# Machine Learning for Retail Trading

Saakshi More
Machine Learning for Finance - Blog Post
Nov 28, 2023

## Abstract

This report delves into the convergence of machine learning and retail trading, unraveling a revolutionary transformation in trading strategies. Emphasizing the shift from an artistic approach to a scientific one in trading, the study showcases the practical implementation of a Random Forest Classifier. Technical details explore the model's incorporation of various technical indicators, such as Relative Strength Index (RSI), Normalized Stock Price, and Moving Averages, to enable informed and strategic retail trading decisions. The findings provide nuanced insights into the model's performance, underlining its potential in reshaping how technology intersects with finance for retail traders.

## Introduction

In the fast-paced world of finance, the fusion of machine learning and retail trading is a game-changer, democratizing access to advanced strategies. Introduced to investing during the pandemic, my early strategy focused on diversified holdings in stocks with steady 5-year growth. Despite reasonable success, I've since explored the scientific side of trading with algorithms. This post examines the transformative power of machine learning for retail traders.

The crux of my trading strategy revolves around determining whether a specific stock on a given date should be bought (1), sold (-1), or held (0). Crafting a dataset for this strategy demanded meticulous attention, particularly in label generation since defining whether each data point should be bought/sold/held was also part of the process. Figure 1 demonstrates the buy/sell signals generated by my strategy for Amazon’s stock price.

It's crucial to clarify that the strategy focuses on recommending the action to take for a stock on a given day, not the volume of shares to transact. This strategy aids in determining whether to buy, sell, or hold a stock at any given time, with the assumption that only one stock is held at a time.

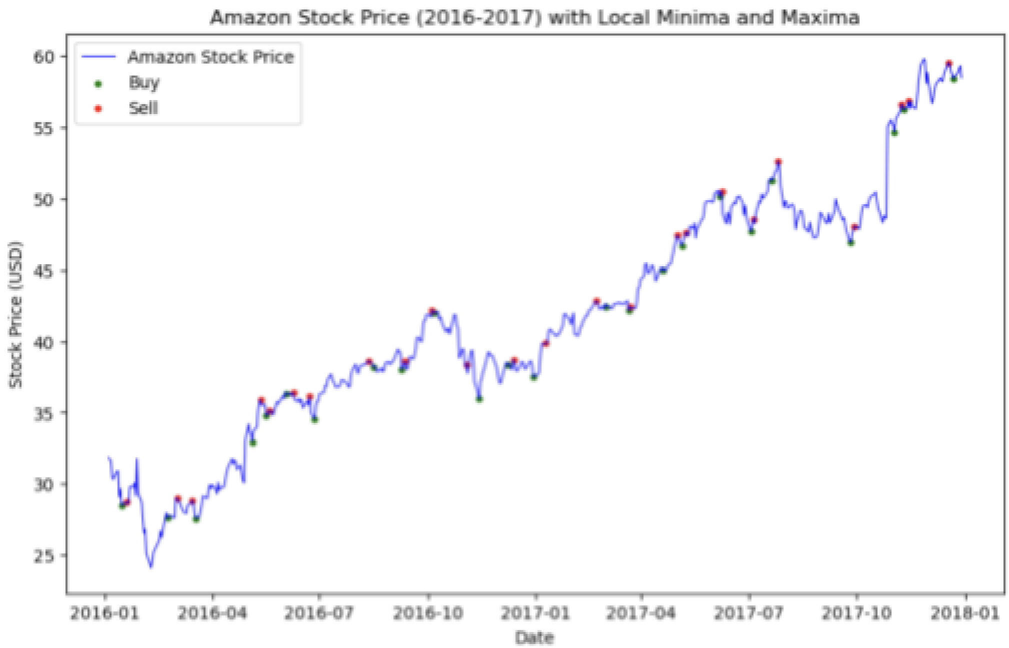



*Figure 1: AMZN stock price action from 2016 to 2017*

## Dataset Generation

Pepsi's historical stock information was used to create the dataset for this strategy but the process described can be replicated for any stock. The training and validation data spanned between 2010 and 2020 (including). The testing data was from 2021. At its core, the label generation strategy was designed to identify opportune moments to buy at local minima and sell at local maxima within 30-day periods, as illustrated in figures 1 and 2.

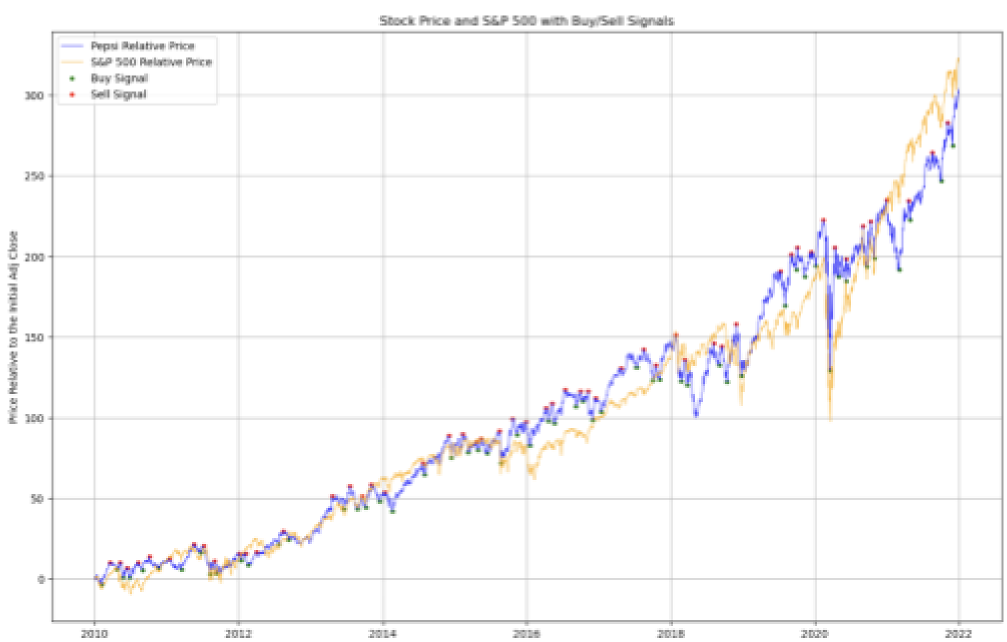



*Figure 2. Pepsi Relative Stock Price with Buy/Sell Signals and S&P 500*

The code snippet is given below:

```python
def generate_labels(data, window_size=15):
    temp = 0
    # Create an empty list to store the labels
    labels = []
    # Find minima and maxima indices
    minima_idx = argrelextrema(data['relative_price'].values, np.less, order=window_size)[0]
    maxima_idx = argrelextrema(data['relative_price'].values, np.greater, order=window_size)[0]

    # Iterate through rows to update labels
    for i in range(len(data)):
        # Buy condition: If current index is a minima and temp is 0
        if i in minima_idx and temp == 0:
            labels.append(1)  # Set to 1 for buy
            temp = 1  # Update temp to 1
        # Sell condition: If current index is a maxima and temp is 1
        elif i in maxima_idx and temp == 1:
            labels.append(-1)  # Set to -1 for sell
            temp = 0  # Update temp to 0
        else:
            labels.append(0)  # No action
    # Create 'Label' column based on buy/sell conditions
    data['label'] = labels
    return data
```

## ML Model

The input data of the ML model included the yahoo finance data along with trend, momentum, and volatility-focused technical indicators as features to the model. The output was a label: -1, 0, 1 - a multi-classification problem.

**Input Parameters:**

1. Trend Indicators:
    a. Closing Price: The closing price of the stock at the end of a trading day.
    b. Normalized Stock Price: The stock price normalized to a specific scale or baseline
    $$\text{Normalized\_Price} = \frac{\text{Close} - \text{Low}}{\text{High} - \text{Low}}$$
    c. 5-day Moving Average: The average of the closing prices over the last 5 trading days, smoothing out short-term fluctuations.
    d. 25-day Moving Average: same as (c) but over the last 25 trading days, providing a longer-term trend perspective
    e. 60-day Moving Average: same as (d) but over the last 60 trading days
    f. Moving Average Convergence Divergence (MACD): calculated by subtracting the 26-day Exponential Moving Average (EMA) from the 12-day EMA.

2. Momentum Indicators:

a. Percentage Change compared to S&P 500 Percentage Change

$$\text{Percentage Change of Pepsi's Closing Price to that of S\&P 500} = \frac{\text{Percentage change in Pepsi's Closing Price}}{\text{Percentage change in S\&P 500}}$$

b. Relative Strength Index (RSI): A momentum oscillator that measures the speed and change of price movements, indicating overbought or oversold conditions.

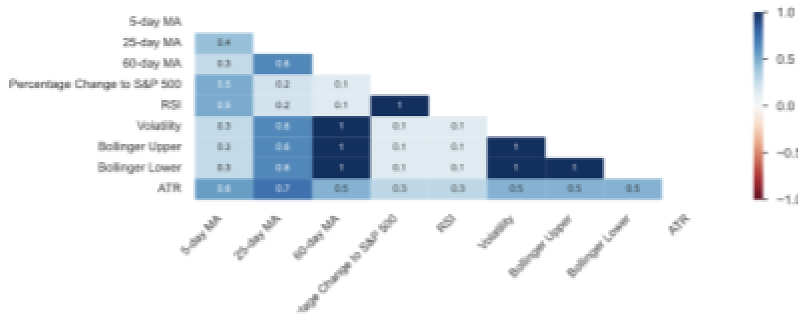
3. Volatility Indicators:
   a. Volatility: Standard Deviation from the 60-day Average Stock Price
   b. Bollinger Bands: Bands plotted around the stock price, representing volatility and consisting of a middle band being an N-period simple moving average and upper and lower bands being N-period standard deviations above and below the moving average.
   c. Average True Range (ATR): A measure of market volatility, representing the average range between the high and low prices over a specified period.
   d. MACD Histogram: The visual representation of the difference between the MACD line and the Signal line, offering insights into the strength of a trend.

4. General Indicators:
   a. Volume: The total number of shares traded during a specific period, providing insights into market activity.

**Exploratory Data Analysis:**

We started with the EDA and made the following observations:

- The absence of 60-day MA is highly correlated with absence of volatility-related indicators, namely, volatility and bollinger bands. To further investigate this, we did a scatter plot matrix and found that these indicators are in fact strongly correlated with each other (see figure 3).
- Missing values are a result of the shift in calculation; the 60-day MA cannot be calculated for the first 60 data points. Thus, all missing values were accounted for.
- The MACD indicator values were all unique.
- The distribution of the labels was heavily imbalanced (see figure 4)
- Many trending indicators were also correlated with each other (see figure 5)




The correlation heatmap measures nullity correlation: how strongly the presence or absence of one variable affects the presence of another.

