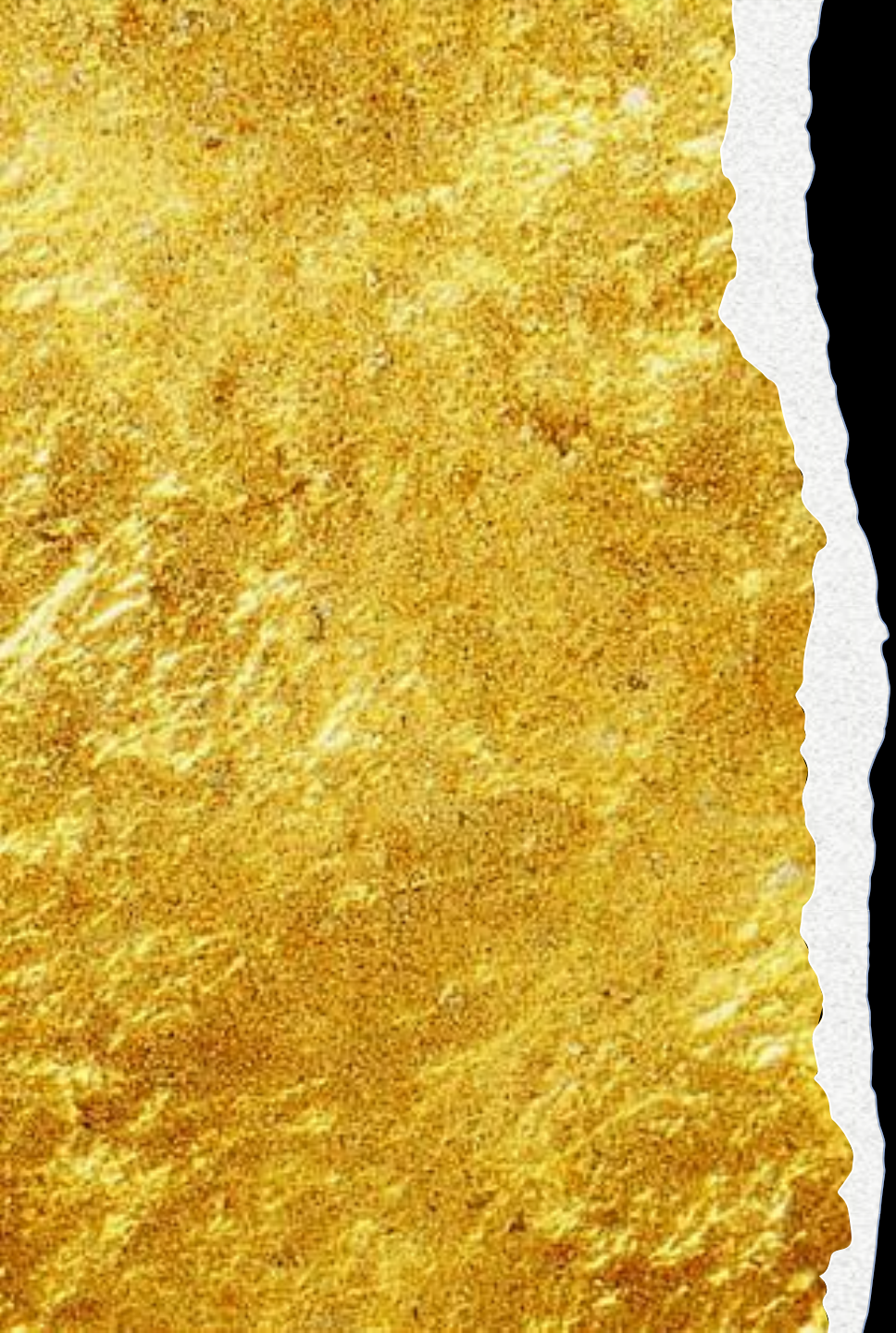


GOLD PRICE PREDICTION

MUQADDAS ALI

TABLE OF CONTENT

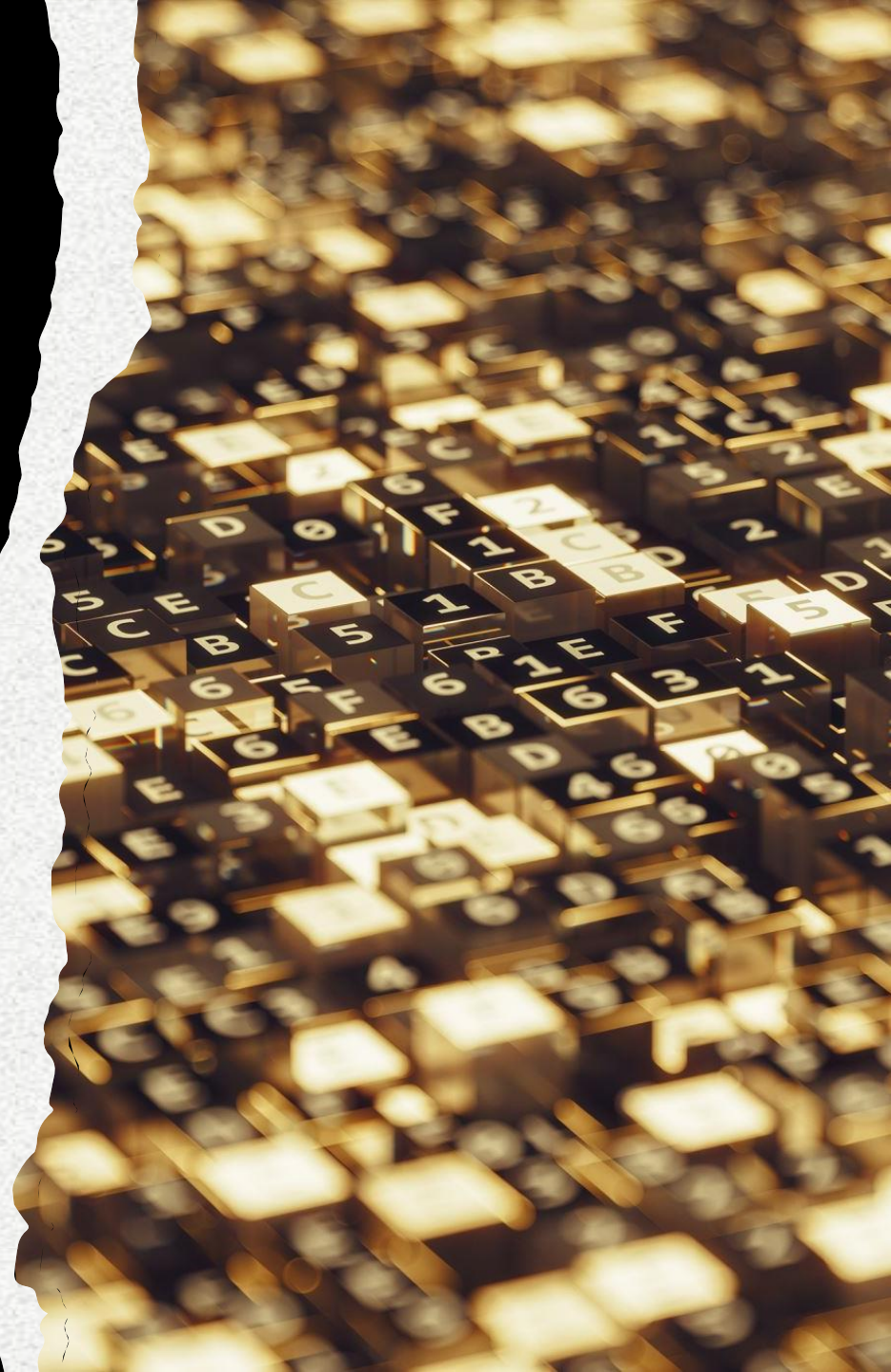
- Overview
- Objectives
- Key Matrics
- Data Preprocessing
- Time Series Analysis
- Predictive Modeling
- Trading Strategy
- Market Sentiment Analysis
- Satistical Analysis

OVERVIEW

This project aims to leverage a comprehensive dataset of daily gold prices spanning from January 19,2014, to January 22, 2024, obtained from Nasdaq. The dataset encompasses key financial metrics for each trading day, including the opening and closing prices, trading volume, as well as the highest and lowest prices recorded during the day.

OBJECTIVES

This project aims to thoroughly analyze the dynamics of the gold market by examining historical price trends, identifying seasonal patterns and long-term trends. It involves developing predictive models for future gold prices, evaluating different forecasting methods, formulating and testing trading strategies based on price and volume data, studying how market events influence gold prices through sentiment analysis, and conducting statistical analyses to understand price movements and their relationships with external factors and macroeconomic indicators. Ultimately, the project aims to provide insights to aid decision-making and predict future trends in the gold market.



KEY METRICS

