is divided into **mini-batches** of size `batch_size`.
3. **Mini-Batch Processing**:
   - For each mini-batch:
     - The **predicted values** for the batch are computed using the `predict` function.
     - The **error** (difference between predicted and actual values) is calculated.

- **Gradients** for the weights and intercept are computed:
  - **Weights Gradient**: The average gradient for the coefficients is derived as:

$$\nabla_{\text{weights}} = \frac{1}{\text{batch size}} \cdot X_{\text{batch}}^T \cdot \left(y_{\text{pred}} - y_{\text{batch}} \right)$$

  - **Intercept Gradient**: The average gradient for the bias term is:

$$\nabla_{\text{intercept}} = \frac{1}{\text{batch size}} \cdot \sum \left(y_{\text{pred}} - y_{\text{batch}} \right)$$

- The **weights** and **intercept** are updated using the gradients:

$$\text{weights} = \text{weights} - \eta \cdot \nabla_{\text{weights}}$$

$$\text{intercept} = \text{intercept} - \eta \cdot \nabla_{\text{intercept}}$$

4. **Global RMSE Calculation**:
   - At the end of each epoch, the model's performance is evaluated using the **global RMSE** on the full dataset.
   - The RMSE value is stored in `rmse_history`.
5. **Progress Display**:
   - The function prints the RMSE every 100 epochs to monitor training progress.
6. **Return Values**:
   - **Updated weights**: Final optimized coefficients.
   - **Updated intercept**: Final optimized bias term.
   - **RMSE history**: A list of RMSE values for each epoch to track model improvement.

III) Model Performance Metrics

In this section, we evaluate the performance of the regression model using several commonly used metrics. Each metric provides a unique perspective on the model's accuracy, error, and overall performance.

---

1. **Mean Absolute Error (MAE)**

- **Definition**:
  The **Mean Absolute Error (MAE)** measures the average absolute difference

between the predicted values ($\hat{y}$) and the actual values (y_{true}). It gives an indication of how far the predictions are, on average, from the true values.

- **Formula**:

$$\text{MAE} = \frac{1}{n} \sum_{i=1}^{n} \left| y_{\text{true},i} - y_{\text{pred},i} \right|$$

- **Characteristics**:
  - MAE treats all errors equally because it uses the absolute value of the differences.
  - It is **simple to interpret** and is expressed in the same units as the target variable.
- **Interpretation**:
  A lower MAE value indicates that the model's predictions are closer to the actual values.

---

2. **R-Squared Score (R^2)**

- **Definition**:
  The **R-squared score** (also known as the coefficient of determination) measures the proportion of variance in the target variable (y_{true}) that is explained by the model's predictions ($\hat{y}$).

- **Formula**:

$$R^2 = 1 - \frac{\sum_{i=1}^{n} \left(y_{\text{true},i} - y_{\text{pred},i} \right)^2}{\sum_{i=1}^{n} \left(y_{\text{true},i} - \acute{y}_{\text{true}} \right)^2}$$

  Where:

  - $y_{\text{true},i}$: Actual values.
  - $y_{\text{pred},i}$: Predicted values.
  - $\acute{y}_{\text{true}}$: Mean of the actual values.
  - n: Number of data points.
- **Characteristics**:

  - The R^2 score ranges from $-\infty$ to 1.
  - An R^2 value of 1 indicates a **perfect fit** where the model explains 100% of the variance in the target variable.
  - A negative R^2 indicates that the model performs worse than simply predicting the mean of the target variable.
- **Interpretation**:
  A higher R^2 value indicates that the model can explain more of the variance in the target variable.

---

3. **Root Mean Squared Error (RMSE)**

- **Definition**:
  The **Root Mean Squared Error (RMSE)** measures the square root of the average squared differences between the predicted values and the actual values.
- **Formula**:

$$\text{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(y_{\text{true},i} - y_{\text{pred},i}\right)^2}$$

  Where:
  - $y_{\text{true},i}$: Actual values.
  - $y_{\text{pred},i}$: Predicted values.
  - n: Number of data points.
- **Characteristics**:
  - RMSE **penalizes larger errors** more heavily due to the squaring operation, making it sensitive to outliers.
  - It is expressed in the same units as the target variable, which makes it easy to interpret.
- **Interpretation**:
  A lower RMSE value indicates better performance, as it reflects that the predictions are closer to the actual values.

---

4. **Relative Accuracy**

- **Definition**:
  The **relative accuracy** measures how close the model's predictions are to the actual values as a percentage. It quantifies the model's performance relative to the target values.

- **Formula**:

$$\text{Relative Accuracy} = 100 \cdot \left(1 - \frac{1}{n} \sum_{i=1}^{n} \left| \frac{y_{\text{true},i} - y_{\text{pred},i}}{y_{\text{true},i}} \right| \right)$$

  Where:

  – $y_{\text{true},i}$: Actual values.

  – $y_{\text{pred},i}$: Predicted values.

  – n: Number of data points.

- **Characteristics**:

  – Relative accuracy is expressed as a **percentage**.

  – It directly compares the magnitude of the errors to the actual values, making it useful for interpreting the model's performance.

- **Interpretation**:
  A higher relative accuracy (closer to 100%) indicates that the model's predictions are very close to the actual values.

Metric	Purpose	Ideal Value	Interpretation
MAE	Measures the average magnitude of errors.	Lower is better.	Average absolute deviation from the true values.
R^2	Measures the proportion of variance explained.	Closer to 1.0.	Higher values indicate better model fit.
RMSE	Penalizes large errors, measuring total error.	Lower is better.	Reflects error magnitude in target variable units.
Relative Accuracy	Measures prediction closeness as a percentage.	Closer to 100%.	Higher values indicate greater prediction accuracy.

IV) Evaluation of Model Performance Before Optimization

```
intercept = coefficients_list[0]
weights = coefficients_list[1:]

# Compute predictions using the optimized weights and intercept
y_pred_before = predict(X_test_final, weights, intercept)

# Compute MAE
mae_before = mean_absolute_error(Y_test, y_pred_before)

# Compute R^2
r2_before = r2_score(Y_test, y_pred_before)

# Compute RMSE
rmse_before = compute_rmse(Y_test, y_pred_before)

# Compute relative accuracy
relative_accuracy_before = 100 * (1 - np.mean(np.abs((Y_test - 
y_pred_before) / Y_test)))
```

In this step, we evaluate the model's performance on the test dataset using the **initial weights** and **intercept** extracted from the coefficients. The results will serve as a **baseline** to compare against the optimized model after applying gradient descent.

Key Steps

1. **Extract Model Parameters**:
   - The intercept (bias term) is assigned to `coefficients_list[0]`.
   - The remaining elements of `coefficients_list` represent the model's weights (coefficients).
2. **Predictions**:
   - Predicted values for the test dataset `X_test_final` are computed using the `predict` function with the initial weights and intercept.
3. **Performance Metrics**:
   The following metrics are calculated to quantify the model's baseline performance:
   - **Mean Absolute Error (MAE)**: Average magnitude of prediction errors.
   - **R-Squared Score (R^2)**: Proportion of variance in the target variable explained by the model.
   - **Root Mean Squared Error (RMSE)**: Standard deviation of prediction errors.

- **Relative Accuracy:** Measures prediction closeness to actual values as a percentage.

These baseline metrics provide a clear understanding of the model's initial performance **before optimization.** By comparing these values with the results after gradient descent, we can assess the improvements achieved through the optimization process.