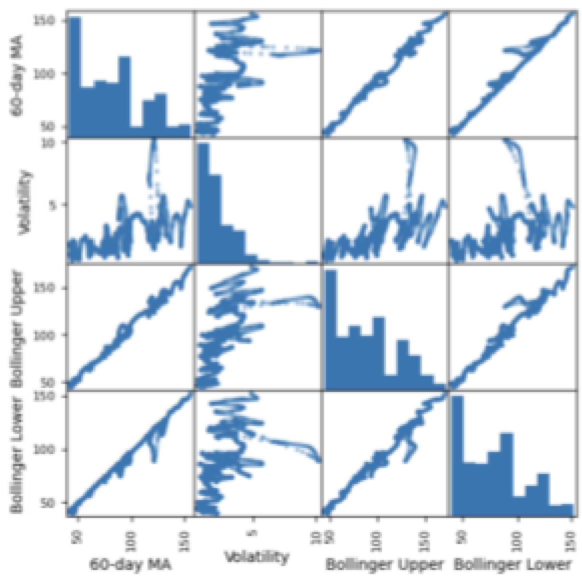



*Figure 3. Missing Values Heatmap (top)*
*Scatter Plot Matrix (bottom)*

label

Categorical

IMBALANCE

| | |
|---|---|
| **Distinct** | 3 |
| **Distinct (%)** | 0.1% |
| **Missing** | 0 |
| **Missing (%)** | 0.0% |
| **Memory size** | 111.7 KiB |


*Figure 4. EDA Report reflects that there is an Imbalanced Classification*

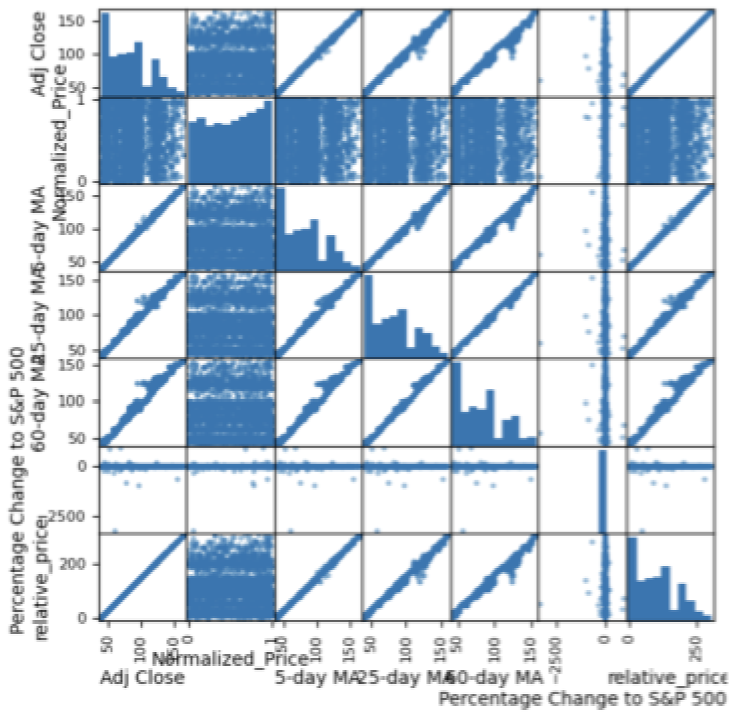



*Figure 5. Scatter Plot Matrix for Trending indicators*

**Data Preprocessing and Feature Selection:**

All null values in the data set were replaced with 0's to avoid complications.

A few iterations of the model made me realize that the underrepresentation of classes 1 and -1 was negatively affecting the performance of the model since the model was hardly ever predicting these classes correctly. To deal with this skewed distribution, two decisions were made:

- Undersampled the data: Only 20% of the data points corresponding to the hold signal or class '0' were retained in the train + test datasets
- Use the Random Forest Classifier as the ML model since it efficiently deals with imbalanced classification because of its complexity

As mentioned earlier, the data from 2010-2020 formed the training dataset and 2021 was the testing dataset.

For feature selection, the feature importance method of Random Forest Regressor was used to identify the top 7 features which would be used by the machine out of the 25 features that were contained in the original dataset. The regressor identified the following features as the top ones:

- Relative Strength Index (RSI)
- Normalized_Price:
- Percentage Change to S&P 500:
- Volume of shares bought/sold
- MACD Histogram
- Volatility: Calculated as the standard deviation in the Adjusted Closing price within a rolling window of 60 days
- Day of the stock

**Optimizing Metric**

Many ML models optimize for maximum accuracy. However, for classification problems, accuracy is not a good representation of the model problem since it does not account for the performance of underrepresented classes. Since our problem statement did not particularly favor minizming false positives (precision) and negatives (recall), the optimizing metric was chosen to be F1-score (combines precision and recall).

Since our problem statement is a multi-classification problem, the F1-score couldn't directly be calculated since these metrics are better suited for binary classification problems. The metrics that we could work with were:

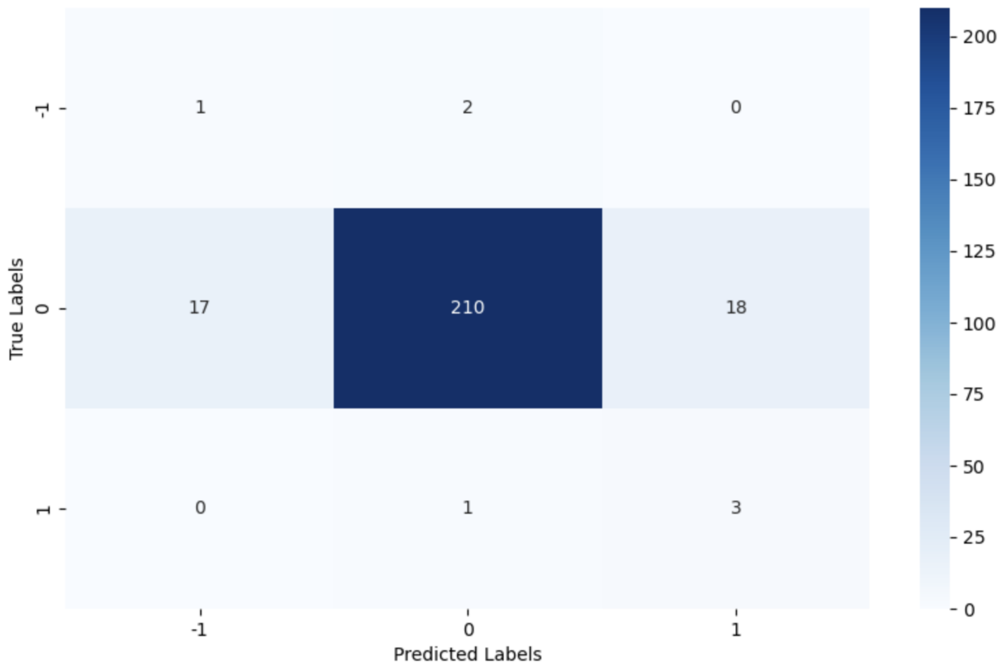
- Micro-averaged F1 score: normal F1 formula but calculated using the total number of True Positives (TP), False Positives (FP) and False Negatives (FN), instead of individually for each class
- Macro-averaged F1 score: unweighted mean of the F1 scores calculated per class

(source: https://stephenallwright.com/micro-vs-macro-f1-score/)

In the first few iterations, the model hardly ever predicted the classes 1 and -1 correctly. However, the micro-averaged F1 score did not reflect this (>0.8), which could easily illusion one into thinking that the model performed well. The macro-averaged score was less than 0.5, which correctly reflected the performance of the model. Thus, macro-averaged F1 score was chosen as the metric that the model had to optimize for.

I used this kaggle notebook that defined the evaluation metrics for a multiclassification problem to define the macro-averaged F1 score of our problem as well.

Grid search was used to determine the best hyperparameters of the random forest classifier and a macro-averaged f1 score of 0.42 was achieved with the following confusion matrix:




While this score is far from ideal, when we plotted the model's recommendations against the closing price of Pepsi, the result was fairly satisfying as most buy signals were at local minima and sell signals at local maxima (see figure 6).

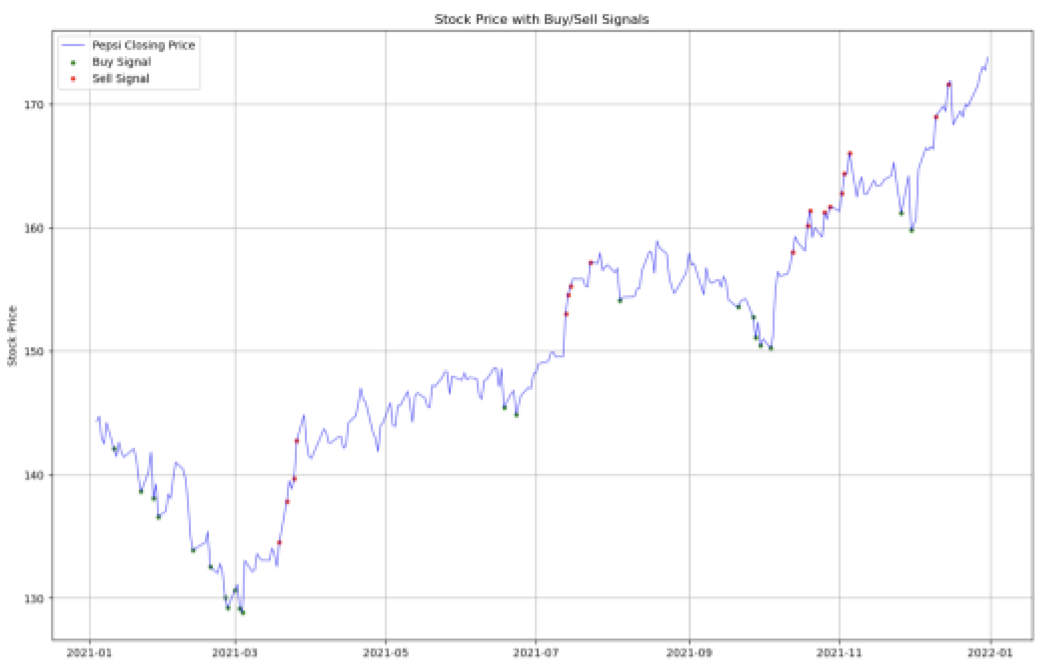



*Figure 6. Closing Price of PEP with model-predicted Buy/Sell signals*

At this point, it is important to note that the 'ground truth' labels were also generated by the code and doesn't necessarily represent the ideal scenario. Hence, the visual representation of the model's output is a better estimate of the model performance than the evaluation metric.