1. Date: A unique identifier for each trading day.
2. Close: Closing price of gold on the respective date.
3. Volume: Gold trading volume on the corresponding date.
4. Open: Opening price of gold on the respective date.
5. High: The highest recorded price of gold during the trading day.
6. Low: The lowest price recorded for gold in the trading day.

DATA PREPROCESSING

Importing the pandas library and then loads the dataset from the file named 'cleaned_goldstock.csv'.

After loading the dataset, it displays the first few rows of the data using the 'head()' function.

```
# Load dataset
df = pd.read_csv('./cleaned_goldstock.csv')

# Display the first few rows of the dataset
df.head()
```

	Date	Unnamed: 0	Close	Volume	Open	High	Low
0	2024-01-19	0	2029.3	166078.0	2027.4	2041.9	2022.2
1	2024-01-18	1	2021.6	167013.0	2009.1	2025.6	2007.7
2	2024-01-17	2	2006.5	245194.0	2031.7	2036.1	2004.6
3	2024-01-16	3	2030.2	277995.0	2053.4	2062.8	2027.6
4	2024-01-12	4	2051.6	250946.0	2033.2	2067.3	2033.1

DATA PREPROCESSING

- Converting the Date column to Datetime format.
- Checking the data types of each column in the Data Frame to confirm that the 'Date' column is now of type datetime
- Setting the 'Date' column as the index of the Data Frame df using the set_index() method with inplace=True
- Displaying the first few rows of the Data Frame df to confirm that the 'Date' column has been successfully set as the index.