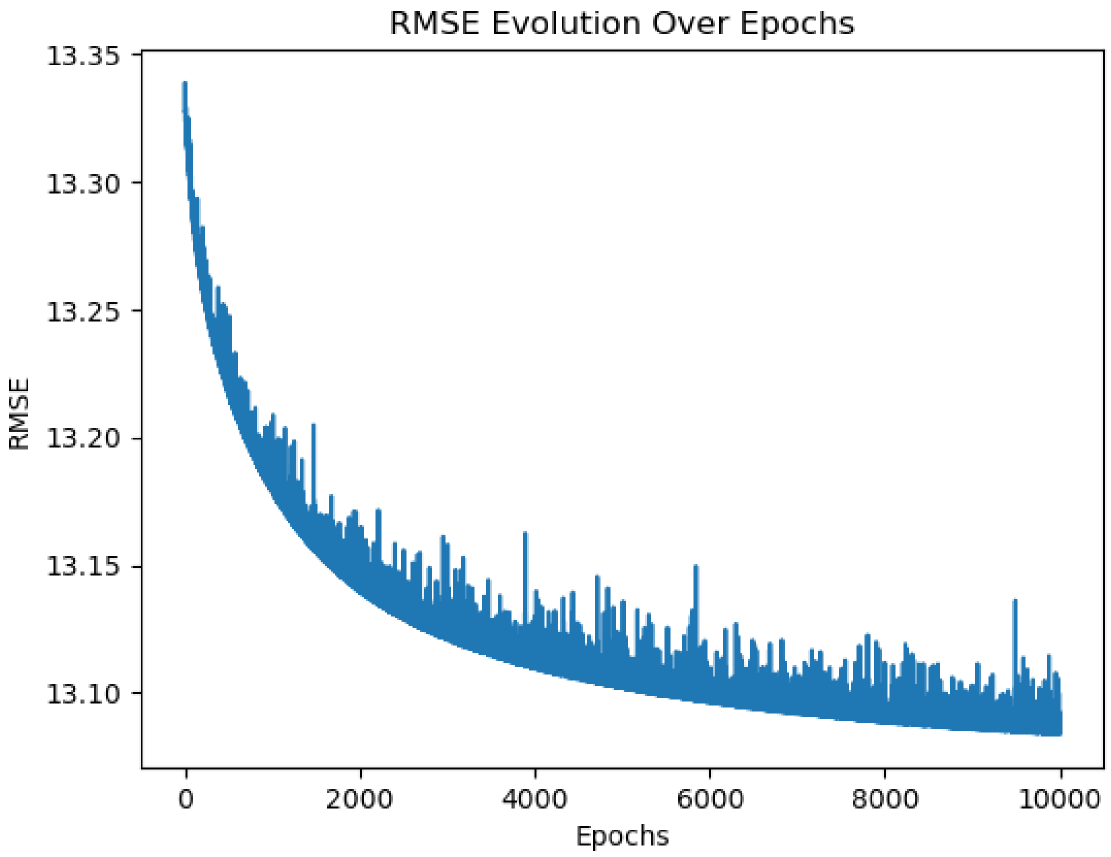
V) Evaluation of Model Performance After Optimization

1) we update the coefs using the training set

```
# Initialize the coefficients and intercept from the final model
initial_weights = weights  # Excludes the intercept
initial_intercept = intercept  # Takes the intercept
variables_list = variables_list[1:]  # List of variables without
intercept

# Prepare data for gradient descent
# Ensure X_train contains only the final variables (in the correct
order)
X_train_final = X_train[variables_list].values  # Convert to numpy
array for matrix computation
Y_train = Y_train.values  # Convert Y_train to an array if necessary

# Apply mini-batch gradient descent with initial values
learning_rate = 0.01
epochs = 10000
batch_size = 32  # Mini-batch size
final_weights, final_intercept, rmse_history =
mini_batch_gradient_descent(X_train_final, Y_train, initial_weights,
initial_intercept, learning_rate, epochs, batch_size)

# Display final results
print("Optimized coefficients:", final_weights)
print("Optimized intercept:", final_intercept)
print("Final RMSE:", rmse_history[-1])

plt.plot(rmse_history)
plt.xlabel("Epochs")
plt.ylabel("RMSE")
plt.title("RMSE Evolution Over Epochs")
plt.show()
```

```
Epoch 0: RMSE = 13.327367728061962
Epoch 100: RMSE = 13.281426091274827
Epoch 200: RMSE = 13.269757082970862
Epoch 300: RMSE = 13.240267270281867
Epoch 400: RMSE = 13.227429006452368
Epoch 500: RMSE = 13.216677441920224
Epoch 600: RMSE = 13.207050581981587
Epoch 700: RMSE = 13.198450277197464
Epoch 800: RMSE = 13.191752858567131
Epoch 900: RMSE = 13.18674018546858
Epoch 1000: RMSE = 13.178991959819408
Epoch 1100: RMSE = 13.172539503649219
Epoch 1200: RMSE = 13.169511641665213
Epoch 1300: RMSE = 13.1684629006996
Epoch 1400: RMSE = 13.161603560273276
Epoch 1500: RMSE = 13.158479008895016
Epoch 1600: RMSE = 13.152104298099694
Epoch 1700: RMSE = 13.148002004629992
Epoch 1800: RMSE = 13.151004358830113
Epoch 1900: RMSE = 13.142217862784495
Epoch 2000: RMSE = 13.140435207952201
Epoch 2100: RMSE = 13.139794279487136
Epoch 2200: RMSE = 13.134757518575647
Epoch 2300: RMSE = 13.132591050848298
Epoch 2400: RMSE = 13.131260679270705
Epoch 2500: RMSE = 13.1485573836013
Epoch 2600: RMSE = 13.130625385999643
Epoch 2700: RMSE = 13.125205536382477
Epoch 2800: RMSE = 13.125981715084215
Epoch 2900: RMSE = 13.122951242399232
Epoch 3000: RMSE = 13.121471443005598
Epoch 3100: RMSE = 13.1241413898614
Epoch 3200: RMSE = 13.126308164315747
Epoch 3300: RMSE = 13.117194728822692
Epoch 3400: RMSE = 13.115806242271525
Epoch 3500: RMSE = 13.118685092683297
Epoch 3600: RMSE = 13.113342785110408
Epoch 3700: RMSE = 13.118249313394267
Epoch 3800: RMSE = 13.122248940650293
Epoch 3900: RMSE = 13.111049473016283
Epoch 4000: RMSE = 13.113141658216268
Epoch 4100: RMSE = 13.108563592842211
Epoch 4200: RMSE = 13.109081293962287
Epoch 4300: RMSE = 13.121776450836677
Epoch 4400: RMSE = 13.113808677283082
Epoch 4500: RMSE = 13.106710953176512
Epoch 4600: RMSE = 13.10472720286127
Epoch 4700: RMSE = 13.105681608167147
Epoch 4800: RMSE = 13.105376228718432
Epoch 4900: RMSE = 13.107645032224502
Epoch 5000: RMSE = 13.102758199931934
Epoch 5100: RMSE = 13.102317663609577
Epoch 5200: RMSE = 13.100897217554813
Epoch 5300: RMSE = 13.1012703950847
Epoch 5400: RMSE = 13.104417389516861
Epoch 5500: RMSE = 13.099875464929928
Epoch 5600: RMSE = 13.098914695367764
Epoch 5700: RMSE = 13.097788653555819
Epoch 5800: RMSE = 13.132378950664794
Epoch 5900: RMSE = 13.118054756551135
Epoch 6000: RMSE = 13.096691657887828
Epoch 6100: RMSE = 13.096339318070234
Epoch 6200: RMSE = 13.097952317372158
Epoch 6300: RMSE = 13.094818982503858
Epoch 6400: RMSE = 13.095400655427097
Epoch 6500: RMSE = 13.11149440753463
Epoch 6600: RMSE = 13.09933842407896
Epoch 6700: RMSE = 13.094759488299388
Epoch 6800: RMSE = 13.098633135660831
Epoch 6900: RMSE = 13.093438458779282
Epoch 7000: RMSE = 13.094797650299503
Epoch 7100: RMSE = 13.092643960812087
Epoch 7200: RMSE = 13.101055604934286
Epoch 7300: RMSE = 13.094163687464233
Epoch 7400: RMSE = 13.0942025634231l
Epoch 7500: RMSE = 13.091208422539346
Epoch 7600: RMSE = 13.090013581049574
Epoch 7700: RMSE = 13.108278831795134
Epoch 7800: RMSE = 13.091630107853478
Epoch 7900: RMSE = 13.091615583542678
Epoch 8000: RMSE = 13.089820290060045
Epoch 8100: RMSE = 13.088165202176965
Epoch 8200: RMSE = 13.087976858377454
Epoch 8300: RMSE = 13.088431114737773
Epoch 8400: RMSE = 13.10209950359728
Epoch 8500: RMSE = 13.088322655444685
Epoch 8600: RMSE = 13.086784751334347
Epoch 8700: RMSE = 13.087029105476741
Epoch 8800: RMSE = 13.087924957992437
Epoch 8900: RMSE = 13.086354235888644
Epoch 9000: RMSE = 13.08593128616955
Epoch 9100: RMSE = 13.085580728677652
Epoch 9200: RMSE = 13.086073347268796
Epoch 9300: RMSE = 13.08589692590587
Epoch 9400: RMSE = 13.085359347397304
Epoch 9500: RMSE = 13.099277878981445
Epoch 9600: RMSE = 13.084837316686734
Epoch 9700: RMSE = 13.089293740459219
Epoch 9800: RMSE = 13.08624618425575
Epoch 9900: RMSE = 13.084436411300706
Optimized coefficients: [209.32969019   6.67115498   6.63035829
121.27980387  67.52170979
  10.57675093  -1.51099671  32.46706685 -31.78206851  32.42440106
  63.54731379  84.4542611  -13.85840206   1.29611889]
```

```
Optimized intercept: 2084.8419010445677
Final RMSE: 13.085080916305591
```