## Conclusion

In conclusion, this trading strategy provides a practical approach for retail traders unfamiliar with intricate hedge-fund methodologies, offering a reliable means to determine optimal stock purchase timings. Improvements to this model could entail:

- delving into diverse models (neural networks) and conducting comparative analyses to gauge their respective performances
- adding constraints, like generating a sell signal only if a stock has been bought previously
- integrating volume considerations for optimal stock allocation

To elevate this strategy, extending its evaluation to multiple stocks and incorporating additional contextual factors, such as industry dynamics, geographical influences, and sentiment analysis, would enrich its predictive capabilities. By expanding the scope and embracing advanced techniques, this strategy can evolve into a comprehensive tool for retail traders seeking a nuanced and informed approach to navigate the dynamic landscape of financial markets.

finml-blog-post

November 29, 2023

# 1 Machine Learning for Retail Trading

[2]:
```
!pip install yfinance matplotlib
```

```
Requirement already satisfied: yfinance in
/Users/saakshimore/miniconda3/lib/python3.10/site-packages (0.2.32)
Requirement already satisfied: matplotlib in
/Users/saakshimore/miniconda3/lib/python3.10/site-packages (3.7.1)
Requirement already satisfied: html5lib>=1.1 in
/Users/saakshimore/miniconda3/lib/python3.10/site-packages (from yfinance) (1.1)
Requirement already satisfied: numpy>=1.16.5 in
/Users/saakshimore/miniconda3/lib/python3.10/site-packages (from yfinance)
(1.24.3)
Requirement already satisfied: requests>=2.31 in
/Users/saakshimore/miniconda3/lib/python3.10/site-packages (from yfinance)
(2.31.0)
Requirement already satisfied: lxml>=4.9.1 in
/Users/saakshimore/miniconda3/lib/python3.10/site-packages (from yfinance)
(4.9.3)
Requirement already satisfied: frozendict>=2.3.4 in
/Users/saakshimore/miniconda3/lib/python3.10/site-packages (from yfinance)
(2.3.9)
Requirement already satisfied: multitasking>=0.0.7 in
/Users/saakshimore/miniconda3/lib/python3.10/site-packages (from yfinance)
(0.0.11)
Requirement already satisfied: appdirs>=1.4.4 in
/Users/saakshimore/miniconda3/lib/python3.10/site-packages (from yfinance)
(1.4.4)
Requirement already satisfied: pytz>=2022.5 in
/Users/saakshimore/miniconda3/lib/python3.10/site-packages (from yfinance)
(2023.3.post1)
Requirement already satisfied: pandas>=1.3.0 in
/Users/saakshimore/miniconda3/lib/python3.10/site-packages (from yfinance)
(2.0.3)
Requirement already satisfied: peewee>=3.16.2 in
/Users/saakshimore/miniconda3/lib/python3.10/site-packages (from yfinance)
(3.17.0)
Requirement already satisfied: beautifulsoup4>=4.11.1 in
```

```
/Users/saakshimore/miniconda3/lib/python3.10/site-packages (from yfinance)
(4.12.0)
Requirement already satisfied: cycler>=0.10 in
/Users/saakshimore/miniconda3/lib/python3.10/site-packages (from matplotlib)
(0.11.0)
Requirement already satisfied: pyparsing>=2.3.1 in
/Users/saakshimore/miniconda3/lib/python3.10/site-packages (from matplotlib)
(3.0.9)
Requirement already satisfied: contourpy>=1.0.1 in
/Users/saakshimore/miniconda3/lib/python3.10/site-packages (from matplotlib)
(1.0.7)
Requirement already satisfied: python-dateutil>=2.7 in
/Users/saakshimore/miniconda3/lib/python3.10/site-packages (from matplotlib)
(2.8.2)
Requirement already satisfied: packaging>=20.0 in
/Users/saakshimore/miniconda3/lib/python3.10/site-packages (from matplotlib)
(23.1)
Requirement already satisfied: pillow>=6.2.0 in
/Users/saakshimore/miniconda3/lib/python3.10/site-packages (from matplotlib)
(9.4.0)
Requirement already satisfied: kiwisolver>=1.0.1 in
/Users/saakshimore/miniconda3/lib/python3.10/site-packages (from matplotlib)
(1.4.4)
Requirement already satisfied: fonttools>=4.22.0 in
/Users/saakshimore/miniconda3/lib/python3.10/site-packages (from matplotlib)
(4.39.2)
Requirement already satisfied: soupsieve>1.2 in
/Users/saakshimore/miniconda3/lib/python3.10/site-packages (from
beautifulsoup4>=4.11.1->yfinance) (2.4)
Requirement already satisfied: webencodings in
/Users/saakshimore/miniconda3/lib/python3.10/site-packages (from
html5lib>=1.1->yfinance) (0.5.1)
Requirement already satisfied: six>=1.9 in
/Users/saakshimore/miniconda3/lib/python3.10/site-packages (from
html5lib>=1.1->yfinance) (1.16.0)
Requirement already satisfied: tzdata>=2022.1 in
/Users/saakshimore/miniconda3/lib/python3.10/site-packages (from
pandas>=1.3.0->yfinance) (2023.3)
Requirement already satisfied: urllib3<3,>=1.21.1 in
/Users/saakshimore/miniconda3/lib/python3.10/site-packages (from
requests>=2.31->yfinance) (1.26.14)
Requirement already satisfied: idna<4,>=2.5 in
/Users/saakshimore/miniconda3/lib/python3.10/site-packages (from
requests>=2.31->yfinance) (3.4)
Requirement already satisfied: charset-normalizer<4,>=2 in
/Users/saakshimore/miniconda3/lib/python3.10/site-packages (from
requests>=2.31->yfinance) (2.0.4)
Requirement already satisfied: certifi>=2017.4.17 in
```

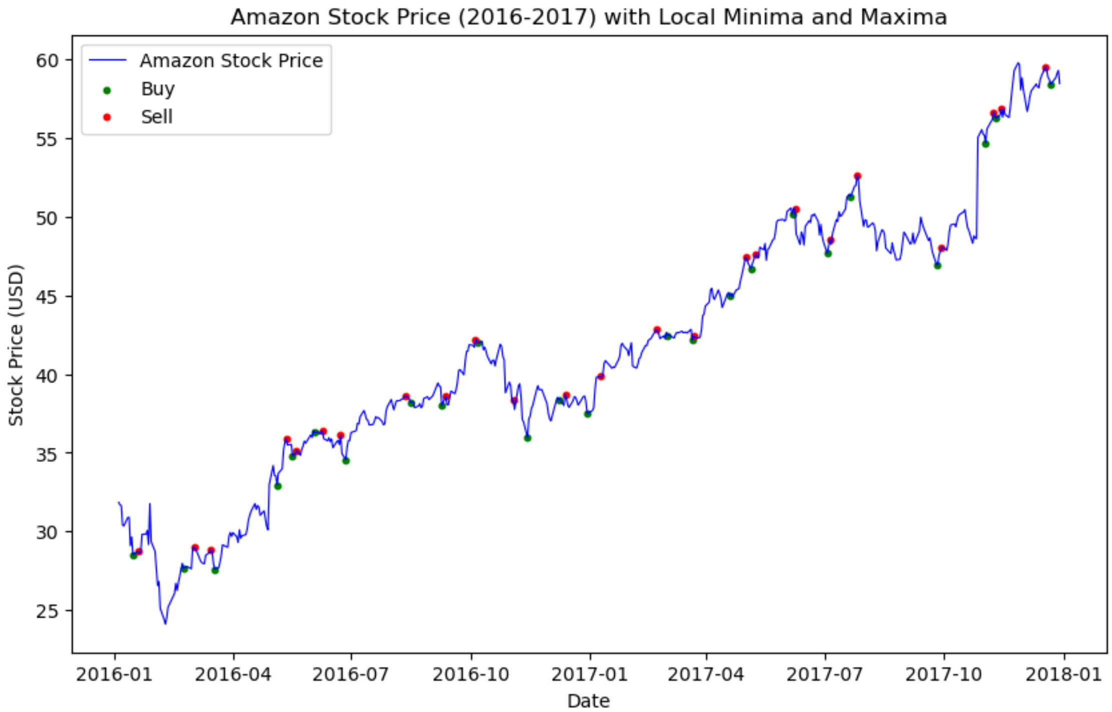
```
/Users/saakshimore/miniconda3/lib/python3.10/site-packages (from
requests>=2.31->yfinance) (2023.7.22)
```

[3]:
```
import yfinance as yf
import pandas as pd
import numpy as np
from ydata_profiling import ProfileReport

from scipy.signal import argrelextrema
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import train_test_split, cross_val_score,␣
↪GridSearchCV
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import make_scorer, accuracy_score
from sklearn import metrics
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import classification_report

%matplotlib inline
import matplotlib.pyplot as plt
import seaborn as sns
from pandas.plotting import scatter_matrix

pd.options.mode.chained_assignment = None
```

[4]:
```
# Fetch Amazon stock data from Yahoo Finance
symbol = "AMZN"
start_date = "2016-01-01"
end_date = "2017-12-31"

amazon_data = yf.download(symbol, start=start_date, end=end_date)

# Find local minima and maxima
minima_idx = argrelextrema(amazon_data["Close"].values, np.less)[0]
maxima_idx = argrelextrema(amazon_data["Close"].values, np.greater)[0]

# Set window size
window_size = 5
# Function to filter only the most extreme value within each window
def filter_extrema_indices(extrema_indices):
    filtered_indices = []
    for i in range(0, len(extrema_indices), window_size):
        window_indices = extrema_indices[i:i+window_size]
        if len(window_indices) > 0:  # Check if the window has at least one␣
↪element
            # Choose the most extreme value within the window
```

```
            extreme_index = min(window_indices, key=lambda idx: abs(amazon_data["Close"].iloc[idx])) if i // window_size % 2 == 0 else max(window_indices, key=lambda idx: abs(amazon_data["Close"].iloc[idx]))
            filtered_indices.append(extreme_index)
    return filtered_indices

# Filter extrema to only include the most extreme value within each window
filtered_minima = filter_extrema_indices(minima_idx)
filtered_maxima = filter_extrema_indices(maxima_idx)

# Plot the stock price over time
plt.figure(figsize=(10, 6))
plt.plot(amazon_data.index, amazon_data["Close"], label="Amazon Stock Price", color="blue", linewidth=0.8)

# Mark local minima and maxima over a 1-week period
plt.scatter(amazon_data.index[filtered_minima], amazon_data["Close"].iloc[filtered_minima], color="green", label="Buy", s=10)
plt.scatter(amazon_data.index[filtered_maxima], amazon_data["Close"].iloc[filtered_maxima], color="red", label="Sell", s=10)

# Set plot labels and title
plt.xlabel("Date")
plt.ylabel("Stock Price (USD)")
plt.title("Amazon Stock Price (2016-2017) with Local Minima and Maxima")
plt.legend()

# Display the plot
plt.show()
```

```
[*********************100%%**********************]  1 of 1 completed
```




## 1.1 Dataset Generation

[5]:
```
# Fetch pepsi stock data from Yahoo Finance
symbol = "PEP"
start_date = "2010-01-01"
end_date = "2022-01-31"
snp500data = yf.download('^SPX', start=start_date, end=end_date)
pepsi_data = yf.download(symbol, start=start_date, end=end_date)
pepsi_data['snp500'] = snp500data["Adj Close"]
pepsi_data.reset_index(inplace=True)
```

```
[*********************100%***********************]  1 of 1 completed
[*********************100%***********************]  1 of 1 completed
```