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

# Check the data types to confirm the change
df.dtypes
```

```
Date            datetime64[ns]
Unnamed: 0               int64
Close                  float64
Volume                 float64
Open                   float64
High                   float64
Low                    float64
dtype: object
```

```
# Set 'Date' as index
df.set_index('Date', inplace=True)

# Display the first few rows to confirm the change
df.head()
```

Date	Unnamed: 0	Close	Volume	Open	High	Low
2024-01-19	0	2029.3	166078.0	2027.4	2041.9	2022.2
2024-01-18	1	2021.6	167013.0	2009.1	2025.6	2007.7
2024-01-17	2	2006.5	245194.0	2031.7	2036.1	2004.6
2024-01-16	3	2030.2	277995.0	2053.4	2062.8	2027.6
2024-01-12	4	2051.6	250946.0	2033.2	2067.3	2033.1

DATA PREPROCESSING

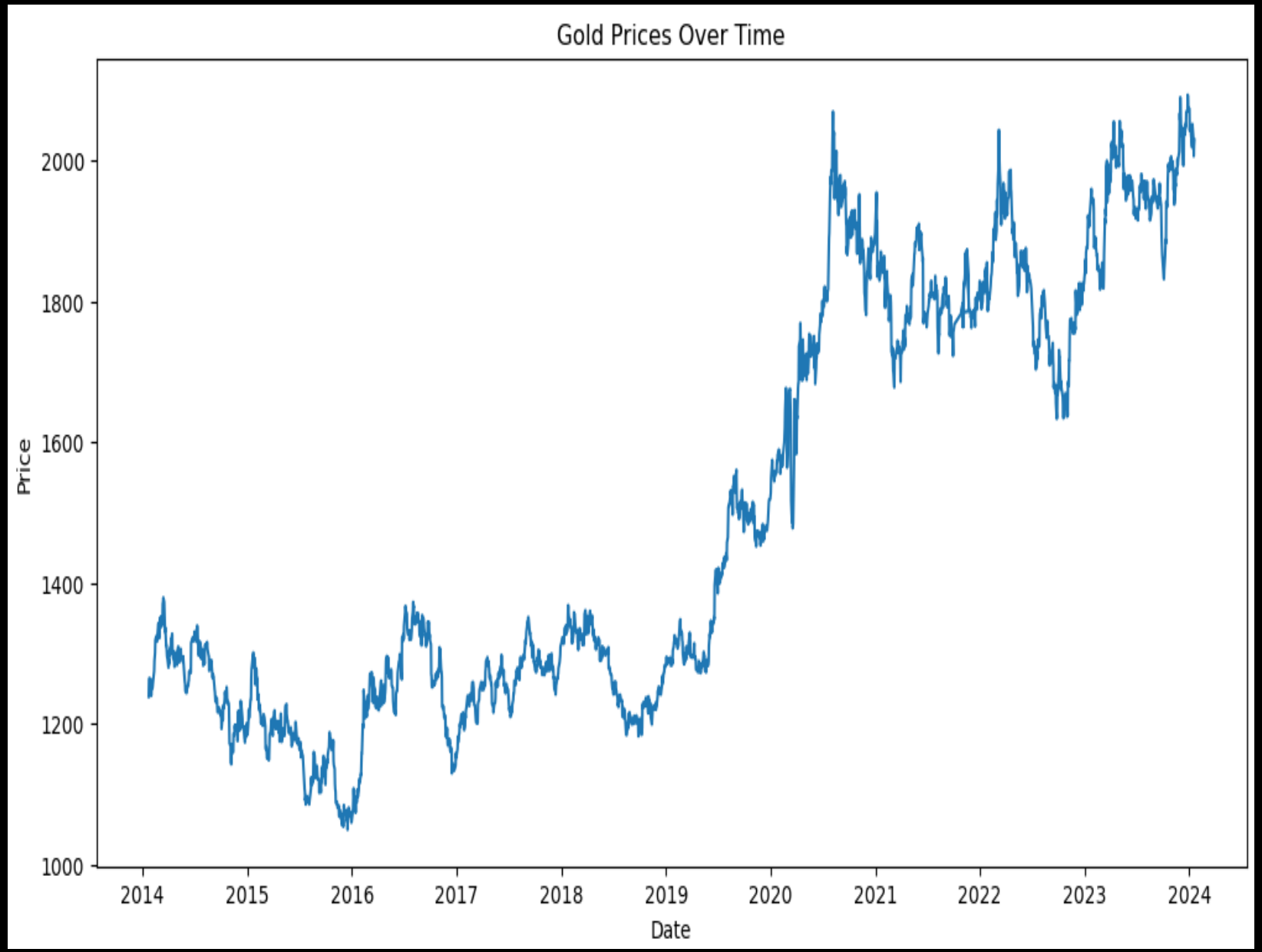
Checking for missing values in each column of the DataFrame df and fills them using the forward fill method to ensure continuity in the data. It then verifies that all missing values have been addressed. Finally, it removes any duplicate rows from the dataset, producing a cleaned version stored in cleaned_goldstock. These steps are crucial for preparing reliable data for subsequent analysis or modeling tasks.

```python
# Check for missing values
missing_values = df.isnull().sum()
print("Missing values in each column:\n", missing_values)

# Fill missing values using forward fill method
df.fillna(method='ffill', inplace=True)

# Confirm there are no missing values
missing_values_after = df.isnull().sum()
print("Missing values after forward fill:\n", missing_values_after)