1. **Initialization of Model Parameters**:
   - The **weights** and **intercept** are initialized using the previously obtained pre-optimized values.

   - The intercept is extracted as the first element of the coefficients list, and the remaining elements represent the model weights.
2. **Data Preparation**:
   - The input data (`X_train`) is filtered to include only the selected features, ensuring alignment with the initialized weights.

   - Both the input matrix `X_train` and the target variable `Y_train` are converted to **NumPy arrays** to facilitate efficient matrix operations during gradient descent.

The **Mini-Batch Gradient Descent** algorithm is applied with the following configuration:

- **Learning Rate**: $\eta = 0.01 $
- **Epochs**: 10,000 iterations

- **Batch Size**: 32 (mini-batch size)

At each epoch:

- The model updates the **weights** and **intercept** by calculating gradients based on randomly sampled mini-batches of the training data.

- The **RMSE** is computed on the full training dataset and recorded to monitor the model's performance over time.

The evolution of the **RMSE** over the epochs is visualized to illustrate the convergence process. The plot reveals the gradual reduction in error, with minor fluctuations typical of mini-batch gradient descent due to the randomness in batch selection.

Final Results

The optimized model parameters and performance metrics are as follows:

- **Optimized Weights**:
  The final coefficients reflect the importance and contribution of each input feature in predicting the target variable.

- **Optimized Intercept**:
  The bias term that adjusts the model predictions is approximately **2084.84**.

- **Final RMSE**:
  The Root Mean Squared Error after 10,000 epochs is approximately **13.085**, indicating the average magnitude of the model's prediction errors.

```
Optimized coefficients: [209.33, 6.67, 6.63, 121.28, 67.52, 10.58, -
1.51, 32.47, -31.78, 32.42, 63.55, 84.45, -13.86, 1.30]
Optimized intercept: 2084.84
Final RMSE: 13.085
```

Observations

1. **RMSE Reduction**:
   - The RMSE decreases steadily from an initial value of **13.33** at epoch 0 to **13.085** at epoch 10,000.

   - The convergence process demonstrates the algorithm's ability to iteratively improve the model's performance.
2. **Fluctuations**:
   - Minor oscillations in RMSE occur during the optimization process, which is expected in mini-batch gradient descent due to batch-level noise.
3. **Convergence**:

- The gradual reduction and stabilization of RMSE suggest that the algorithm has successfully converged to a near-optimal solution.

As a conclusion, The **Mini-Batch Gradient Descent** algorithm effectively optimized the model parameters, achieving a significant reduction in RMSE. The convergence plot provides a clear visual representation of the learning process, highlighting the algorithm's efficiency in improving the model's performance over successive epochs.

VI) Evaluation of Model Performance After Optimization

```python
# Compute predictions using the optimized weights and intercept
y_pred = predict(X_test_final, final_weights, final_intercept)

# Compute MAE
mae_after = mean_absolute_error(Y_test, y_pred)

# Compute R^2
r2_after = r2_score(Y_test, y_pred)

# Compute RMSE
rmse_after = compute_rmse(Y_test, y_pred)

# Compute relative accuracy
relative_accuracy_after = 100 * (1 - np.mean(np.abs((Y_test - y_pred)
/ Y_test)))
```

The optimized weights and intercept are used to generate predictions on the test dataset. The performance of the model is then reassessed to observe the improvements achieved through the optimization process.

```python
print("\nPerformance Metrics:")
print("Mean Absolute Error (MAE) before :", mae_before)
print("Mean Absolute Error (MAE) after : ", mae_after, "\n")

print("Coefficient of Determination (R^2) before :", r2_before)
print("Coefficient of Determination (R^2) after : ", r2_after, "\n")

print("RMSE before :", rmse_before)
print("RMSE after : ", rmse_after, "\n")

print("Relative Accuracy before :", relative_accuracy_before, "%")
print("Relative Accuracy after : ", relative_accuracy_after, "%")
```

```
Performance Metrics:
Mean Absolute Error (MAE) before : 10.392124184600956
Mean Absolute Error (MAE) after :  10.012033908321024

Coefficient of Determination (R^2) before : 0.998764844629593
Coefficient of Determination (R^2) after :  0.9988402212150296
```

```
RMSE before : 12.957627769065809
RMSE after :  12.556028260545332

Relative Accuracy before : 99.51961009120694 %
Relative Accuracy after :  99.53296397462005 %
```

The following table summarizes the model's performance metrics before and after applying the **Mini-Batch Gradient Descent** optimization.

Metric	Before Optimization	After Optimization	Improvement
Mean Absolute Error (MAE)	10.39	10.01	Decreased by 0.38
Coefficient of Determination (R^2)	0.99876	0.99884	Slight improvement
Root Mean Squared Error (RMSE)	12.96	12.56	Decreased by 0.40
Relative Accuracy (%)	99.52	99.53	Improved by 0.01%

Observations:

1. **Error Reduction**: Both the **MAE** and **RMSE** decreased after optimization, indicating improved prediction accuracy.

2. **R^2 Improvement**: The **Coefficient of Determination** shows a slight increase, reflecting a better fit of the model to the data.

3. **Relative Accuracy**: The relative accuracy increased slightly, showing minimal but measurable improvement in the model's overall performance.

These results confirm that the optimization process successfully enhanced the model's performance, reducing error and improving accuracy.

PART E: Final Regression Equation

```
# Generate the final equation as a readable string
equation = f"High_tomorrow = {final_intercept:.4f} * Intercept"  #
Include the intercept first

# Add terms corresponding to the variables
for coef, var in zip(final_weights, variables_list):
    equation += f" + {coef:.4f} * {var}"

# Display the final equation
print("Final equation:")
print(equation)
```

```
Final equation:
High_tomorrow = 2084.8419 * Intercept + 209.3297 * Close + 6.6712 *
Close_GOLD + 6.6304 * Close_VIX + 121.2798 * Close_T_chomage + 67.5217
* Close_NASDAQ + 10.5768 * Close_CPI + -1.5110 * Close_NFP + 32.4671 *
MA_20 + -31.7821 * MA_50 + 32.4244 * MA_100 + 63.5473 * RSI_28 +
84.4543 * High + -13.8584 * Stoch_D + 1.2961 * MACD
```

SECTION III

Motivation for Comparing Linear Regression and Decision Tree Models

In this section, we compare the performance of the **Linear Regression model** with the **Decision Tree Regressor**. While Linear Regression assumes a **linear relationship** between the input features and the target variable, the Decision Tree Regressor can **introduce non-linearity** by performing recursive splits in the feature space. This makes Decision Trees particularly effective in capturing complex, non-linear patterns in the data that Linear Regression may fail to model.