[6]:
```
# Function to calculate technical indicators
def calculate_technical_indicators(data):
    # Add technical indicators to the DataFrame
    data['Normalized_Price'] = (data['Close'] - data['Low']) / (data['High'] -
 ↪data['Low'])
    data['5-day MA'] = data['Adj Close'].rolling(window=5).mean()
    data['25-day MA'] = data['Adj Close'].rolling(window=25).mean()
    data['60-day MA'] = data['Adj Close'].rolling(window=60).mean()
    data['MACD'] = data['Adj Close'].ewm(span=12, adjust=False).mean() -
 ↪data['Adj Close'].ewm(span=26, adjust=False).mean()
```

```
    data['MACD Signal'] = data['MACD'].ewm(span=9, adjust=False).mean()
    data['MACD Histogram'] = data['MACD'] - data['MACD Signal']
    data['Percentage Change to S&P 500'] = data['Close'].pct_change() /␣
↪data['snp500'].pct_change()
    data['RSI'] = 100 - (100 / (1 + (data['Close'].diff(1).where(data['Close'].
↪diff(1) > 0, 0).rolling(window=14, min_periods=1).mean() / -data['Close'].
↪diff(1).where(data['Close'].diff(1) < 0, 0).rolling(window=14,␣
↪min_periods=1).mean())))
    data['Volatility'] = data['Adj Close'].rolling(window=60).std()
    data['Bollinger Upper'] = data['Adj Close'].rolling(window=20).mean() + 2 *␣
↪data['Volatility']
    data['Bollinger Lower'] = data['Adj Close'].rolling(window=20).mean() - 2 *␣
↪data['Volatility']
    data['ATR'] = data['High'] - data['Low']
    data['ATR'] = data['ATR'].rolling(window=14).mean()
    data['relative_price'] = (data['Adj Close'] / data['Adj Close'].iloc[0] -␣
↪1) * 100
    return data
```

[7]:
```
def generate_labels(data, window_size=15):
    temp = 0
    # Create an empty list to store the labels
    labels = []
    # Find minima and maxima indices
    minima_idx = argrelextrema(data['relative_price'].values, np.less,␣
↪order=window_size)[0]
    maxima_idx = argrelextrema(data['relative_price'].values, np.greater,␣
↪order=window_size)[0]

    # Iterate through rows to update labels
    for i in range(len(data)):
        # Buy condition: If current index is a minima and temp is 0
        if i in minima_idx and temp == 0:
            labels.append(1)  # Set to 1 for buy
            temp = 1  # Update temp to 1
        # Sell condition: If current index is a maxima and temp is 1
        elif i in maxima_idx and temp == 1:
            labels.append(-1)  # Set to -1 for sell
            temp = 0  # Update temp to 0
        else:
            labels.append(0)  # No action
    # Create 'Label' column based on buy/sell conditions
    data['label'] = labels
    return data
```

[8]:
```python
# Apply technical indicators and generate labels
pepsi_data = calculate_technical_indicators(pepsi_data)
pepsi_data = generate_labels(pepsi_data)
pepsi_data = pepsi_data[pepsi_data['Date'].dt.year != 2022] #we only included␣
↪the additional 2022 data to calculate the future returns
pepsi_data
```

[8]:
```
           Date        Open        High         Low       Close   Adj Close  \
0    2010-01-04   61.189999   61.520000   60.639999   61.240002   41.054611
1    2010-01-05   61.000000   62.099998   60.900002   61.980000   41.550697
2    2010-01-06   61.990002   62.470001   61.230000   61.360001   41.135056
3    2010-01-07   61.349998   61.380001   60.529999   60.970001   40.873600
4    2010-01-08   60.759998   60.820000   60.270000   60.770000   40.739529
…           …         …           …           …           …           …
3016 2021-12-27  169.990005  171.559998  169.770004  171.470001  163.579407
3017 2021-12-28  171.460007  172.789993  171.199997  172.360001  164.428452
3018 2021-12-29  172.789993  173.460007  171.929993  172.970001  165.010376
3019 2021-12-30  173.539993  173.619995  172.229996  172.669998  164.724182
3020 2021-12-31  172.460007  174.020004  172.110001  173.710007  165.716324

        Volume       snp500  Normalized_Price    5-day MA  …  MACD Signal  \
0      6585900  1132.989990          0.681820         NaN  …     0.000000
1      8886000  1136.520020          0.900001         NaN  …     0.007915
2      9998000  1137.140015          0.104839         NaN  …     0.013726
3     10792000  1141.689941          0.517649         NaN  …     0.013712
4      8674700  1144.979980          0.909092   41.070699  …     0.007876
…          …           …                 …          …  …          …
3016   2868800  4791.189941          0.949723  162.093091  …     2.111788
3017   2332100  4786.350098          0.729564  162.654037  …     2.144337
3018   2299500  4793.060059          0.679738  163.426764  …     2.187987
3019   1988900  4778.729980          0.316549  163.941919  …     2.226753
3020   2914900  4766.180176          0.837698  164.691748  …     2.271180

      MACD Histogram  Percentage Change to S&P 500         RSI  Volatility  \
0           0.000000                           NaN         NaN         NaN
1           0.031659                      3.878315  100.000000         NaN
2           0.023245                    -18.336996   54.411736         NaN
3          -0.000057                     -1.588503   42.285684         NaN
4          -0.023341                     -1.138315   37.948668         NaN
…               …                           …           …          …
3016        0.076528                      0.719280   66.710763    5.098360
3017        0.130197                     -5.138231   69.386445    5.032430
3018        0.174599                      2.524517   69.993766    4.927995
3019        0.155065                      0.580122   69.605407    4.819844
3020        0.177707                     -2.293483   66.232872    4.856210

      Bollinger Upper  Bollinger Lower       ATR  relative_price  label
```

```
0                 NaN              NaN       NaN        0.000000      0
1                 NaN              NaN       NaN        1.208357      0
2                 NaN              NaN       NaN        0.195945      0
3                 NaN              NaN       NaN       -0.440903      0
4                 NaN              NaN       NaN       -0.767472      0
…              …              …       …             …     …
3016       169.562425       149.168984  2.297142      298.443444      0
3017       169.875202       149.745483  2.317856      300.511530      0
3018       170.346639       150.634659  2.237142      301.928970      0
3019       170.778329       151.498952  2.191428      301.231865      0
3020       171.475442       152.050603  2.169285      303.648504      0

[3021 rows x 23 columns]
```

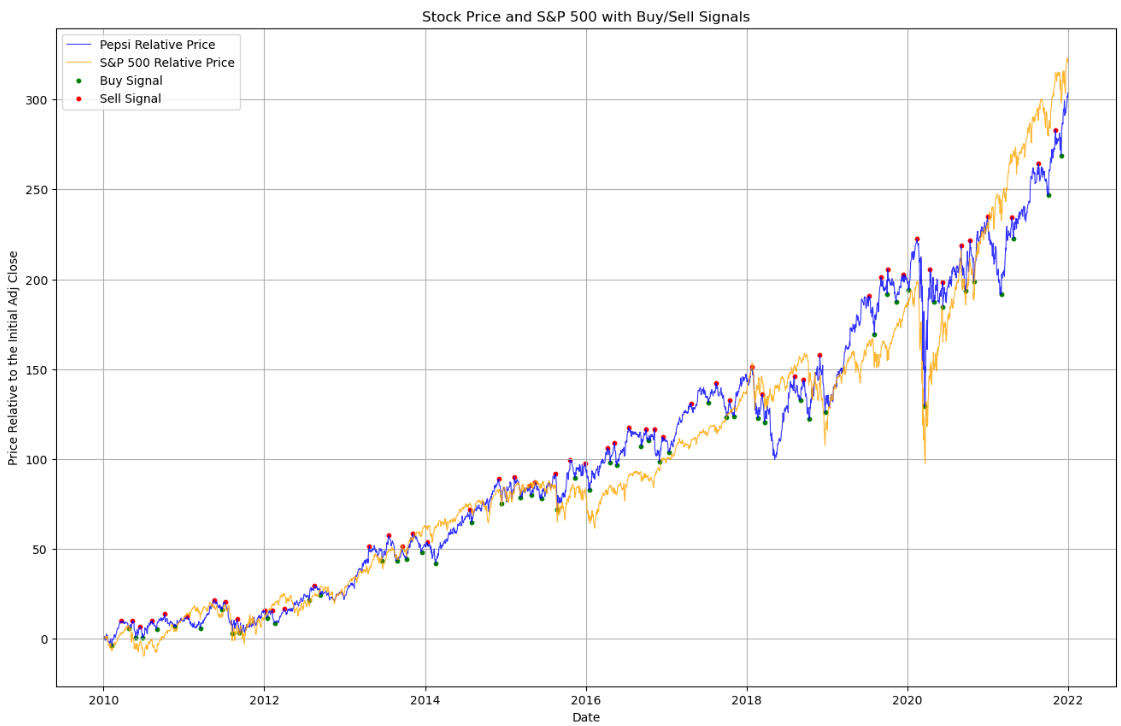
[9]:
```python
pepsi_data['label'].value_counts()
```

[9]:
```
label
 0    2914
 1      54
-1      53
Name: count, dtype: int64
```

[10]:
```python
snp500data.reset_index(inplace=True)
snp500data = snp500data[snp500data['Date'].dt.year != 2022]
snp500_normalized = (snp500data['Close'] - snp500data['Low']) /␣
 ↪(snp500data['High'] - snp500data['Low'])
snp500data['relative_price'] = (snp500data['Adj Close'] / snp500data['Adj␣
 ↪Close'].iloc[0] - 1) * 100
pepsi_data['snp500_relative_price']=snp500data['relative_price']
```

[11]:
```python
# Plot stock price over time
plt.figure(figsize=(16, 10))
plt.plot(pepsi_data['Date'], pepsi_data["relative_price"], label='Pepsi␣
 ↪Relative Price', color='blue', alpha = 0.8, linewidth=0.8)
# Plot normalized S&P 500
plt.plot(snp500data['Date'], snp500data['relative_price'], label='S&P 500␣
 ↪Relative Price', color='orange', alpha = 0.8, linewidth=0.8)

# # Add green dot for 'Label' == 1 and red dot for 'Label' == -1
buy_indices = pepsi_data.loc[pepsi_data['label'] == 1].index
sell_indices = pepsi_data.loc[pepsi_data['label'] == -1].index

# Scatter plot for Buy signals
plt.scatter(pepsi_data.loc[buy_indices, 'Date'], pepsi_data.loc[buy_indices,␣
 ↪'relative_price'], color='green', label='Buy Signal', s=10)
# Scatter plot for Sell signals
```

```
plt.scatter(pepsi_data.loc[sell_indices, 'Date'], pepsi_data.loc[sell_indices,␣
↪'relative_price'], color='red', label='Sell Signal', s=10)

# Customize the plot
plt.title('Stock Price and S&P 500 with Buy/Sell Signals')
plt.xlabel('Date')
plt.ylabel('Price Relative to the Initial Adj Close')
plt.legend()
plt.grid(True)
plt.show()
```