# Remove duplicates
cleaned_goldstock = df.drop_duplicates()
```

```
Missing values in each column:
 Unnamed: 0    0
Close          0
Volume         0
Open           0
High           0
Low            0
dtype: int64
Missing values after forward fill:
 Unnamed: 0    0
Close          0
Volume         0
Open           0
High           0
Low            0
dtype: int64
```

TIME SERIES ANALYSIS

This plot displaying the historical trend of gold prices based on daily closing prices. The x-axis represents dates, while the y-axis represents the corresponding price values. The plot provides a visual representation of how gold prices have fluctuated over the specified time period.

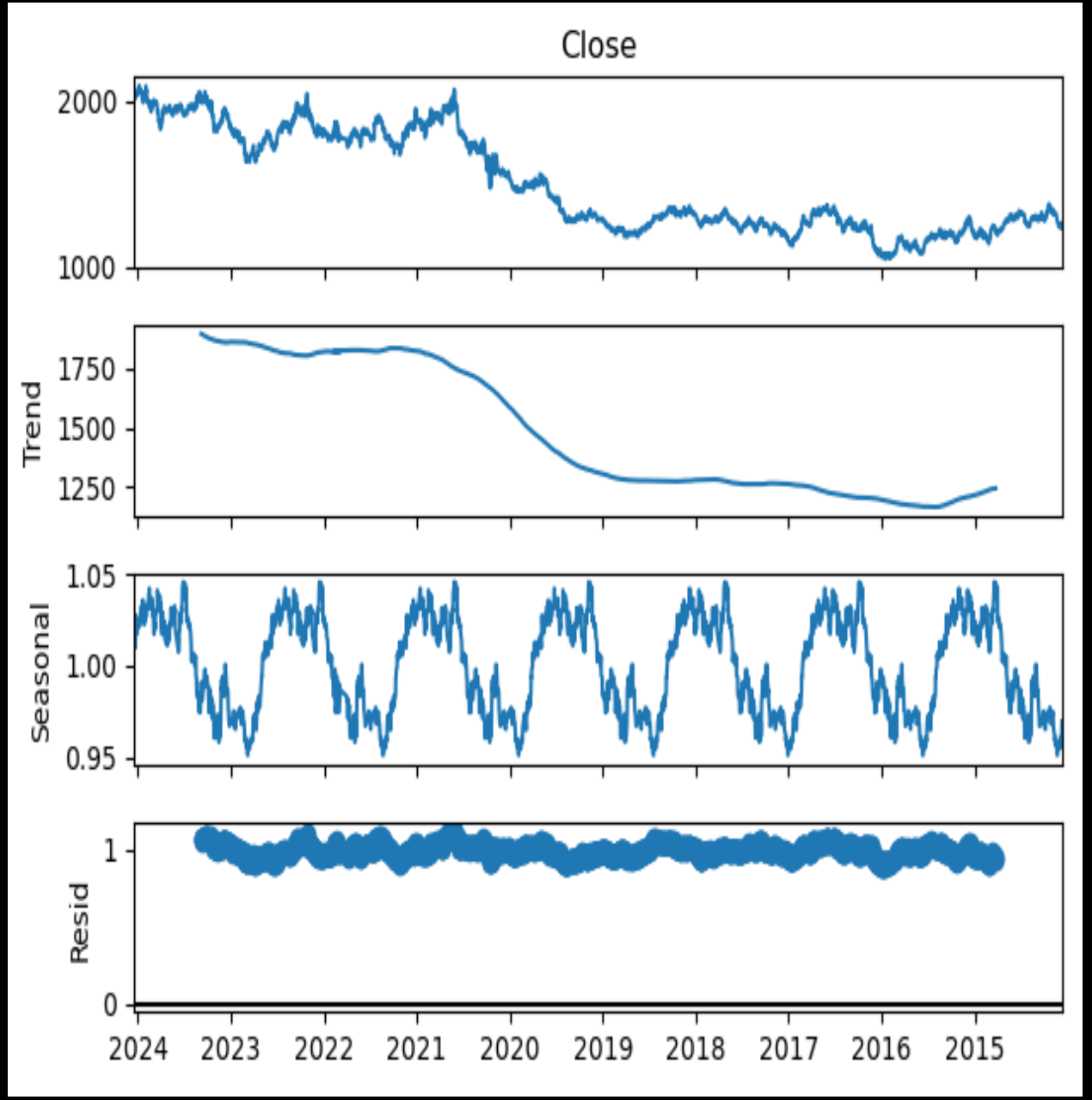


TIME SERIES ANALYSIS

Trend: The trend component represents the long-term progression of the gold prices, showing the overall direction over the entire period.

Seasonality: The seasonal component captures the repeating short-term patterns in the data, such as monthly or yearly cycles that occur at regular intervals.

Residual: The residual component contains the random noise or irregular fluctuations in the data that are not explained by the trend or seasonal components.



PREDICTIVE MODELING

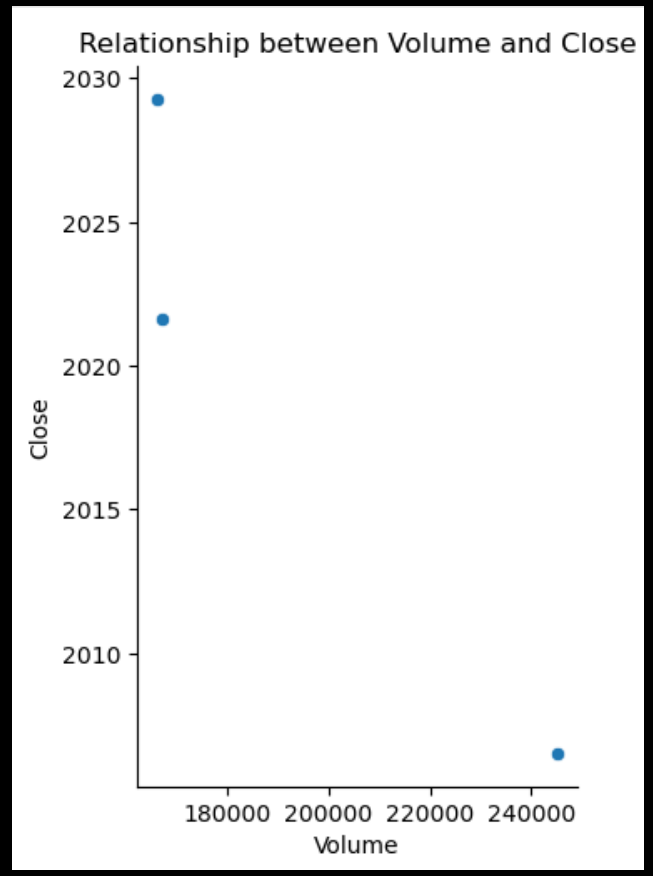
Creating a DataFrame with goldstock data and scales the numerical columns ('Close', 'Volume', 'Open', 'High', 'Low') using StandardScaler. The scaled values replace the original values in a copy of the DataFrame, and the first few rows of this scaled DataFrame are then displayed.