By comparing these two approaches, we aim to:

- Assess whether introducing **non-linearity** improves predictive accuracy.
- Evaluate the trade-offs in terms of model interpretability, performance, and overfitting risk.

PART A: Decision Tree Model

I) Overview

A **Decision Tree** is a non-parametric, supervised machine learning model used for both **classification** and **regression** tasks. It works by recursively splitting the data into subsets based on the values of input features, creating a tree-like structure.

Key Concepts

1. **Structure**:
   - A Decision Tree consists of **nodes**, **branches**, and **leaves**:
     - **Root Node**: The top node where the first split occurs.
     - **Internal Nodes**: Represent decisions based on feature values.
     - **Branches**: Connect nodes and represent the outcomes of decisions.
     - **Leaf Nodes**: Represent the final predictions (outputs) for a given input.

2. **Splitting Criteria**:
   - The model splits data at each node using a specific criterion to minimize impurity (the degree of mix between target values):
     - **For Classification**: Measures like *Gini Index* or *Entropy* are used.
     - **For Regression**: The model minimizes the **variance** of target values in subsets.
3. **Recursive Partitioning**:
   - The tree-building process is iterative and recursive:
     - At each step, the feature and threshold that best split the data are chosen.
     - The process stops when a stopping criterion is met (e.g., maximum depth, minimum samples per node).
4. **Advantages**:
   - Easy to interpret and visualize.
   - Handles both numerical and categorical data.
   - Requires minimal data preprocessing (e.g., no need for scaling).
   - Captures complex, non-linear relationships in the data.
5. **Disadvantages**:
   - Prone to **overfitting**, especially on noisy or small datasets.
   - Highly sensitive to changes in the data (small variations can result in a different tree).
   - Less effective for extrapolation beyond the training data.

II) Decision Tree Regressor

The **Decision Tree Regressor** is a specialized type of Decision Tree model used for regression tasks, where the target variable is continuous.

1) How It Works:

1. **Splitting Rule**:
   - At each node, the feature and split point are chosen to minimize the variance of the target values in the resulting subsets.
   - The variance reduction is computed as:

$$\Delta\,\mathrm{Var} = \mathrm{Var}_{\mathrm{parent}} - \left(\frac{n_{\mathrm{left}}}{n_{\mathrm{parent}}} \cdot \mathrm{Var}_{\mathrm{left}} + \frac{n_{\mathrm{right}}}{n_{\mathrm{parent}}} \cdot \mathrm{Var}_{\mathrm{right}} \right)$$

Where:

- $\mathrm{Var}_{\text{parent}}$: Variance of the parent node.
- $\mathrm{Var}_{\text{left}}$ and $\mathrm{Var}_{\text{right}}$: Variance of the left and right child nodes.
- n_{left} and n_{right}: Number of samples in the left and right subsets.

2. **Prediction**:
   – The Decision Tree Regressor predicts the output for a given input by traversing the tree until it reaches a leaf node.

   – The prediction is the **mean** of the target values in that leaf node.
3. **Stopping Criteria**:
   – To prevent the tree from growing too large and overfitting, stopping criteria can be applied, such as:
   - Maximum depth of the tree.
   - Minimum number of samples per leaf.
   - Minimum reduction in variance required for a split.

---

2) Advantages of Decision Tree Regressor

1. **Interpretability**:
   – The tree structure is easy to visualize and interpret, making it suitable for explaining predictions.
2. **Non-Linearity**:
   – It can model non-linear relationships between input features and the target variable effectively.
3. **Feature Importance**:
   – Provides a measure of feature importance by evaluating the contribution of each feature to variance reduction.
4. **No Scaling Required**:
   – It does not require normalization or standardization of input features.
5. **Handles Missing Values**:
   – Decision Trees can handle missing values during training without the need for imputation.

---

3) Disadvantages of Decision Tree Regressor

1. **Overfitting**:
   – Decision Trees tend to overfit the training data if not pruned or regularized, leading to poor generalization.
2. **Instability**:
   – Small variations in the data can lead to a completely different tree structure.

3. **Bias Toward Features with More Levels**:
   - Features with a larger number of distinct values can dominate splits, leading to biased results.
4. **Limited Extrapolation**:
   - Decision Trees are ineffective for predicting values outside the range of training data.

III) Hyperparameter Tuning for Decision Tree Regressor

In this section, we focus on finding the optimal configuration of the **Decision Tree Regressor** by tuning its hyperparameters using **Grid Search**. Hyperparameter optimization is crucial to ensure that the model performs at its best without overfitting or underfitting the data.

---

1) Hyperparameters Considered

1. **Criterion**:
   The function used to measure the quality of a split. The following criteria are evaluated:
   - **Squared Error**: Minimizes the variance of the target variable in each split.
   - **Friedman MSE**: An enhancement of MSE that is more robust against noisy data.
   - **Absolute Error**: Minimizes the absolute differences between the target values and their predictions, making it more robust to outliers.
   - **Poisson**: Used for data that follows a Poisson distribution, particularly effective for count data.
2. **Max Depth**:
   Controls the maximum depth of the decision tree. A higher depth allows the model to capture more details in the data but increases the risk of overfitting.
   - **Range**: We explore depths from **1 to 10** to balance model complexity and performance.

---

2) Grid Search Procedure

- **Definition**: Grid Search is a systematic method for hyperparameter optimization that exhaustively evaluates all possible combinations of specified hyperparameter values to find the best-performing configuration.

- **Process**:
  - A predefined set of values is selected for each hyperparameter (e.g., `Criterion` and `Max Depth` in our case).
  - The model is trained and evaluated for each combination of these hyperparameters, ensuring every possible configuration is tested.

- **Cross-Validation**:

- To ensure robust evaluation, **cross-validation** is used:
    - The data is split into multiple subsets (folds).
    - For each combination of hyperparameters, the model is trained on (k-1) folds and validated on the remaining fold.
    - This process is repeated (k) times (once for each fold), and the average performance across all folds is calculated.
    - Cross-validation reduces the risk of overfitting during hyperparameter tuning by ensuring the model is evaluated on different portions of the data.

---

3) Purpose

The hyperparameter tuning process aims to optimize the Decision Tree Regressor for our specific dataset, ensuring that it captures the underlying patterns while avoiding overfitting. By systematically evaluating various configurations and metrics, we identify the setup that provides the best balance between accuracy and complexity.