## 1.2 Exploratory Data Analysis

[12]:
```
pd.set_option('display.precision', 3)
pepsi_data.describe()
```

[12]:

| | Date | Open | High | Low | Close | \ |
|---|---|---|---|---|---|---|
| count | 3021 | 3021.000 | 3021.000 | 3021.000 | 3021.000 | |
| mean | 2016-01-02 11:43:04.707050496 | 100.484 | 101.164 | 99.813 | 100.518 | |
| min | 2010-01-04 00:00:00 | 59.290 | 59.660 | 58.500 | 58.960 | |
| 25% | 2013-01-03 00:00:00 | 72.410 | 72.670 | 72.060 | 72.430 | |
| 50% | 2016-01-04 00:00:00 | 99.280 | 99.970 | 98.740 | 99.450 | |
| 75% | 2019-01-03 00:00:00 | 117.390 | 118.320 | 116.710 | 117.480 | |

```
max          2021-12-31 00:00:00    173.540    174.020    172.230    173.710
std                          NaN     27.885     28.114     27.644     27.883

       Adj Close     Volume    snp500  Normalized_Price  5-day MA  …  \
count   3021.000  3.021e+03  3021.000          3021.000  3017.000  …
mean      83.010  5.253e+06  2260.488             0.526    82.983  …
min       39.526  8.833e+05  1022.580             0.000    40.028  …
25%       52.659  3.711e+06  1461.400             0.260    52.807  …
50%       79.586  4.741e+06  2088.480             0.542    79.552  …
75%      100.999  6.050e+06  2798.360             0.790   100.833  …
max      165.716  2.756e+07  4793.060             1.000   164.692  …
std       31.051  2.486e+06   890.502             0.297    30.969  …

       MACD Histogram  Percentage Change to S&P 500       RSI  Volatility  \
count        3021.000                      3020.000  3020.000    2962.000
mean            0.003                          -inf    53.532       2.166
min            -2.347                          -inf     6.977       0.508
25%            -0.135                        -0.375    43.023       1.150
50%             0.002                         0.512    54.477       1.748
75%             0.133                         1.374    64.751       2.800
max             2.347                       939.100   100.000      10.165
std             0.303                           NaN    15.493       1.485

       Bollinger Upper  Bollinger Lower       ATR  relative_price      label  \
count         2962.000         2962.000  3008.000        3021.000  3.021e+03
mean            87.777           79.113     1.350         102.194  3.310e-04
min             44.531           39.588     0.463          -3.723 -1.000e+00
25%             59.174           51.402     0.862          28.265  0.000e+00
50%             84.721           75.880     1.146          93.853  0.000e+00
75%            107.770           94.990     1.590         146.010  0.000e+00
max            171.475          152.051     9.508         303.649  1.000e+00
std             32.498           28.708     0.848          75.632  1.882e-01

       snp500_relative_price
count               3021.000
mean                  99.515
min                   -9.745
25%                   28.986
50%                   84.333
75%                  146.989
max                  323.045
std                   78.597

[8 rows x 24 columns]
```

[13]:
```python
profile = ProfileReport(pepsi_data, title="Profiling Report")
profile
```

```
Summarize dataset:   0%|          | 0/5 [00:00<?, ?it/s]

Generate report structure:   0%|          | 0/1 [00:00<?, ?it/s]

Render HTML:   0%|          | 0/1 [00:00<?, ?it/s]

<IPython.core.display.HTML object>
```

[13]:

The missing value in the 'Percentage Change to S&P 500' column was suspicious. Thus, I decided to dig deeper into this issue.

[14]:
```python
pepsi_data['Percentage Change to S&P 500'].sort_values()
```

[14]:
```
1767        -inf
831    -3300.767
2200    -983.056
1285    -847.101
2988    -658.340
          …
238      200.893
1586     204.828
769      884.222
1155     939.100
0            NaN
Name: Percentage Change to S&P 500, Length: 3021, dtype: float64
```

[15]:
```python
pepsi_data[['snp500','snp500_relative_price','Percentage Change to S&P␣
↪500','relative_price']].iloc[1765:1768 + 1]
```

[15]:
```
       snp500  snp500_relative_price  Percentage Change to S&P 500  \
1765  2276.98                100.971                        -0.407
1766  2268.90                100.258                         2.965
1767  2268.90                100.258                          -inf
1768  2275.32                100.824                        -0.555

      relative_price
1765         109.750
1766         107.543
1767         104.554
1768         104.233
```

After identifying the problem, we redefined the column.

[16]:
```python
pepsi_data['Percentage Change to S&P 500'] = np.where(pepsi_data['snp500'].
↪pct_change() != 0,
         pepsi_data['Close'].pct_change() / pepsi_data['snp500'].pct_change(),
         np.nan)
```

### 1.2.1 Scatter Plot Matrix

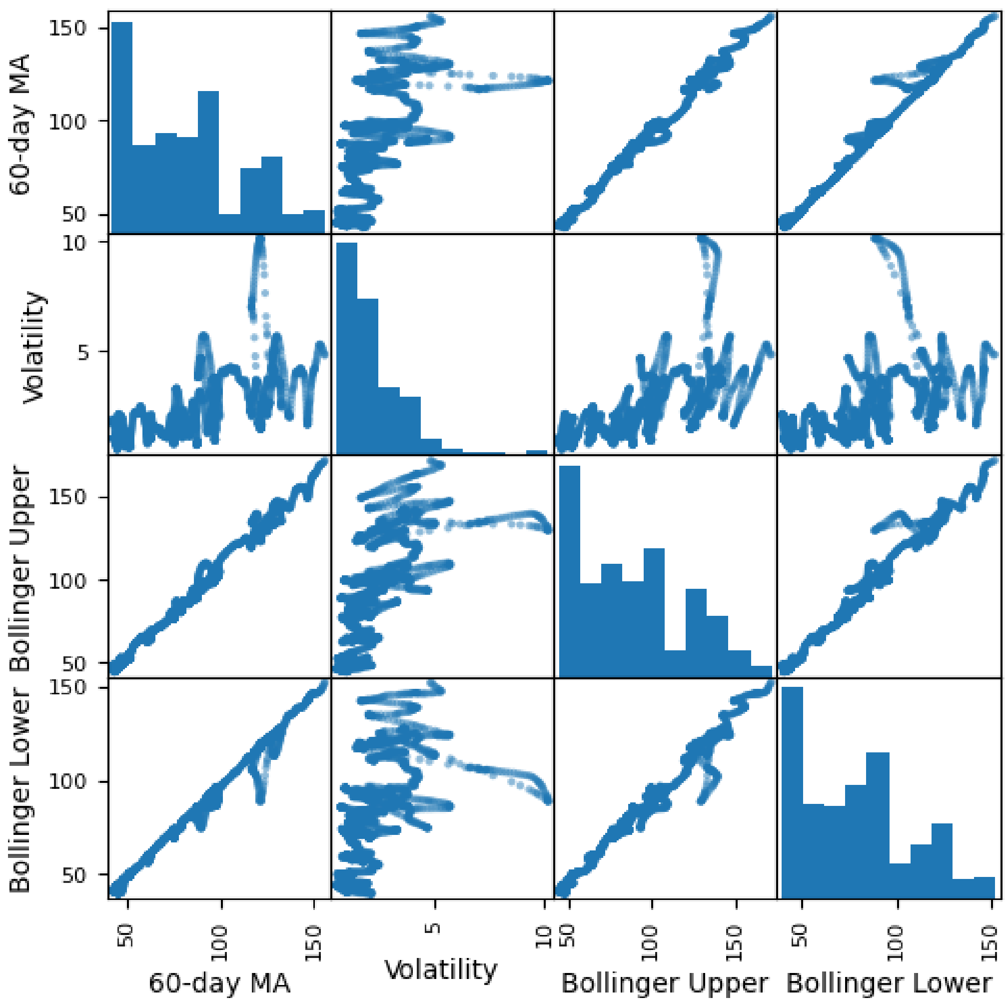
[17]:
```
pepsi_data.columns
```

[17]:
```
Index(['Date', 'Open', 'High', 'Low', 'Close', 'Adj Close', 'Volume', 'snp500',
       'Normalized_Price', '5-day MA', '25-day MA', '60-day MA', 'MACD',
       'MACD Signal', 'MACD Histogram', 'Percentage Change to S&P 500', 'RSI',
       'Volatility', 'Bollinger Upper', 'Bollinger Lower', 'ATR',
       'relative_price', 'label', 'snp500_relative_price'],
      dtype='object')
```

[18]:
```
sub_df = pepsi_data[['60-day MA','Volatility','Bollinger Upper','Bollinger␣
 ↪Lower']]
```

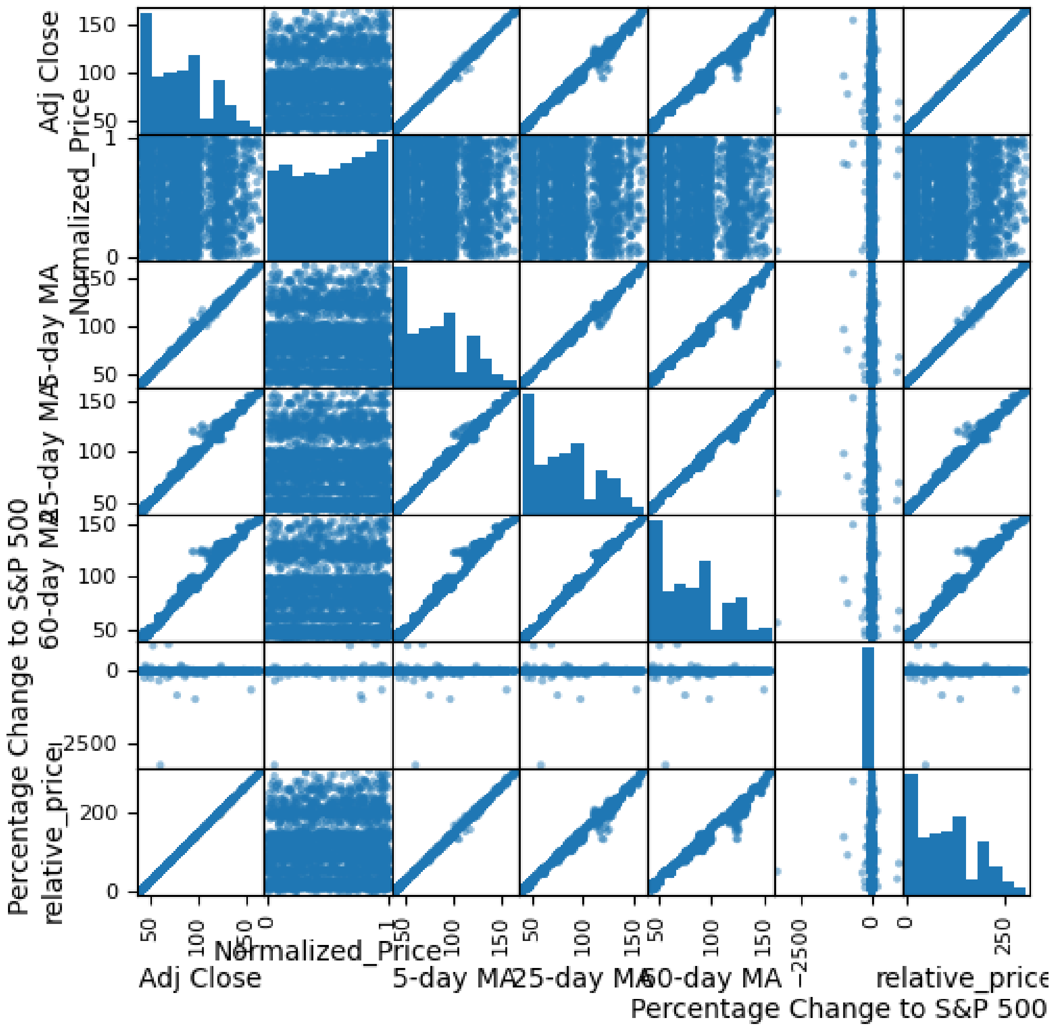
[19]:
```
plt.figure(figsize=(6,6))
scatter_matrix(sub_df,figsize=(6,6))
plt.show()
```

```
<Figure size 600x600 with 0 Axes>
```



[20]:
```
plt.figure(figsize=(6,6))
scatter_matrix(pepsi_data[['Adj Close','Normalized_Price', '5-day MA', '25-day␣
↪MA', '60-day MA','Percentage Change to S&P␣
↪500','relative_price']],figsize=(6,6))
plt.show()
```

```
<Figure size 600x600 with 0 Axes>
```




Now, we have a sense of the ways in which the features are correlated with each other. In the next part, we will preprocess the data and perform feature selection.