```
# df is DataFrame containing the dataset cleaned_goldstock
df = pd.DataFrame({
    'Date': ['2024-01-19', '2024-01-18', '2024-01-17'],
    'Close': [2029.3, 2021.6, 2006.5],
    'Volume': [166078.0, 167013.0, 245194.0],
    'Open': [2027.4, 2009.1, 2031.7],
    'High': [2041.9, 2025.6, 2036.1],
    'Low': [2022.2, 2007.7, 2004.6]
})

# List of numerical columns to scale
numerical_columns = ['Close', 'Volume', 'Open', 'High', 'Low']

# Initialize StandardScaler
scaler = StandardScaler()

# Scale numerical columns
df_scaled = df.copy()  # Create a copy of the DataFrame cleaned_goldstock
df_scaled[numerical_columns] = scaler.fit_transform(df[numerical_columns])

# Display the scaled DataFrame
print(df_scaled.head())
```

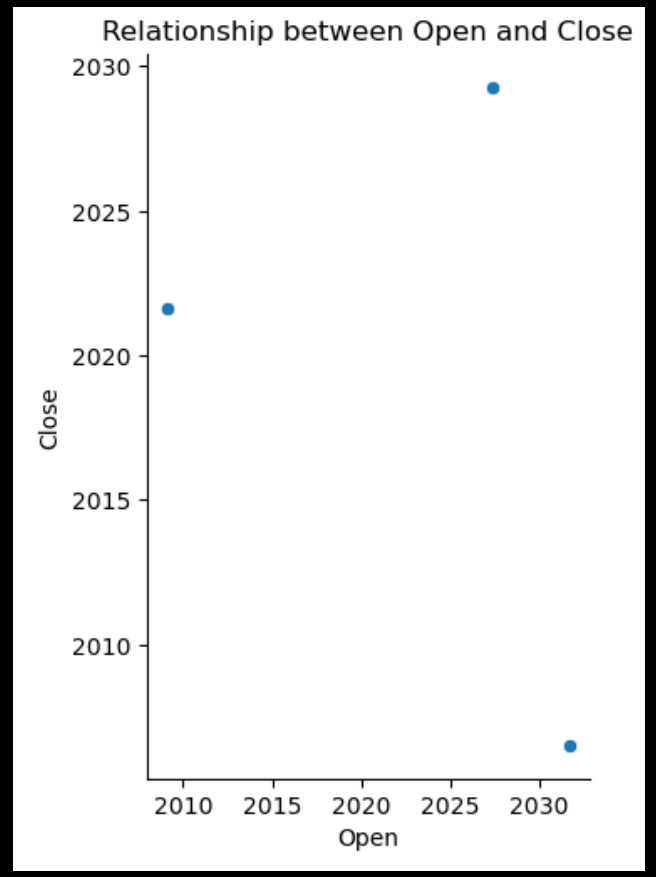
```
         Date     Close    Volume      Open      High       Low
0  2024-01-19  1.073558 -0.719678  0.476251  1.092001  1.394835
1  2024-01-18  0.260470 -0.694460 -1.391334 -1.324236 -0.495362
2  2024-01-17 -1.334027  1.414139  0.915082  0.232235 -0.899473
```

PREDICTIVE MODELING

Generating a scatter plot to visualize the relationship between the Volume and Close prices of gold. The x-axis represents the trading volume, and the y-axis represents the closing prices. This visualization helps in identifying any correlations or patterns between the trading volume and the closing prices.