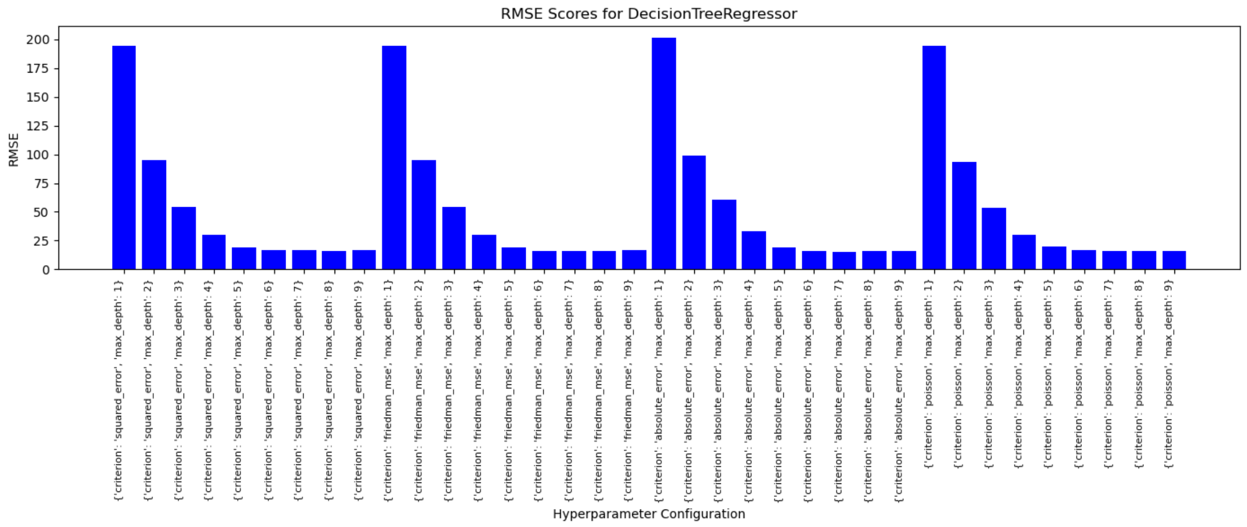
```
from sklearn.tree import DecisionTreeRegressor
from sklearn.model_selection import GridSearchCV

# Define hyperparameters for DecisionTreeRegressor
param_grid = {
    "criterion": ["squared_error", "friedman_mse", "absolute_error",
"poisson"],
    "max_depth": range(1, 10)
}

# Define evaluation metrics
scoring = {
    'mae': make_scorer(mean_absolute_error, greater_is_better=False),
    'rmse': make_scorer(mean_squared_error, greater_is_better=False,
squared=False),
    'r2': make_scorer(r2_score)
}

# Initialize the model
model = DecisionTreeRegressor()

# GridSearchCV for hyperparameter tuning
search = GridSearchCV(
    estimator=model,
    param_grid=param_grid,
    cv=3,
    scoring=scoring,
    refit='r2'  # Optimizes for the R^2 score
)

# Train the model with the full training set
search.fit(X_train, Y_train)

# Store results in a list
results = []
for i in range(len(search.cv_results_['params'])):
    params = search.cv_results_['params'][i]
    mae = search.cv_results_['mean_test_mae'][i]
    rmse = search.cv_results_['mean_test_rmse'][i]
    r2 = search.cv_results_['mean_test_r2'][i]
    results.append({
        'params': params,
        'mae': -mae,  # Reverse the sign because MAE is negative in
GridSearchCV
        'rmse': -rmse,
        'r2': r2
    })

# Convert results to a DataFrame
results_df = pd.DataFrame(results)
print("Overview of hyperparameter tuning results:")
print(results_df)

# Visualize results
# Create a readable identifier for each configuration on the x-axis
results_df['config'] = results_df['params'].apply(lambda x: str(x))

# Bar chart for RMSE
plt.figure(figsize=(14, 6))
plt.bar(results_df['config'], results_df['rmse'], color='blue')
plt.title("RMSE Scores for DecisionTreeRegressor")
plt.xlabel('Hyperparameter Configuration')
plt.ylabel('RMSE')
plt.xticks(rotation=90, fontsize=8, ha='right')
plt.tight_layout()
plt.show()

# Bar chart for MAE
plt.figure(figsize=(14, 6))
plt.bar(results_df['config'], results_df['mae'], color='orange')
plt.title("MAE Scores for DecisionTreeRegressor")
plt.xlabel('Hyperparameter Configuration')
plt.ylabel('MAE')
plt.xticks(rotation=90, fontsize=8, ha='right')
plt.tight_layout()
plt.show()

# Bar chart for R^2
```

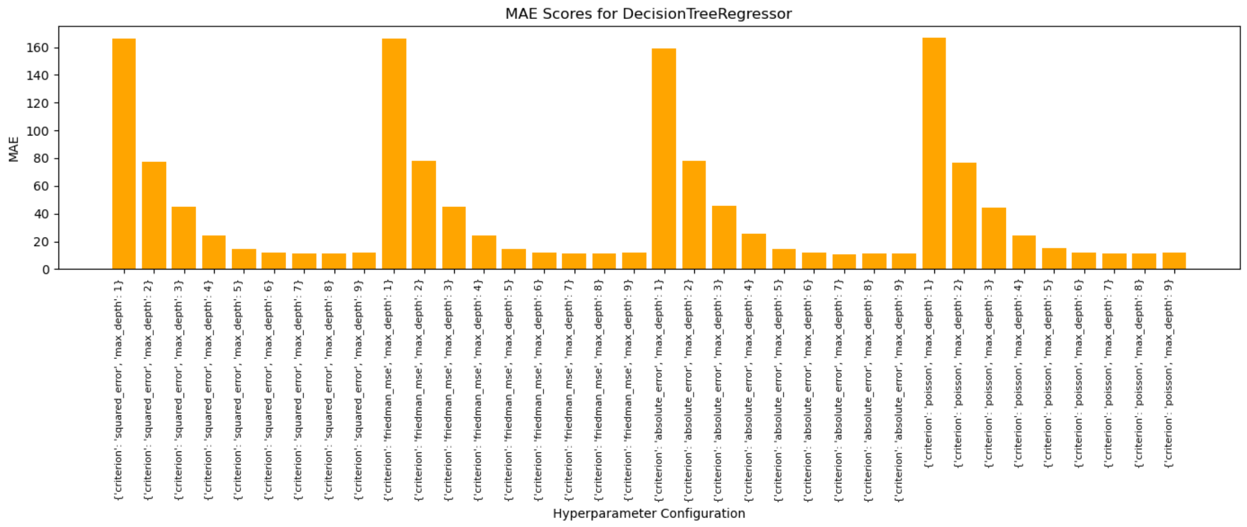
```
plt.figure(figsize=(14, 6))
plt.bar(results_df['config'], results_df['r2'], color='green')
plt.title("R^2 Scores for DecisionTreeRegressor")
plt.xlabel('Hyperparameter Configuration')
plt.ylabel('R^2')
plt.xticks(rotation=90, fontsize=8, ha='right')
plt.tight_layout()
plt.show()
```