### 1.3 Data Pre-processing and Feature Selection

[21]:
```
def undersample(data):
    # Assuming 'label' is the column you want to check for the condition
    # and 'percentage_to_keep' is the percentage of rows with label=0 to keep
    percentage_to_keep = 0.2  # Change this to your desired percentage
    # Identify rows with label=0
    label_0_rows = data[data['label'] == 0]
    # Calculate the number of rows to keep based on the specified percentage
    num_rows_to_keep = int(len(label_0_rows) * percentage_to_keep)
    # Sample a subset of rows with label=0
    undersampled_label_0_rows = label_0_rows.sample(n=num_rows_to_keep,␣
↪random_state=42)
    # Combine the sampled rows with the rest of the data
    undersampled_data = pd.concat([data[data['label'] != 0],␣
↪undersampled_label_0_rows])
    # Resetting index after undersampling
    undersampled_data = undersampled_data.reset_index(drop=True)
    return(undersampled_data)
```

[22]:
```
pepsi_data = pepsi_data.fillna(0)
# pepsi_data['Date'] = pepsi_data.to_datetime(df['Date'])
pepsi_data['Year'] = pepsi_data['Date'].dt.year
pepsi_data['Month'] = pepsi_data['Date'].dt.month
pepsi_data['Day'] = pepsi_data['Date'].dt.day
train_data = pepsi_data[pepsi_data['Year'] < 2021]
undersampled_train_data = undersample(train_data)
test_data = pepsi_data[pepsi_data['Year'] == 2021]
# X = pepsi_data.drop(['label','Date'], axis=1) #since date objects cannot be␣
↪processed by ML models as is
# y = pepsi_data['label']
X_train = undersampled_train_data.drop(columns=['label','Date'], axis=1)
y_train = undersampled_train_data['label']
X_test = test_data.drop(columns=['label','Date'], axis=1)
y_test = test_data['label']
```

[23]:
```
X_train
```

[23]:
```
      Open   High    Low  Close  Adj Close    Volume   snp500  \
0    59.60  59.66  58.75  58.96     39.526   7047700  1056.74
1    66.39  67.00  66.24  66.86     45.141   6412800  1174.17
2    64.93  65.20  64.07  64.23     43.365   8615300  1183.71
3    67.24  67.61  66.81  66.94     45.195  13170400  1171.67
4    63.06  63.06  61.04  61.23     41.339  15152900  1067.95
..     …      …      …      …         …        …        …
628  69.18  69.48  68.64  69.33     50.950   8055900  1462.42
629  84.93  84.99  84.36  84.76     63.175   3832600  1690.91
```

```
630  134.41  134.82  131.22  131.28    119.960   8310100  3097.74
631   66.01   66.65   65.96   66.14     44.654   7374400  1192.13
632   94.20   95.48   94.10   95.34     74.610   8810400  2108.10

     Normalized_Price  5-day MA  25-day MA  …     RSI  Volatility  \
0               0.231    40.297     40.832  …  25.329       0.000
1               0.816    44.941     43.255  …  85.294       0.000
2               0.142    43.844     44.593  …  24.566       1.690
3               0.163    44.448     44.320  …  67.869       1.071
4               0.094    42.518     43.915  …  31.870       0.786
..                …         …         …   …     …           …
628             0.821    50.456     51.206  …  38.900       0.638
629             0.635    63.023     62.934  …  29.842       1.408
630             0.017   120.247    119.956  …  45.627       3.725
631             0.261    44.750     44.771  …  43.869       1.895
632             0.899    74.134     75.728  …  36.524       1.352

     Bollinger Upper  Bollinger Lower    ATR  relative_price  \
0              0.000            0.000  0.964          -3.723
1              0.000            0.000  0.659           9.953
2             47.887           41.127  0.706           5.627
3             46.358           42.073  1.184          10.084
4             45.547           42.403  1.120           0.694
..               …                …     …              …
628           52.490           49.939  0.689          24.104
629           66.237           60.607  0.924          53.880
630          127.261          112.362  2.930         192.197
631           48.600           41.019  0.617           8.768
632           78.140           72.732  1.228          81.733

     snp500_relative_price  Year  Month  Day
0                   -6.730  2010      2    8
1                    3.635  2010      3   23
2                    4.477  2010      4   27
3                    3.414  2010      5   12
4                   -5.741  2010      5   26
..                     …     …     …    …
628                 29.076  2013      1    2
629                 49.243  2013      8    7
630                173.413  2020      6   19
631                  5.220  2010      4   16
632                 86.065  2015      3   20

[633 rows x 25 columns]
```

[24]:
```
# X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
↪random_state=40)
```

```
k=7
scaler = StandardScaler()
model = RandomForestRegressor()
model.fit(X_train, y_train)
feature_importances = model.feature_importances_
selected_features = X_train.columns[feature_importances.argsort()[-k:][::-1]]
```

[25]:
```
selected_features
```

[25]:
```
Index(['RSI', 'Normalized_Price', 'Percentage Change to S&P 500',
       'MACD Histogram', 'Volume', 'Volatility', 'Day'],
      dtype='object')
```

### 1.4 ML Model Optimizing Metric Decision

Now we need to determine which metric to optimize for. Accuracy is not the best scoring metric for a classification problem, especially in the case of imbalanced classification. For classification problems, it is always best to look at precision, or recall, or at the combination of these two metrics: F1-score.

Our problem has one more layer of complexity: it is a multi-class classification problem since we are dealing with 3 outputs, as opposed to the typical binary classification. Thus, the definition of these scoring metrics change significantly. Since we do not have a special inclination towards avoiding false negatives or false negatives, we will work with average f1-score as our scoring metric.

We can either opt for a macro-averaged F1 score or micro-averaged one. The former treats all classes equally whereas the latter treats all instances equally. Let's print both to see which one accurately represents the efficiency of the model.

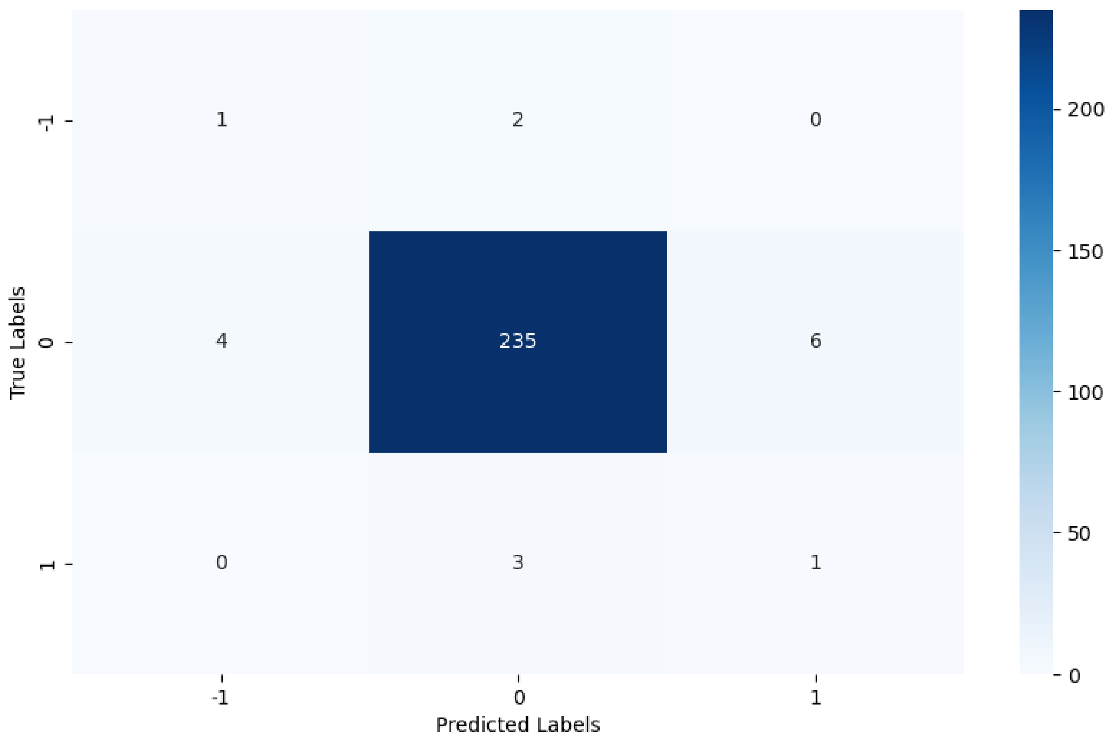
[26]:
```
# X_train = scaler.fit_transform(X_train[selected_features])
# X_test = scaler.transform(X_test[selected_features])
rf_classifier = RandomForestClassifier(n_estimators =
 ↪1000,class_weight='balanced')
X_train_scaled = scaler.fit_transform(X_train[selected_features])
X_test_scaled = scaler.transform(X_test[selected_features])
rf_classifier.fit(X_train_scaled, y_train)
y_pred = rf_classifier.predict(X_test_scaled)
micro_averaged_f1 = metrics.f1_score(y_test, y_pred, average = 'micro')
macro_averaged_f1 = metrics.f1_score(y_test, y_pred, average = 'macro')
print(classification_report(y_test, y_pred, zero_division=0))
print(f'Micro Averaged F1 score with Selected Features: {micro_averaged_f1}')
print(f'Macro Averaged F1 score with Selected Features: {macro_averaged_f1}')
```

```
              precision    recall  f1-score   support

          -1       0.20      0.33      0.25         3
           0       0.98      0.96      0.97       245
           1       0.14      0.25      0.18         4
```

```
    accuracy                           0.94       252
   macro avg       0.44      0.51      0.47       252
weighted avg       0.96      0.94      0.95       252

Micro Averaged F1 score with Selected Features: 0.9404761904761905
Macro Averaged F1 score with Selected Features: 0.46696344892221187
```

[27]:

```
cm = metrics.confusion_matrix(y_test, y_pred)
plt.figure(figsize = (10,6))
sns.heatmap(cm, annot = True, fmt='d', xticklabels = np.sort(y_test.unique()),␣
 ↪yticklabels = np.sort(y_test.unique()), cmap = plt.cm.Blues)
plt.xlabel('Predicted Labels')
plt.ylabel('True Labels')
plt.show()
```




The model hardly ever predicts classes 1 and -1 correctly. However, the micro-averaged F1 score does not reflect this since the value of that metric is 0.94, which could easily illusion one into thinking that the model performed well. The macro-averaged score is 0.47 which correctly reflects the performance of the model at this point. Now we know that our model should be optimizing for the macro-averaged F1 score. Hence, we will do grid search to determine the best parameters with our scoring metric as the macro-averaged F1 score.