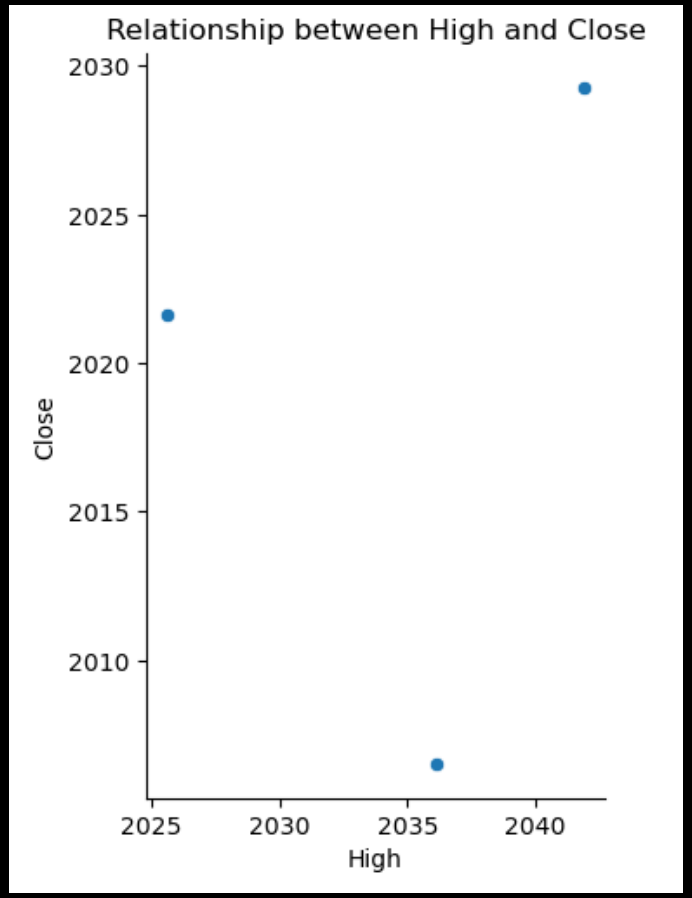


A scatter plot to visualize the relationship between the Open and Close prices of gold. The x-axis represents the opening prices, and the y-axis represents the closing prices. This visualization helps in identifying any correlations or patterns between the opening and closing prices.

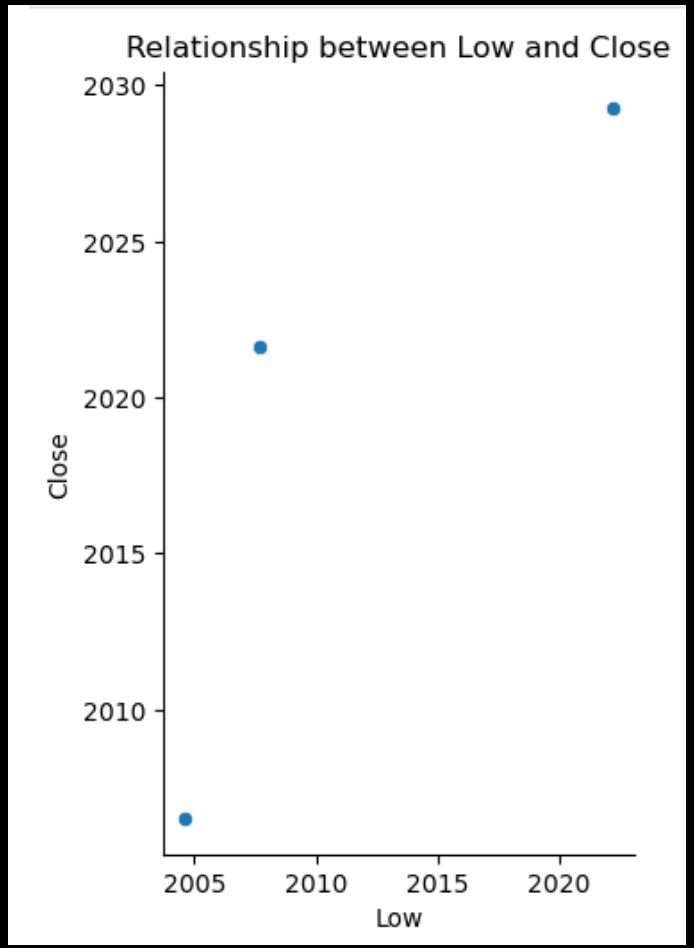


PREDICTIVE MODELING

Scatter plot to visualize the relationship between the High and Close prices of gold. The x-axis represents the highest prices recorded during the trading day, and the y-axis represents the closing prices. This visualization helps in identifying any correlations or patterns between the highest and closing prices.



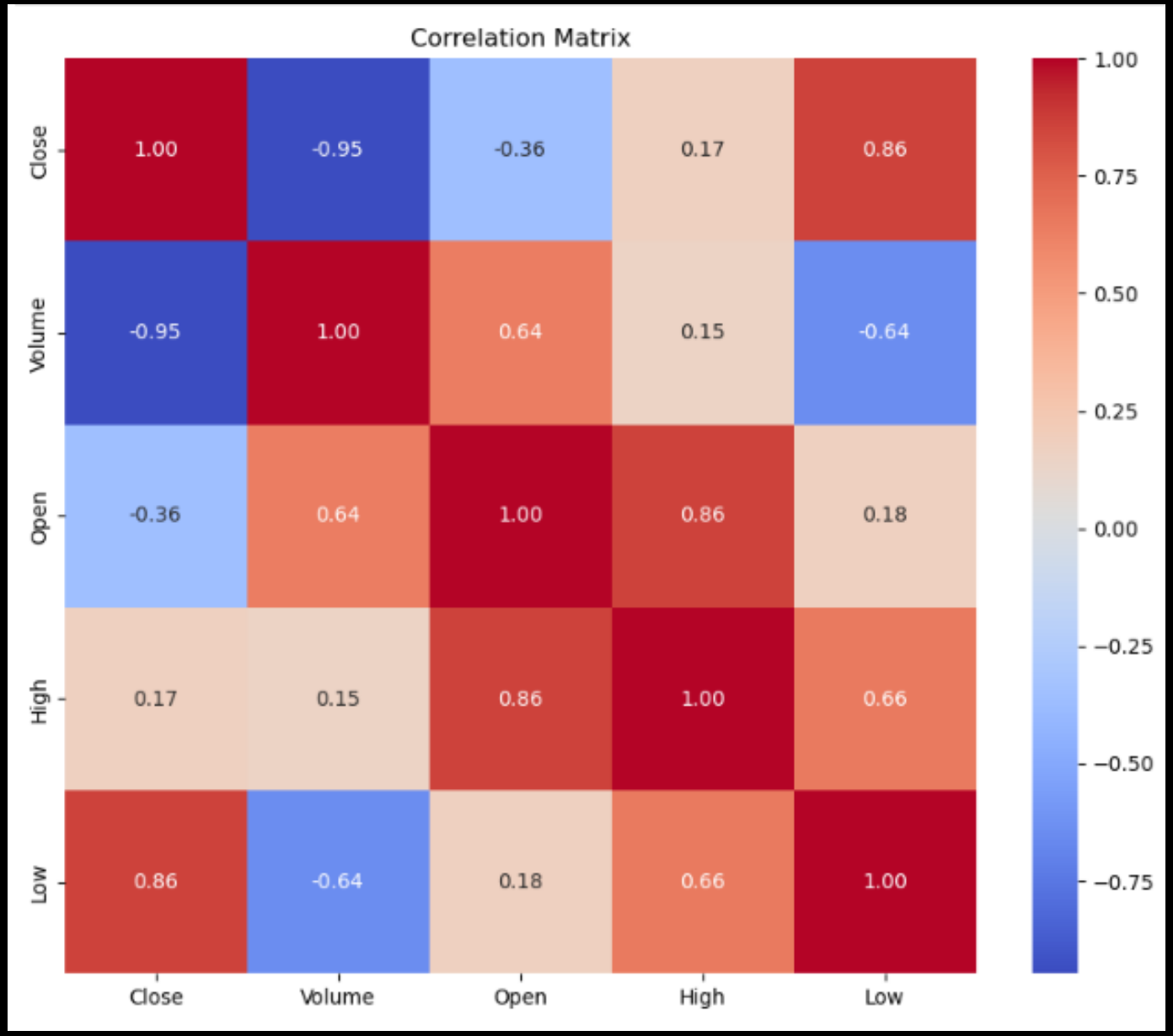
Another scatter plot to visualize the relationship between the Low and Close prices of gold. The x-axis represents the lowest prices recorded during the trading day, and the y-axis represents the closing prices. This visualization helps in identifying any correlations or patterns between the lowest and closing prices.



PREDICTIVE MODELLING

The heatmap shows the correlation matrix of the numerical columns in the DataFrame. Each cell in the heatmap represents the correlation coefficient between two variables, which can range from -1 to 1:

- **Positive Correlation:** Values closer to 1 indicate a strong positive correlation, meaning that as one variable increases, the other variable also tends to increase.
- **Negative Correlation:** Values closer to -1 indicate a strong negative correlation, meaning that as one variable increases, the other variable tends to decrease.
- **No Correlation:** Values around 0 indicate no significant correlation between the variables.



PREDICTIVE MODELING

- Spliting the dataset into training and testing sets, with features ('Volume', 'Open', 'High', 'Low') and target variable ('Close'). It uses 80% of the data for training and 20% for testing. Then, it initializes and trains a Linear Regression model using the training data.
- Predicting the closing prices using the trained Linear Regression model and evaluates its performance. It calculates predictions (y_pred) for the test set (X_test). The model's accuracy is then assessed using Mean Squared Error (MSE), which measures the average squared difference between actual and predicted values, and R-squared (R^2), which indicates how well the model explains the variance in the target variable.