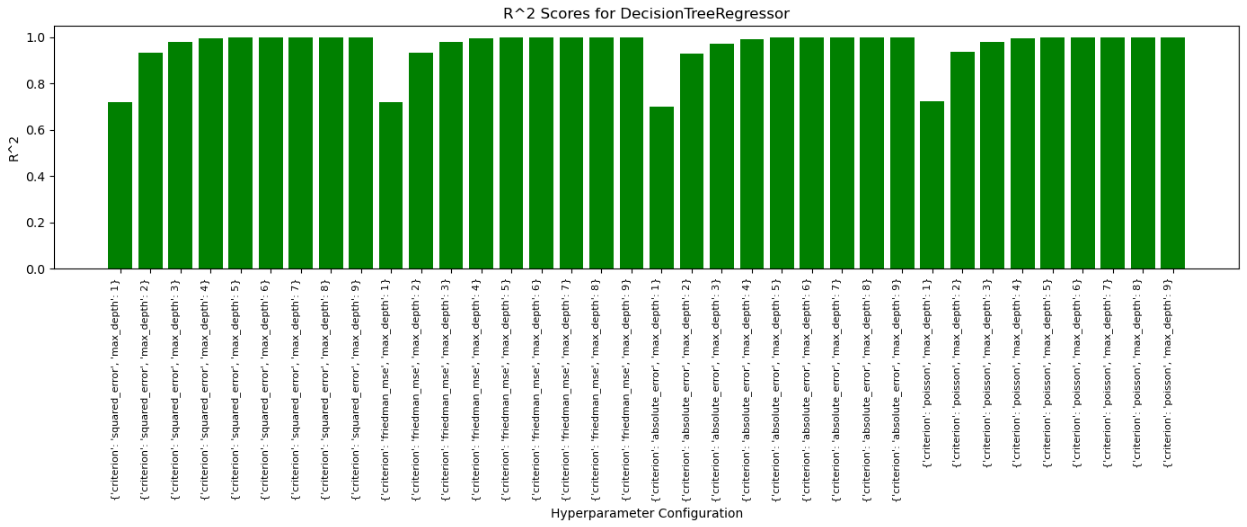
```
Overview of hyperparameter tuning results:
                                                  params         mae
rmse  \
0    {'criterion': 'squared_error', 'max_depth': 1}  166.425693
194.796683
1    {'criterion': 'squared_error', 'max_depth': 2}   77.677594
95.108710
2    {'criterion': 'squared_error', 'max_depth': 3}   44.939361
54.289991
3    {'criterion': 'squared_error', 'max_depth': 4}   24.445115
30.145934
4    {'criterion': 'squared_error', 'max_depth': 5}   14.377385
19.277611
5    {'criterion': 'squared_error', 'max_depth': 6}   11.824829
16.484486
6    {'criterion': 'squared_error', 'max_depth': 7}   11.499010
16.413804
7    {'criterion': 'squared_error', 'max_depth': 8}   11.423525
16.186305
8    {'criterion': 'squared_error', 'max_depth': 9}   11.900218
16.753071
9     {'criterion': 'friedman_mse', 'max_depth': 1}  166.425161
194.795794
10    {'criterion': 'friedman_mse', 'max_depth': 2}   77.766927
95.272507
11    {'criterion': 'friedman_mse', 'max_depth': 3}   45.024419
54.471978
12    {'criterion': 'friedman_mse', 'max_depth': 4}   24.383003
30.048157
13    {'criterion': 'friedman_mse', 'max_depth': 5}   14.517210
19.277771
14    {'criterion': 'friedman_mse', 'max_depth': 6}   11.609694
15.979336
15    {'criterion': 'friedman_mse', 'max_depth': 7}   11.355334
16.143297
16    {'criterion': 'friedman_mse', 'max_depth': 8}   11.528805
16.160391
17    {'criterion': 'friedman_mse', 'max_depth': 9}   12.059430
16.914149
18  {'criterion': 'absolute_error', 'max_depth': 1}  159.401417
201.335237
```

```
19  {'criterion': 'absolute_error', 'max_depth': 2}   78.153118
98.648039
20  {'criterion': 'absolute_error', 'max_depth': 3}   45.653896
60.913013
21  {'criterion': 'absolute_error', 'max_depth': 4}   25.599522
33.446185
22  {'criterion': 'absolute_error', 'max_depth': 5}   14.304209
18.733719
23  {'criterion': 'absolute_error', 'max_depth': 6}   11.597047
15.772630
24  {'criterion': 'absolute_error', 'max_depth': 7}   10.873723
15.290854
25  {'criterion': 'absolute_error', 'max_depth': 8}   11.182712
15.576419
26  {'criterion': 'absolute_error', 'max_depth': 9}   11.381282
15.984143
27         {'criterion': 'poisson', 'max_depth': 1}  166.737640
194.475165
28         {'criterion': 'poisson', 'max_depth': 2}   76.935106
93.266471
29         {'criterion': 'poisson', 'max_depth': 3}   44.415815
53.152011
30         {'criterion': 'poisson', 'max_depth': 4}   24.055160
29.763466
31         {'criterion': 'poisson', 'max_depth': 5}   14.961060
19.794364
32         {'criterion': 'poisson', 'max_depth': 6}   11.936219
16.330048
33         {'criterion': 'poisson', 'max_depth': 7}   11.506436
15.995351
34         {'criterion': 'poisson', 'max_depth': 8}   11.526391
16.190080
35         {'criterion': 'poisson', 'max_depth': 9}   11.787258
16.179034

          r2
0   0.721186
1   0.933452
2   0.978330
3   0.993315
4   0.997268
5   0.997996
6   0.998020
7   0.998071
8   0.997932
9   0.721189
10  0.933219
11  0.978181
12  0.993359
```

```
13  0.997268
14  0.998122
15  0.998084
16  0.998081
17  0.997897
18  0.701925
19  0.928381
20  0.972736
21  0.991780
22  0.997416
23  0.998170
24  0.998277
25  0.998212
26  0.998109
27  0.722117
28  0.935984
29  0.979219
30  0.993484
31  0.997118
32  0.998037
33  0.998116
34  0.998072
35  0.998076
```







a) Defining Hyperparameters and Metrics

We define the grid of hyperparameters to test and specify the evaluation metrics:

```
param_grid = {
    "criterion": ["squared_error", "friedman_mse", "absolute_error",
"poisson"],
    "max_depth": range(1, 10)
}

scoring = {
    'mae': make_scorer(mean_absolute_error, greater_is_better=False),
    'rmse': make_scorer(mean_squared_error, greater_is_better=False,
squared=False),
    'r2': make_scorer(r2_score)
}
```

- **Hyperparameters**:
  - `criterion`: Defines the function to measure the quality of a split.
  - `max_depth`: Specifies the maximum depth of the decision tree.
- **Evaluation Metrics**:
  - **MAE**: Mean Absolute Error.
  - **RMSE**: Root Mean Squared Error.
  - **R²**: Coefficient of Determination.

---

b) Initializing and Performing Grid Search

The `GridSearchCV` is used to systematically explore all combinations of hyperparameters with 3-fold cross-validation.

```
search = GridSearchCV(
    estimator=DecisionTreeRegressor(),
    param_grid=param_grid,
    cv=3,
    scoring=scoring,
    refit='r2'  # Optimizes the model for the R^2 score
)
search.fit(X_train, Y_train)
```

- **Cross-Validation**: Divides the data into 3 folds to evaluate each hyperparameter combination.
- **Refit**: The model is refitted using the configuration with the **highest R² score**.

---

c) Storing Results

The results of the Grid Search are stored and formatted for analysis:

```
results = []
for i in range(len(search.cv_results_['params'])):
    params = search.cv_results_['params'][i]
    mae = search.cv_results_['mean_test_mae'][i]
    rmse = search.cv_results_['mean_test_rmse'][i]
    r2 = search.cv_results_['mean_test_r2'][i]
    results.append({
        'params': params,
        'mae': -mae,  # Reverse the sign for MAE and RMSE
        'rmse': -rmse,
        'r2': r2
    })
results_df = pd.DataFrame(results)
```

- Each combination of hyperparameters is evaluated, and the corresponding **MAE**, **RMSE**, and **R²** values are stored.

---

d) Visualizing Results

Bar charts are generated to compare the performance of all tested configurations:

```
# RMSE Bar Chart
plt.bar(results_df['config'], results_df['rmse'], color='blue')
plt.title("RMSE Scores for DecisionTreeRegressor")
```

- **RMSE Plot**: Illustrates the model's error magnitude.

- **MAE Plot**: Highlights the absolute error across configurations.

- **R² Plot**: Displays how well the model explains variance in the target variable.

IV) Results Interpretation

The hyperparameter tuning results are summarized as follows:

1. **Optimal Configuration**:
   - **Criterion**: `absolute_error`

   - **Max Depth**: 7
2. **Performance Metrics**:
   - **Mean Absolute Error (MAE)**: 10.873723

   - **Root Mean Squared Error (RMSE)**: 15.290854
   - **R²**: 0.998277
3. **Key Observations**:
   - **Deeper Trees Improve Accuracy**: The performance improves as the maximum depth increases, but stabilizes around depth 7 to 8.

   - **`absolute_error` Criterion**: This splitting criterion achieved the lowest errors, demonstrating its robustness against outliers.

   - **Trade-Off Between Depth and Complexity**: Very shallow trees (e.g., max depth = 1–3) perform poorly, while excessively deep trees risk overfitting with marginal gains.
4. **Visual Insights**:

- **RMSE and MAE**: Both metrics exhibit a sharp decline as the tree depth increases, with minimal improvements beyond depth 7.

- **R²**: The model explains over **99% of the variance** at its optimal configuration, confirming its effectiveness.

V) Evaluation of the Optimized Decision Tree Regressor

```
# Best configuration
best_model = DecisionTreeRegressor(criterion='absolute_error',
max_depth=7)

# Train the model on the training set
best_model.fit(X_train, Y_train)

# Make predictions on the test set
y_pred_DecisionTreeRegressor = best_model.predict(X_test)

# Calculate metrics on the test set
DecisionTreeRegressor_mae = mean_absolute_error(Y_test,
y_pred_DecisionTreeRegressor)
DecisionTreeRegressor_rmse = mean_squared_error(Y_test,
y_pred_DecisionTreeRegressor, squared=False)
DecisionTreeRegressor_r2 = r2_score(Y_test,
y_pred_DecisionTreeRegressor)

# Nouvelle formule pour DecisionTreeRegressor_relative_accuracy
DecisionTreeRegressor_relative_accuracy = 100 * (1 -
np.mean(np.abs((Y_test - y_pred_DecisionTreeRegressor) / Y_test)))
```

The optimized Decision Tree Regressor (with `criterion='absolute_error'` and `max_depth=7`) is retrained on the training set and evaluated on the test set.