### 1.5 ML Model - Grid Search

First, we define our scoring metric - macro averaged F1:

[28]:
```
def true_positive(y_true, y_pred):

    tp = 0

    for yt, yp in zip(y_true, y_pred):

        if yt == 1 and yp == 1:
            tp += 1

    return tp

def true_negative(y_true, y_pred):

    tn = 0

    for yt, yp in zip(y_true, y_pred):

        if yt == 0 and yp == 0:
            tn += 1

    return tn

def false_positive(y_true, y_pred):

    fp = 0

    for yt, yp in zip(y_true, y_pred):

        if yt == 0 and yp == 1:
            fp += 1

    return fp

def false_negative(y_true, y_pred):

    fn = 0

    for yt, yp in zip(y_true, y_pred):

        if yt == 1 and yp == 0:
            fn += 1

    return fn
```

```python
def macro_precision(y_true, y_pred):

    # find the number of classes
    num_classes = len(np.unique(y_true))

    # initialize precision to 0
    precision = 0

    # loop over all classes
    for class_ in list(y_true.unique()):

        # all classes except current are considered negative
        temp_true = [1 if p == class_ else 0 for p in y_true]
        temp_pred = [1 if p == class_ else 0 for p in y_pred]


        # compute true positive for current class
        tp = true_positive(temp_true, temp_pred)

        # compute false positive for current class
        fp = false_positive(temp_true, temp_pred)


        # compute precision for current class
        temp_precision = tp / (tp + fp + 1e-6)
        # keep adding precision for all classes
        precision += temp_precision

    # calculate and return average precision over all classes
    precision /= num_classes

    return precision


def macro_recall(y_true, y_pred):

    # find the number of classes
    num_classes = len(np.unique(y_true))

    # initialize recall to 0
    recall = 0

    # loop over all classes
    for class_ in list(y_true.unique()):

        # all classes except current are considered negative
```

```python
        temp_true = [1 if p == class_ else 0 for p in y_true]
        temp_pred = [1 if p == class_ else 0 for p in y_pred]


        # compute true positive for current class
        tp = true_positive(temp_true, temp_pred)

        # compute false negative for current class
        fn = false_negative(temp_true, temp_pred)


        # compute recall for current class
        temp_recall = tp / (tp + fn + 1e-6)

        # keep adding recall for all classes
        recall += temp_recall

    # calculate and return average recall over all classes
    recall /= num_classes

    return recall

def macro_average_f1(y_true, y_pred):

    # find the number of classes
    num_classes = len(np.unique(y_true))

    # initialize f1 to 0
    f1 = 0

    # loop over all classes
    for class_ in list(y_true.unique()):

        # all classes except current are considered negative
        temp_true = [1 if p == class_ else 0 for p in y_true]
        temp_pred = [1 if p == class_ else 0 for p in y_pred]


        # compute true positive for current class
        tp = true_positive(temp_true, temp_pred)

        # compute false negative for current class
        fn = false_negative(temp_true, temp_pred)

        # compute false positive for current class
        fp = false_positive(temp_true, temp_pred)
```

```
        # compute recall for current class
        temp_recall = tp / (tp + fn + 1e-6)

        # compute precision for current class
        temp_precision = tp / (tp + fp + 1e-6)


        temp_f1 = 2 * temp_precision * temp_recall / (temp_precision +␣
 ↪temp_recall + 1e-6)

        # keep adding f1 score for all classes
        f1 += temp_f1

    # calculate and return average f1 score over all classes
    f1 /= num_classes

    return f1
```

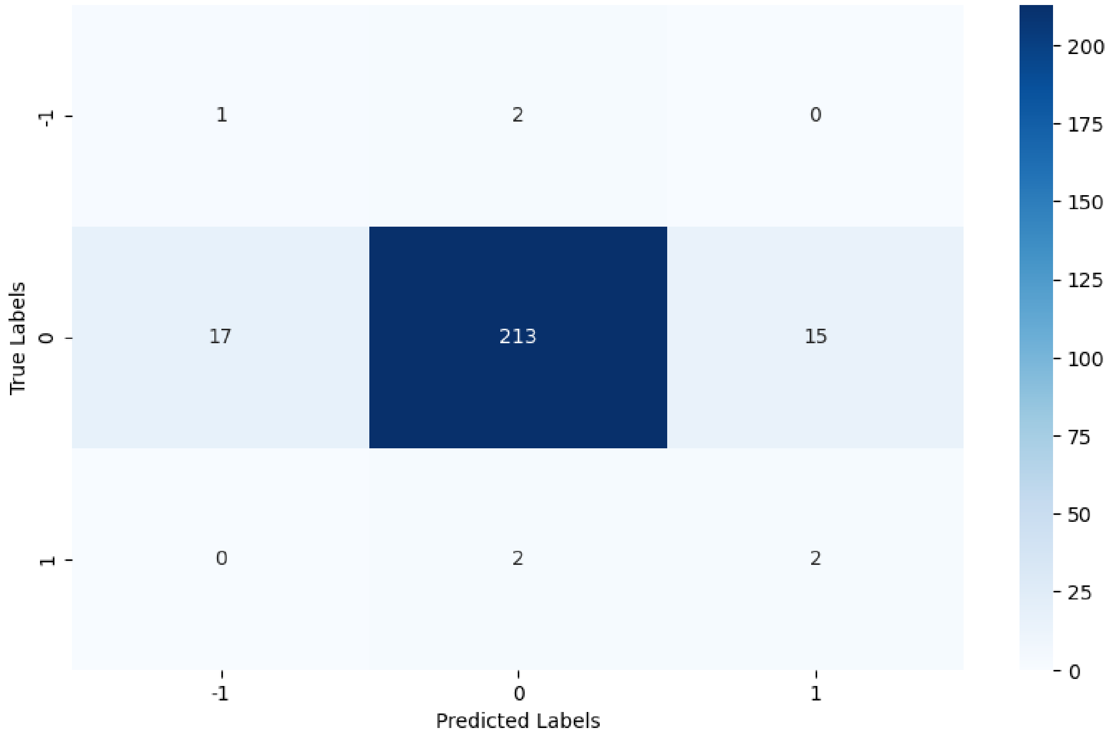
[29]:
```
# X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
 ↪random_state=44)
# X_train = scaler.fit_transform(X_train[selected_features])
# X_test = scaler.transform(X_test[selected_features])
# Define the Random Forest Classifier
rf_classifier = RandomForestClassifier(random_state=38)
# Define the parameter grid for grid search
param_grid = {
    'n_estimators': [50, 100, 150, 400, 800],
    'max_depth': [None, 5, 10, 20],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4],
    'max_features': ['sqrt', 'log2', None],
    'class_weight': ['balanced', None]
}
```

[30]:
```
# Define the scoring metric
scoring_metric = make_scorer(macro_average_f1)
X_train_scaled = scaler.fit_transform(X_train[selected_features])
X_test_scaled = scaler.transform(X_test[selected_features])
# Perform cross-validation
cv_score = cross_val_score(rf_classifier, X_train_scaled, y_train, cv=5,␣
 ↪scoring=scoring_metric)
print(f'Cross-Validation F1-score: {cv_score.mean()}')

# Perform grid search with cross-validation
grid_search = GridSearchCV(rf_classifier, param_grid, cv=5,␣
 ↪scoring=scoring_metric, n_jobs=-1)
```

```
grid_search.fit(X_train_scaled, y_train)

# Get the best hyperparameters from grid search
best_params = grid_search.best_params_
print(f'Best Hyperparameters: {best_params}')

# # Extract feature importances from the best model
# feature_importances = grid_search.best_estimator_.feature_importances_
# # Get the indices of the top k important features
# k = 5  # Change this value based on your preference
# selected_features = X_train.columns[feature_importances.argsort()[-k:][::-1]]
# print(f'Best Features: {selected_features}')

# # Use only the selected features for training and testing
# X_train_selected = X_train[selected_features]
# X_test_selected = X_test[selected_features]

# Train the model on the selected features
best_model_selected = grid_search.best_estimator_
best_model_selected.fit(X_train_scaled, y_train)

# Evaluate the model on the test set with selected features
y_pred = best_model_selected.predict(X_test_scaled)
micro_averaged_f1 = metrics.f1_score(y_test, y_pred, average = 'micro')
macro_averaged_f1 = metrics.f1_score(y_test, y_pred, average = 'macro')
print(classification_report(y_test, y_pred, zero_division=0))
print(f'Micro Averaged F1 score with Selected Features: {micro_averaged_f1}')
print(f'Macro Averaged F1 score with Selected Features: {macro_averaged_f1}')
```

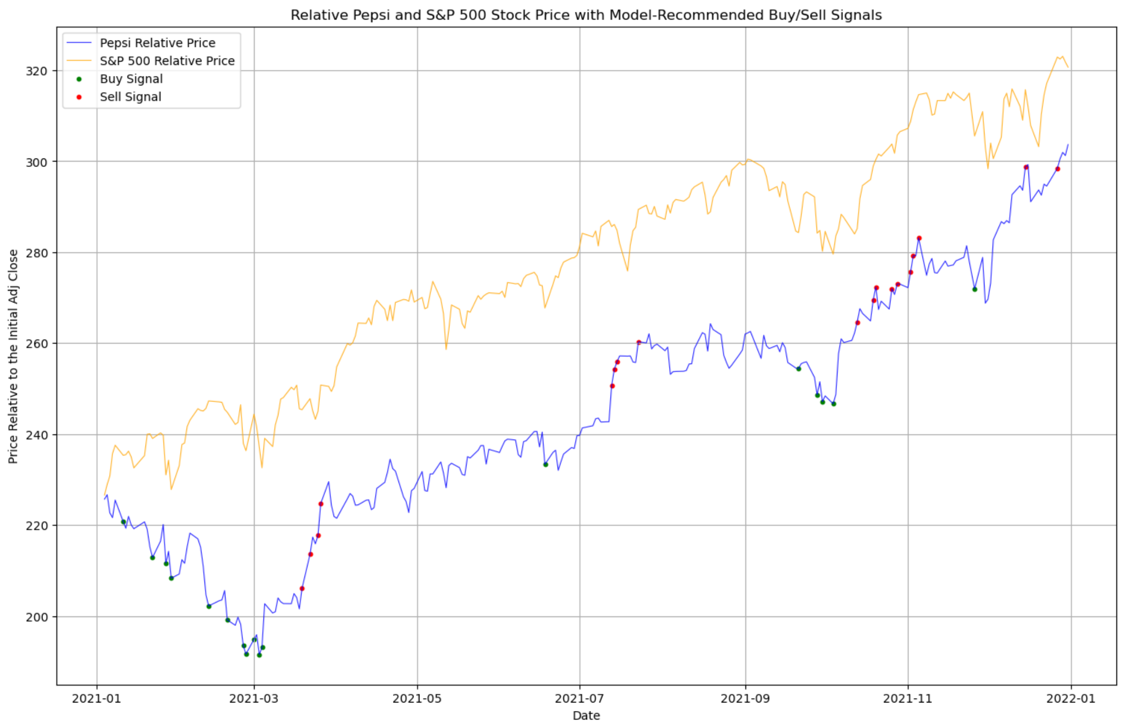
```
Cross-Validation F1-score: 0.5552104815501895
Best Hyperparameters: {'class_weight': 'balanced', 'max_depth': None,
'max_features': 'sqrt', 'min_samples_leaf': 4, 'min_samples_split': 2,
'n_estimators': 100}
              precision    recall  f1-score   support

          -1       0.06      0.33      0.10         3
           0       0.98      0.87      0.92       245
           1       0.12      0.50      0.19         4

    accuracy                           0.86       252
   macro avg       0.38      0.57      0.40       252
weighted avg       0.96      0.86      0.90       252

Micro Averaged F1 score with Selected Features: 0.8571428571428571
Macro Averaged F1 score with Selected Features: 0.4025974025974026
```