```
# Split dataset into training and testing sets
X = df[['Volume', 'Open', 'High', 'Low']]  # Features
y = df['Close']  # Target variable

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

# Linear Regression model
model = LinearRegression()
model.fit(X_train, y_train)
```

```
▾ LinearRegression
LinearRegression()
```

```
# Predicting and evaluating the model
y_pred = model.predict(X_test)

print('Mean Squared Error:', mean_squared_error(y_test, y_pred))
print('R-squared:', r2_score(y_test, y_pred))
```

```
Mean Squared Error: 56.541586860422015
R-squared: nan
```

TRADING STRATEGY

Defining a trading strategy using the Backtrader library. The MyStrategy class implements a simple moving average (SMA) crossover strategy, where it buys if the closing price is above the 20-day SMA and sells if it's below. The strategy is added to the Backtrader engine (cerebro), along with the historical data feed (df), and is run with an initial cash balance of $1,000,000 for backtesting.

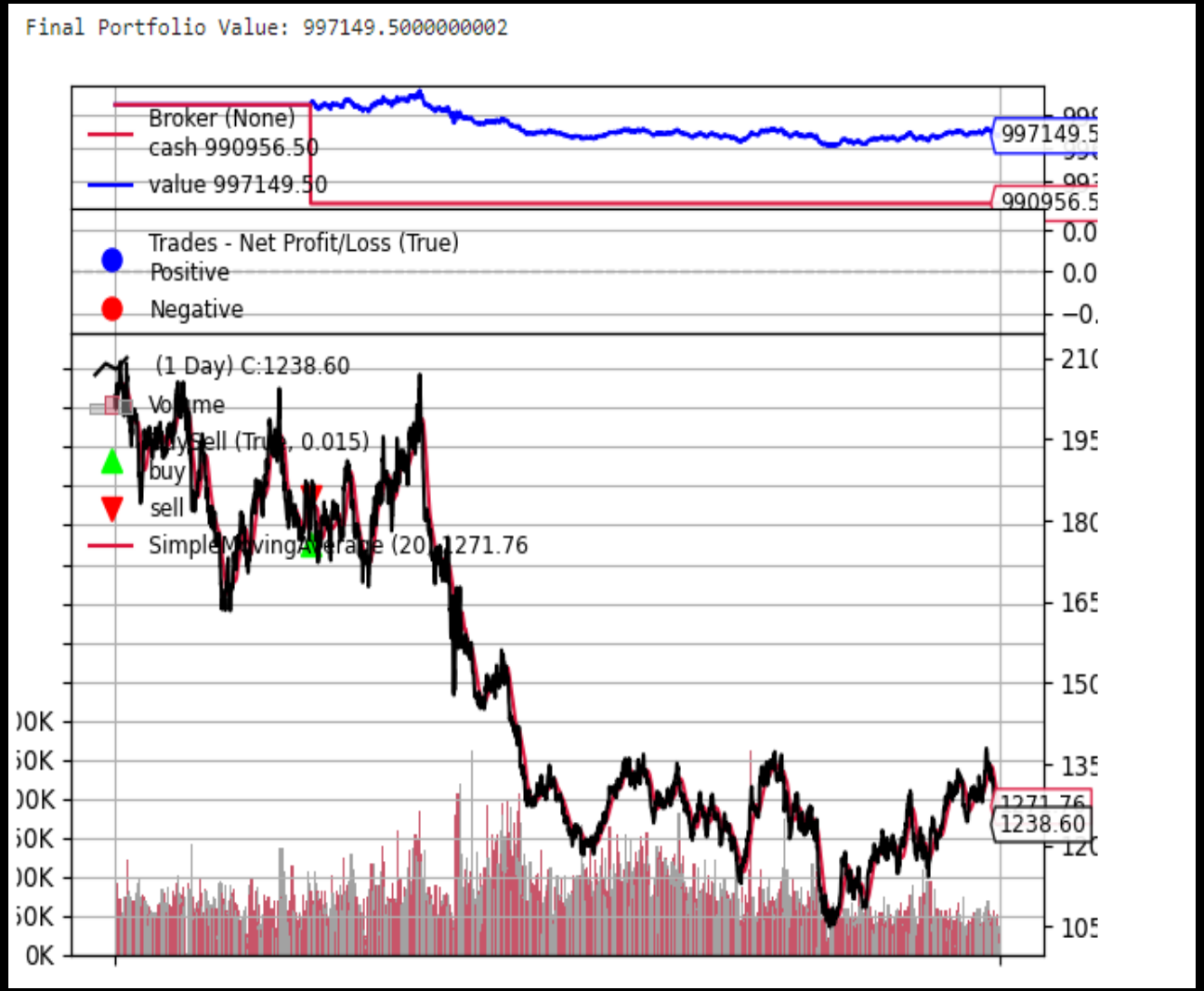
```python
class MyStrategy(bt.Strategy):

    def __init__(self):
        # Initialize indicators or other variables here
        self.sma = bt.indicators.SimpleMovingAverage(self.data.close, period=20)

    def next(self):
        # Implement trading logic here
        if self.data.close[0] > self.sma[0]:  # Buy condition
            self.buy()
        elif self.data.close[0] < self.sma[0]:  # Sell condition
            self.sell()
```

```python
# Initialize Cerebro engine
cerebro = bt.Cerebro()

# Adding data feed
data = bt.feeds.PandasData(dataname=df)
cerebro.adddata(data)

# Adding strategy
cerebro.addstrategy(MyStrategy)

# Set initial cash
cerebro.broker.set_cash(1000000)

# Run the backtest
cerebro.run()
```

```
[<__main__.MyStrategy at 0x28937e97810>]
```

TRADING STRATEGY

The plot shows the performance of the trading strategy over time, including the historical price data of gold, the 20-day Simple Moving Average (SMA), and the buy/sell trades executed based on the SMA crossover. It provides a visual overview of how the strategy's decisions impacted the portfolio value throughout the backtesting period.



TRADING STRATEGY

This plot shows the cumulative returns of two strategies over time: a Buy and Hold strategy and an SMA-based trading strategy. The Buy and Hold line represents the returns from simply holding the asset, while the SMA Strategy line represents the returns generated by trading based on the 20-day and 50-day Simple Moving Average (SMA) crossover signals. This comparison highlights the effectiveness of the SMA strategy relative to the Buy and Hold approach.

MARKET SENTIMENT ANALYSIS

Defining a function to analyze the sentiment of a given text using the 'SentimentIntensityAnalyzer' from the VADER sentiment analysis library. It calculates the sentiment score for a specific event's text, which describes an economic crisis, and prints the sentiment score for that event.

```
def analyze_sentiment(text):
    sia = SentimentIntensityAnalyzer()
    sentiment = sia.polarity_scores(text)['compound']
    return sentiment
```

```
# Analyzing sentiment for a specific event
event_date = '2023-03-15'
event_text = "Economic crisis leads to global uncertainty and volatility in financial markets."

event_sentiment = analyze_sentiment(event_text)
print(f'Sentiment for event on {event_date}: {event_sentiment}')
```

```
Sentiment for event on 2023-03-15: -0.7579
```

MARKET SENTIMENT ANALYSIS

This plot shows the relationship between market sentiment scores and gold prices over time. The blue line with circles represents the sentiment scores, while the red line with crosses represents the closing prices of gold. By overlaying these two data series, the plot allows for a visual comparison of how sentiment changes may correlate with fluctuations in gold prices.

STATISTICAL ANALYSIS

Here are some key statistical insights:

- Trend and Seasonality
- Volatility and Risk Analysis
- Correlation with External Factors
- Macroeconomic Indicators
- Statistical Significance and Hypothesis Testing
- Forecasting and Predictive Modeling
- Market Efficiency and Investor Behavior