1. **Training and Prediction**:
   - The model is trained on the training data and used to predict target values for the test set.
2. **Performance Metrics**:
   - **Mean Absolute Error (MAE)**.
   - **Root Mean Squared Error (RMSE)**.
   - **R² Score**.
   - **Relative Accuracy**.

VI) Performance Comparison: Linear Regression vs. Decision Tree Regressor

```
print("\nPerformance Metrics Comparaison between both models:")
```

```python
print("Mean Absolute Error (MAE) Regression model : ", mae_after)
print("Mean Absolute Error (MAE) DecisionTreeRegressor model :",
DecisionTreeRegressor_mae, "\n")

print("Coefficient of Determination (R^2) Regression model : ",
r2_after)
print("Coefficient of Determination (R^2) DecisionTreeRegressor
model :", DecisionTreeRegressor_r2, "\n")

print("RMSE Regression model : ", rmse_after)
print("RMSE DecisionTreeRegressor model:", DecisionTreeRegressor_rmse,
"\n")

print("Relative Accuracy Regression model : ",
relative_accuracy_after, "%")
print("Relative Accuracy DecisionTreeRegressor model :",
DecisionTreeRegressor_relative_accuracy, "%")
```

```
Performance Metrics Comparaison between both models:
Mean Absolute Error (MAE) Regression model :  10.012033908321024
Mean Absolute Error (MAE) DecisionTreeRegressor model :
10.43444256756756

Coefficient of Determination (R^2) Regression model :
0.9988402212150296
Coefficient of Determination (R^2) DecisionTreeRegressor model :
0.9982834009943852

RMSE Regression model :  12.556028260545332
RMSE DecisionTreeRegressor model: 15.275630076689628

Relative Accuracy Regression model :  99.53296397462005 %
Relative Accuracy DecisionTreeRegressor model : 99.51780216018109 %
```

Metric	Linear Regression	Decision Tree Regressor	Observation
Mean Absolute Error (MAE)	10.01	10.43	Linear Regression performs better.
Coefficient of Determination (R^2)	0.9988	0.9983	Linear Regression explains variance slightly better.
Root Mean Squared Error (RMSE)	12.56	15.28	Linear Regression achieves lower error.
Relative Accuracy (%)	99.53	99.52	Both models achieve high accuracy.

Key Observations

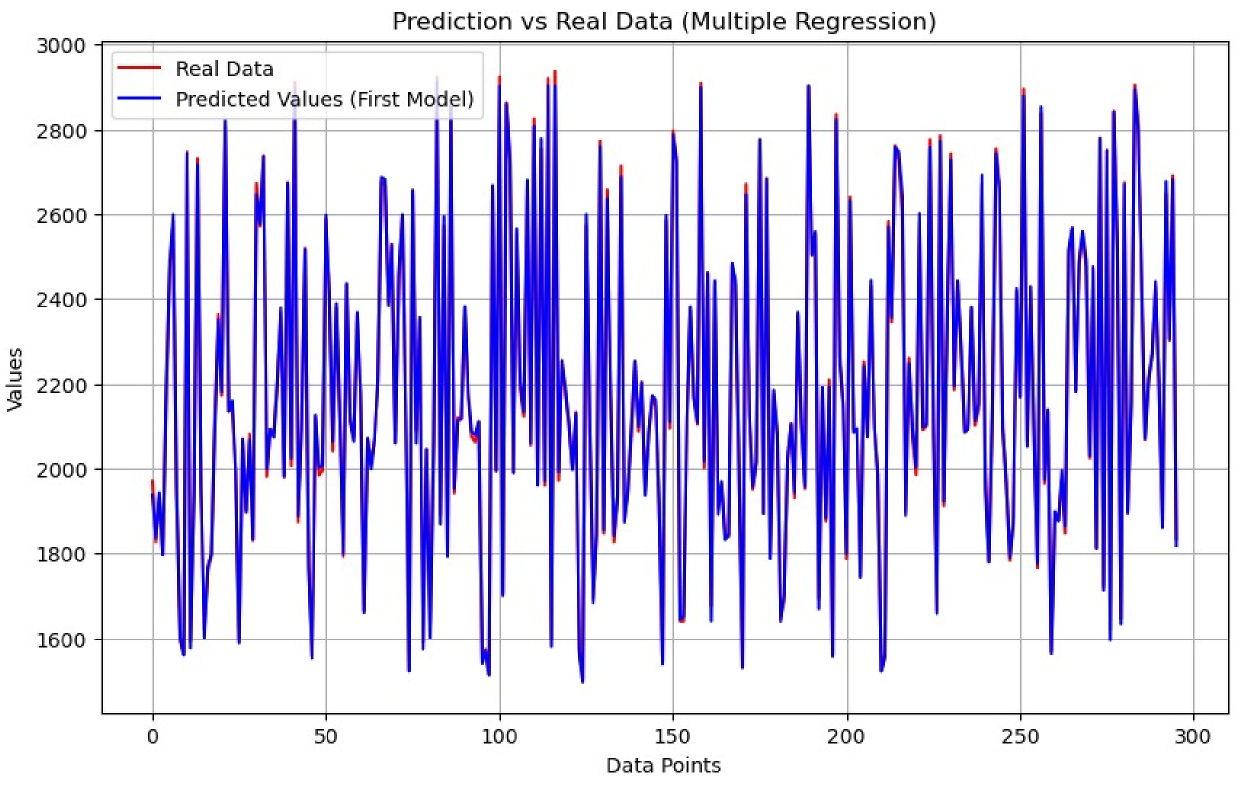
1. **Linear Regression** outperforms the **Decision Tree Regressor** slightly across all metrics, achieving:
   - Lower **MAE** and **RMSE**, indicating smaller prediction errors.

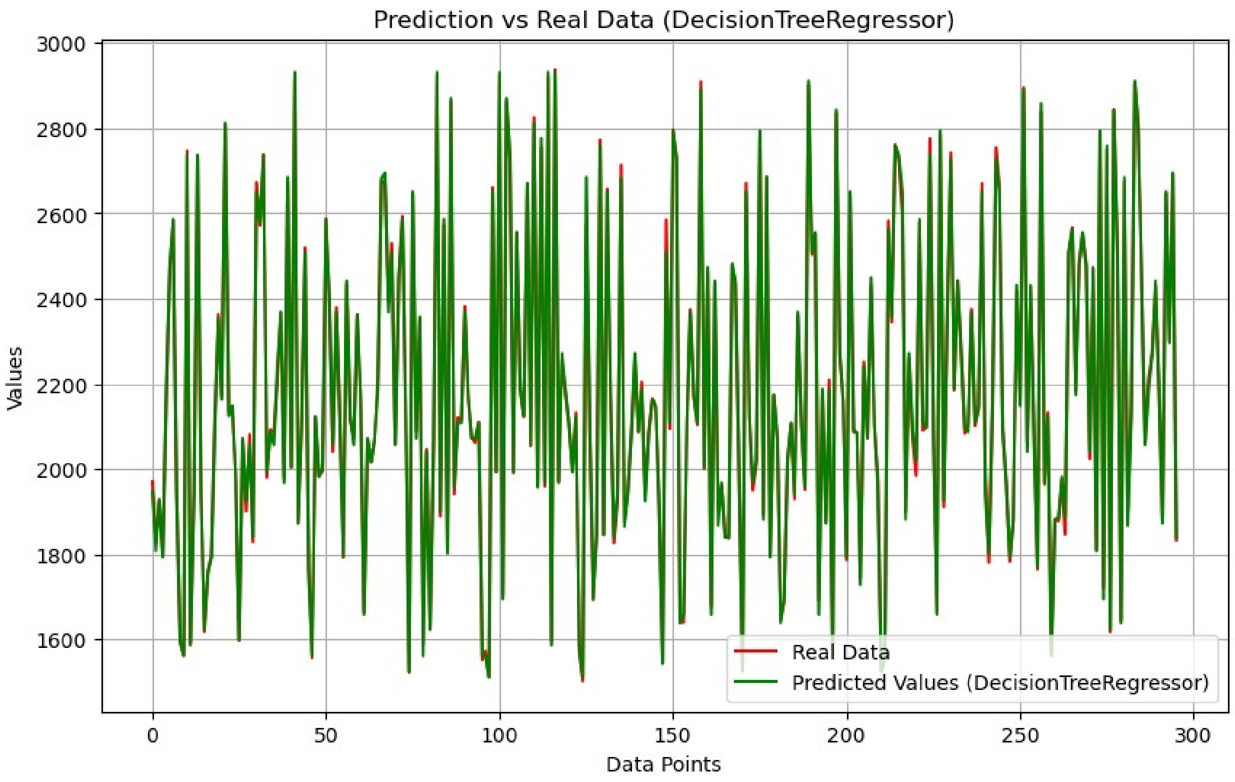
   - A slightly higher **R²** score, meaning it explains more variance in the target variable.
2. The **Decision Tree Regressor**, while capable of handling non-linear relationships, shows marginally higher errors. This could be due to overfitting or the model's sensitivity to hyperparameter settings.
3. **Relative Accuracy** for both models is very high (~99.5%), highlighting their strong predictive capabilities.

While both models perform exceptionally well, our model achieves slightly better performance on this dataset, making it the more suitable choice for this specific task.

VII) Plot the results of both models

```
# Plot 1: Real data vs predicted values for Multiple Regression model
plt.figure(figsize=(10, 6))
plt.plot(range(len(Y_test)), Y_test, color='red', label='Real Data')
plt.plot(range(len(y_pred)), y_pred, color='blue', label='Predicted
Values (First Model)')
plt.title("Prediction vs Real Data (Multiple Regression)")
plt.xlabel("Data Points")
plt.ylabel("Values")
plt.legend()
plt.grid(True)
plt.show()

# Plot 2: Real data vs predicted values for DecisionTreeRegressor
plt.figure(figsize=(10, 6))
plt.plot(range(len(Y_test)), Y_test, color='red', label='Real Data')
plt.plot(range(len(y_pred_DecisionTreeRegressor)),
y_pred_DecisionTreeRegressor, color='green', label='Predicted Values
(DecisionTreeRegressor)')
plt.title("Prediction vs Real Data (DecisionTreeRegressor)")
plt.xlabel("Data Points")
plt.ylabel("Values")
plt.legend()
plt.grid(True)
plt.show()
```


Prediction vs Real Data (Multiple Regression)
Real Data
Predicted Values (First Model)
Values
Data Points
3000
2800
2600
2400
2200
2000
1800
1600
0
50
100
150
200
250
300


Prediction vs Real Data (DecisionTreeRegressor)
Real Data
Predicted Values (DecisionTreeRegressor)
Values
Data Points
3000
2800
2600
2400
2200
2000
1800
1600
0
50
100
150
200
250
300

Plot 1: Multiple Regression Model

- The red line represents the **real data** (true values).
- The blue line represents the **predicted values** from the **Multiple Regression model**.

Observations:

- The predicted values closely align with the real data, showing smooth and consistent trends.
- The Multiple Regression model effectively captures the global patterns and variability in the target variable with minimal deviations.
- The smoother predictions indicate the model's robustness and ability to generalize well to unseen data.

Plot 2: Decision Tree Regressor Model

- The red line represents the **real data** (true values).
- The green line represents the **predicted values** from the **Decision Tree Regressor**.

Observations:

- The predictions generally align with the real data, demonstrating the model's ability to adapt to local variations.
- However, the Decision Tree model produces sharper fluctuations and shows higher sensitivity to individual data points, which may suggest slight overfitting.

Both models demonstrate strong predictive capabilities; however, the **Multiple Regression model** slightly outperforms the **Decision Tree Regressor**:

1. The **Multiple Regression model** produces smoother and more stable predictions, effectively capturing the overall trends and minimizing errors.
2. The **Decision Tree Regressor**, while strong at adapting to local variations, shows more abrupt fluctuations that may reduce its ability to generalize as effectively.

The **Multiple Regression model** is the better-performing model in this case, providing slightly more accurate and stable predictions. Its ability to capture the global structure of the data with fewer fluctuations makes it a more reliable choice for this task. The Decision Tree Regressor remains a strong alternative, particularly for capturing finer, non-linear relationships, but exhibits minor trade-offs in stability.

IV) Conclusion

The final model presented in this report successfully forecasts the next day’s high by incorporating a carefully selected set of predictors, including price variables, moving averages, and economic indicators. Only statistically significant variables were retained, ensuring that each predictor contributes meaningful explanatory power to the model.

The model's performance metrics highlight its strong predictive capabilities and precision. The **Root Mean Squared Error (RMSE)** indicate that the predicted values deviate, on average, by approximately 12.55, respectively, demonstrating a high level of accuracy. Furthermore, the **Coefficient of Determination (R^2)** of 0.9988 confirms that the model explains nearly 99.88% of the variance in the data, leaving only a minimal portion unexplained. The overall **relative accuracy** of 99.53% further underscores the model's reliability and effectiveness in forecasting.

To further enhance the model’s accuracy and robustness, several improvements can be explored. First, comparing the current **Multiple Linear Regression (MLR)** model with more advanced techniques, such as **deep learning**, may help capture intricate non-linear relationships that linear methods might miss. Second, additional experimentation with data preprocessing, including alternative transformations, could uncover hidden patterns and improve predictions. Finally, training the model on periods of greater market volatility, such as times of significant fluctuations in the S&P 500, would increase its adaptability and robustness in handling real-world scenarios, albeit with a minor trade-off in overall accuracy.

In summary, while the model demonstrates impressive predictive performance and precision, future enhancements—such as exploring advanced techniques, refining preprocessing, and accounting for market volatility—could further improve its ability to deliver accurate and resilient forecasts.

V) References

[1]: Stock price prediction using multiple linear regression and support vector machine (regression), ResearchGate.

[2]: BCP Business & Management CMAM 2022, Volume 36 (2023), ResearchGate.

[3]: "Applied Regression Analysis" by Norman R. Draper and Harry Smith.

[4]: Graham, B. "The Intelligent Investor", Harper & Brothers (1949).

[6]: Jiang, Manrui, et al. "The two-stage machine learning ensemble models for stock price prediction", Annals of Operations Research (2020).

[7]: Wen, Yulian, Peiguang Lin, and Xiushan Nie. "Research of stock price prediction based on PCA-LSTM model", IOP Conference Series: Materials Science and Engineering.

[8]: Mehtab, Sidra, Jaydip Sen, and Abhishek Dutta. "Stock price prediction using machine learning and LSTM-based deep learning models", arXiv preprint (2020).

[9]: Hosseinzadeh, A., et al. "Application of artificial neural network and multiple linear regression in modeling nutrient recovery", Bioresource Technology (2020).

[10]: Zhang, G. P. "Time series forecasting using a hybrid ARIMA and neural network model", Neurocomputing (2003).

[12]: Fama, E. F., and French, K. R., "Common Risk Factors in the Returns on Stocks and Bonds", Journal of Financial Economics, 1993.

[13]: Bhushan, T., et al. "Multicollinearity in Regression Analysis", International Journal of Applied Research (2019).

[14] Richard L Burden and J Douglas Faires. Numerical analysis, brooks. Cole, Belmont,CA, 1997.

[15] Graham Upton and Ian Cook. Understanding statistics. Oxford University Press, 1996.