[31]:
```
cm = metrics.confusion_matrix(y_test, y_pred)
plt.figure(figsize = (10,6))
sns.heatmap(cm, annot = True, fmt='d', xticklabels = np.sort(y_test.unique()),␣
↪yticklabels = np.sort(y_test.unique()), cmap = plt.cm.Blues)
plt.xlabel('Predicted Labels')
plt.ylabel('True Labels')
plt.show()
```



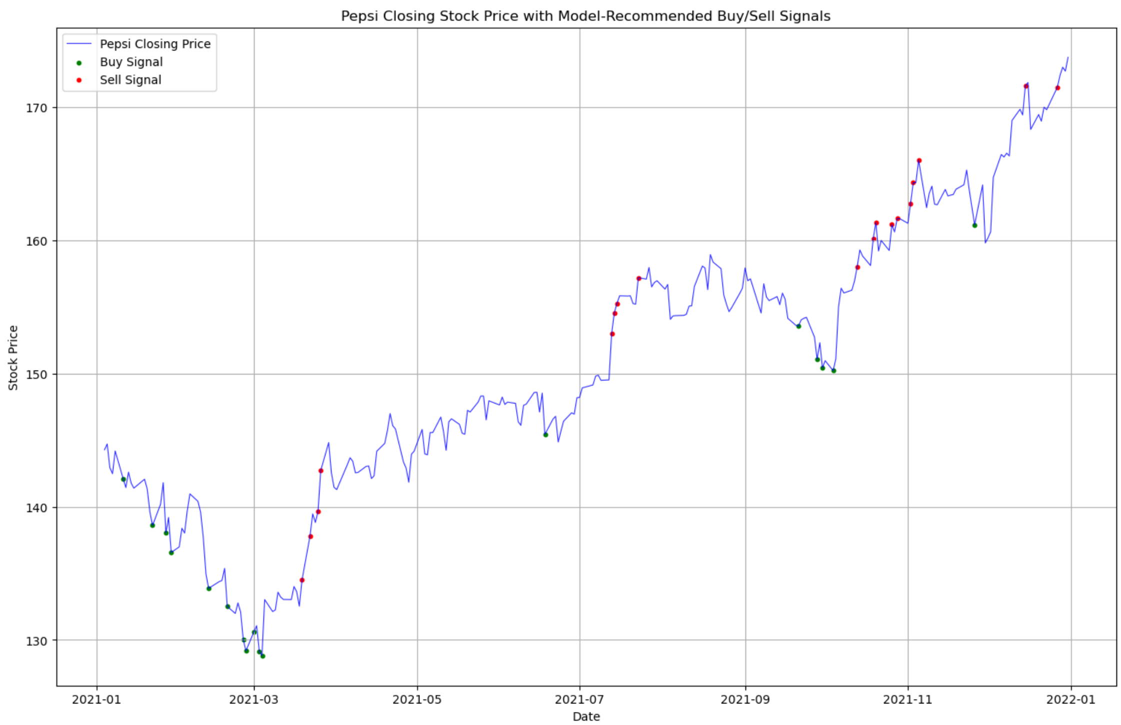

[32]:
```
# Plot stock price over time
plt.figure(figsize=(16, 10))
plt.plot(test_data['Date'], test_data["relative_price"], label='Pepsi Relative␣
↪Price', color='blue', alpha = 0.8, linewidth=0.8)
# Plot normalized S&P 500
plt.plot(test_data['Date'], test_data['snp500_relative_price'], label='S&P 500␣
↪Relative Price', color='orange', alpha = 0.8, linewidth=0.8)

# # Add green dot for 'Label' == 1 and red dot for 'Label' == -1
buy_indices = np.where(y_pred == 1)[0]
sell_indices = np.where(y_pred == -1)[0]
test_data = test_data.reset_index(drop=True)
# Scatter plot for Buy signals
plt.scatter(test_data.loc[buy_indices, 'Date'], test_data.loc[buy_indices,␣
↪'relative_price'], color='green', label='Buy Signal', s=10)
```

```
# Scatter plot for Sell signals
plt.scatter(test_data.loc[sell_indices, 'Date'], test_data.loc[sell_indices,␣
↪'relative_price'], color='red', label='Sell Signal', s=10)
# Customize the plot
plt.title('Relative Pepsi and S&P 500 Stock Price with Model-Recommended Buy/
↪Sell Signals')
plt.xlabel('Date')
plt.ylabel('Price Relative to the Initial Adj Close')
plt.legend()
plt.grid(True)
plt.show()
```



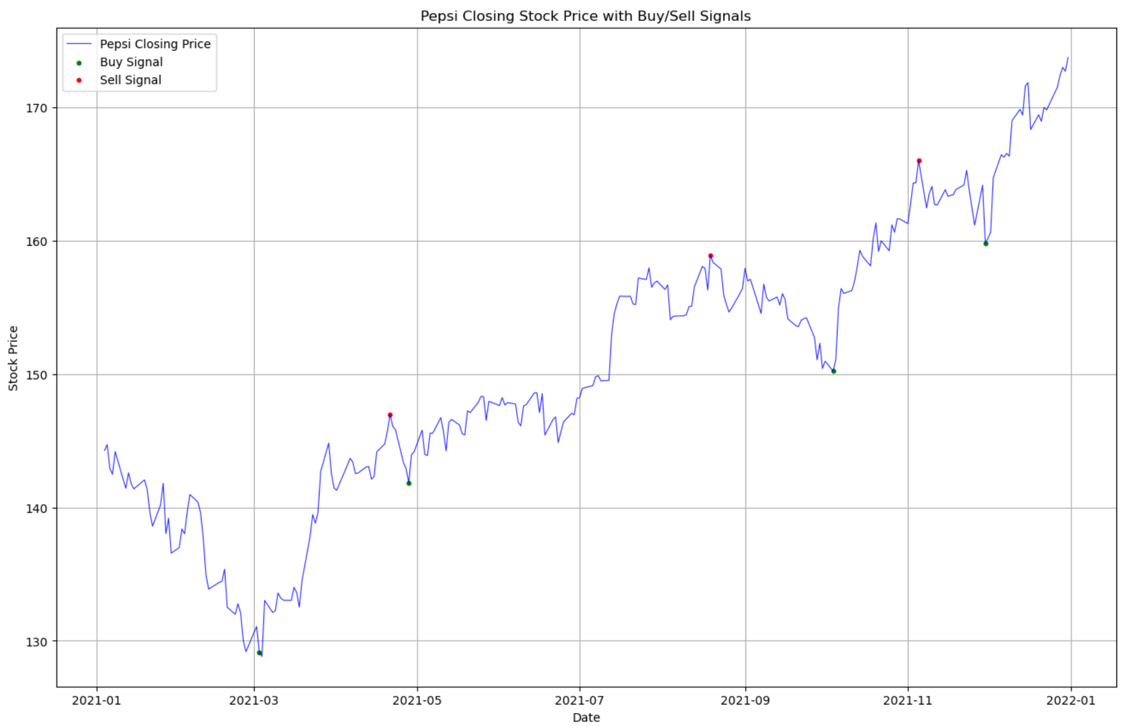

[33]:
```
# Plot stock price over time
plt.figure(figsize=(16, 10))
plt.plot(test_data['Date'], test_data["Close"], label='Pepsi Closing Price',␣
↪color='blue', alpha = 0.8, linewidth=0.8)
# Plot normalized S&P 500
# plt.plot(test_data['Date'], test_data['snp500_relative_price'], label='S&P␣
↪500 Relative Price', color='orange', alpha = 0.8, linewidth=0.8)

# # Add green dot for 'Label' == 1 and red dot for 'Label' == -1
buy_indices = np.where(y_pred == 1)[0]
sell_indices = np.where(y_pred == -1)[0]
```

```
test_data = test_data.reset_index(drop=True)
# Scatter plot for Buy signals
plt.scatter(test_data.loc[buy_indices, 'Date'], test_data.loc[buy_indices,␣
 ↪'Close'], color='green', label='Buy Signal', s=10)
# Scatter plot for Sell signals
plt.scatter(test_data.loc[sell_indices, 'Date'], test_data.loc[sell_indices,␣
 ↪'Close'], color='red', label='Sell Signal', s=10)
# Customize the plot
plt.title('Pepsi Closing Stock Price with Model-Recommended Buy/Sell Signals')
plt.xlabel('Date')
plt.ylabel('Stock Price')
plt.legend()
plt.grid(True)
plt.show()
```




[34]:
```
# Plot stock price over time
plt.figure(figsize=(16, 10))
plt.plot(test_data['Date'], test_data["Close"], label='Pepsi Closing Price',␣
 ↪color='blue', alpha = 0.8, linewidth=0.8)
# Plot normalized S&P 500
# plt.plot(test_data['Date'], test_data['snp500_relative_price'], label='S&P␣
 ↪500 Relative Price', color='orange', alpha = 0.8, linewidth=0.8)
```

```
# # Add green dot for 'Label' == 1 and red dot for 'Label' == -1
buy_indices = np.where(y_test == 1)[0]
sell_indices = np.where(y_test == -1)[0]
test_data = test_data.reset_index(drop=True)
# Scatter plot for Buy signals
plt.scatter(test_data.loc[buy_indices, 'Date'], test_data.loc[buy_indices,␣
↪'Close'], color='green', label='Buy Signal', s=10)
# Scatter plot for Sell signals
plt.scatter(test_data.loc[sell_indices, 'Date'], test_data.loc[sell_indices,␣
↪'Close'], color='red', label='Sell Signal', s=10)
# Customize the plot
plt.title('Pepsi Closing Stock Price with Buy/Sell Signals')
plt.xlabel('Date')
plt.ylabel('Stock Price')
plt.legend()
plt.grid(True)
plt.show()
```




[ ]: