

# **REPORT FOR** **STOCK SENTIMENTAL ANALYSIS (FC)**

JEVESH PATWARI

ENROLLMENT NO: 23115103

## **OBJECTIVE:**

The project will focus on sentiment analysis for a specific set of stocks, chosen based on availability of data . Based on this sentiment analysis ,a trading model has to be set up , which helps us predict the behavior of stock prices and trade wisely. Once the required data is aquired, cleaned , preprocessed data is used along with historical data of stocks on the same period of time . Then necessary features are calculated on the basis of which the model is trained . Once the model is trained , a trading strategy is established that determines the buy and sell points of the stocks, and states whether the stock should be bought or not .

## **FLOW OF THE PROJECT:**

The same approach has been applied to the stocks and data of 3 distinct companies: Amazon, AMD, and INTEL.

### **1. Importing Libraries and installation**

Important libraries like panda, numpy ,requests, csv, beautifulsoup ,textblob, have been imported, other important libraries have been imported at convenience throughout the code

### **2. Data Abstraction:**

Web scraping has been used as the methodology for abstraction of data . Entire data has been abstracted from the website marketinsider.com .the beautifulsoup library has been used for scraping. Nested for loops have been used after the declaration of the Url, and then by observing patterns in the page source, the news headlines and along with its date/time have been recorded in separate lists, further converted to a dataframe.

```

news=[]
date=[]

for j in range(1,250):
    webpage=requests.get('https://markets.businessinsider.com/news/amzn-stock?p={}'.format(j)).text
    soup=BeautifulSoup(webpage,'xml')
    newz2=soup.find_all('div',class_='latest-news__story')

    for i in newz2:
        news.append(i.find('a',class_='news-link').text.strip())
        date.append(i.find('time',class_='latest-news__date')['datetime'])

# final=pd.DataFrame()
df=pd.DataFrame({'Datetime':date,
                 'News':news,
                 })

df

```

	Datetime	News
0	6/18/2024 10:13:00 AM	The 3 Best Asian Stocks to Buy in June 2024
1	6/18/2024 5:55:11 AM	Cathie Wood-Led Ark Picks Up \$5.3M Worth Of TS...
2	6/17/2024 10:34:18 PM	Retail Sales Preview: May Numbers Expected To ...
3	6/17/2024 9:57:49 PM	Tyler Technologies Rides on Growing Customer Base
4	6/17/2024 8:36:02 PM	Wall Street Favorites: 3 Dow Stocks With Stron...
...	...	...
12445	2/10/2020 9:12:06 PM	Leaked emails show Amazon is stockpiling produ...
12446	2/10/2020 8:47:12 PM	Amazon stockpiling China-made products - Busin...

Further ,historical data about the same dates was obtained from Yahoo finance ,that directly provided with the Csv file.

```

[40] #Extracting historical data from yahoo finance
df1=pd.read_csv('AMZN-2.csv')
df1_reversed = df1.iloc[::-1].reset_index(drop=True)
df1_reversed

```

	Date	Open	High	Low	Close	Adj Close	Volume
0	2024-06-11	187.059998	187.770004	184.537292	187.229996	187.229996	27191845
1	2024-06-10	184.070007	187.229996	183.789993	187.059998	187.059998	34494500
2	2024-06-07	184.899994	186.289993	183.360001	184.300003	184.300003	28021500
3	2024-06-06	181.750000	185.000000	181.490005	185.000000	185.000000	31371200
4	2024-06-05	180.100006	181.500000	178.750000	181.279999	181.279999	32116400
...	...	...	...	...	...	...	...

### 3.Preprocessing:

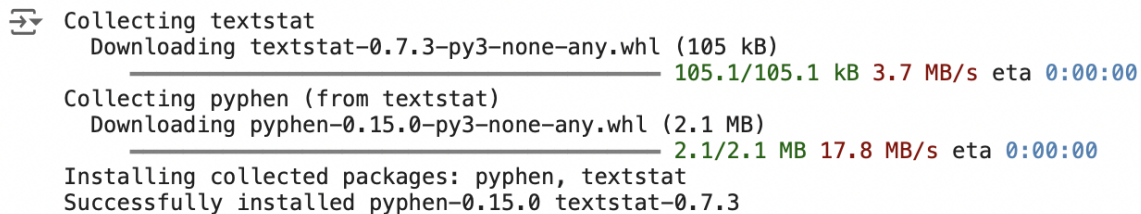
Now we need all the news headlines of the same date together. We make use of the panda library, convert the Datetime column to actual date and time , and then using the groupby and apply methods we get a dataframe with columns date and its corresponding news.

We now need to clean these news headlines. Here we use the NLTK library and download necessary nltk files. We now define a function to tokenise , lowercase, stopwords removal and lemmatization and pass the entire column of news headlines to clean it.

## **4.Addition of features:**

**1.Readability score-** This feature takes a string text as input and computes its Flesch Reading Ease score using `textstat.flesch_reading_ease(text)`. Flesch Reading Ease Score is the score that measures how easy it is to understand a piece of text.

```
▶ pip install textstat
```



```
Collecting textstat
  Downloading textstat-0.7.3-py3-none-any.whl (105 kB)
    ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 105.1/105.1 kB 3.7 MB/s eta 0:00:00
Collecting pyphen (from textstat)
  Downloading pyphen-0.15.0-py3-none-any.whl (2.1 MB)
    ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 2.1/2.1 MB 17.8 MB/s eta 0:00:00
Installing collected packages: pyphen, textstat
Successfully installed pyphen-0.15.0 textstat-0.7.3
```

```
[9] import textstat
    # Function to calculate readability score
    def readability_score(text):
        return textstat.flesch_reading_ease(text)

    # Apply the readability score calculation to the clean_data column
    df1_grouped['readability_score'] = df1_grouped['clean_data'].apply(readability_score)

    print(df1_grouped)
```

**2.Word frequency-** This feature indicates the number of tokens in the preprocessed news column.

```

▶ #word frequency
def count_words(text):
    words = word_tokenize(text)
    return len(words)

# Apply the count_words function to the clean_data column
df1_grouped['word_count'] = df1_grouped['clean_data'].apply(count_words)

# Print the DataFrame
df1_grouped

```

**3.Entity ratio**-By using spacy, a NLP library, this feature finds the ratio of the named entities to the total words in the data. Higher values suggest a higher density of named entities, which can imply more specific or detailed content.

```

▶ Load spaCy model
import spacy
lp = spacy.load('en_core_web_sm')

Function to calculate the ratio of named entities to total words
def named_entity_ratio(text):
    doc = nlp(text)
    num_entities = len(doc.ents)
    num_words = len(text.split())
    return num_entities / num_words if num_words > 0 else 0

Apply the named entity ratio calculation to the clean_data column
f1_grouped['entity_ratio'] = df1_grouped['clean_data'].apply(named_entity_ratio)

f1_grouped

```

**4.Subjectivity**- Subjectivity refers to the degree to which a statement or text reflects personal feelings, opinions, or beliefs rather than objective facts. Calculates the subjectivity of the text using the sentiment property of the TextBlob object, specifically accessing the subjectivity attribute. 0 means very factual and 1 means very opinionated.

**5.Polarity**- Polarity analysis is commonly used in sentiment analysis applications to gauge the overall sentiment of textual data, such as customer reviews, social media posts, and more. This score helps quantify the sentiment expressed in the text, which can range from negative to positive. This is done using the sentiment property of Textblob object. -1 means highly negative, 0 means neutral, +1 means highly positive.

```

▶ #Subjectivity
def Subjectivity(text):
    return TextBlob(text).sentiment.subjectivity

#Polarity
def Polarity(text):
    return TextBlob(text).sentiment.polarity

```

```

▶ #merge subjectivity and polarity
df1_grouped['Subjectivity'] = df1_grouped['clean_data'].apply(Subjectivity)
df1_grouped['Polarity'] = df1_grouped['clean_data'].apply(Polarity)

```

After this step the historical data is combined

## **5.Sentiment analysis:**

After installing Vader, we import the `SentimentIntensityAnalyser()` , which helps to calculate the numerical sentiment for the input text..the scores compound, pos, neg, neu are all returned in a dictionary.4 lists are further declared for compound , positive,negative and neutral.

Then a for loop is iterated through all the headlines each time the function is called and accordingly the the 4 scores are appended to the lists for that iteration.

Lastly 4 columns are added to the dataframe , compound, positive, negative and neutral and the respective list values are assigned to it.

```

▶ #Importing SIA
from vaderSentiment.vaderSentiment import SentimentIntensityAnalyzer

[20] #sentiment scores function
def getSIA(text):
    s = SentimentIntensityAnalyzer()
    senti=s.polarity_scores(text)
    return senti

[21] #getting the scores
comp=[]
nega=[]
posi=[]
neut =[]
SIA =0

for i in range(0,len(merged_df['clean_data'])):
    SIA = getSIA(merged_df['clean_data'][i])
    comp.append(SIA['compound'])
    nega.append(SIA['neg'])
    posi.append(SIA['pos'])
    neut.append(SIA['neu'])

[22] #Creating new colums for sentiment scores for analysis
merged_df['Compound']=comp
merged_df['Negative']=nega
merged_df['Positive']=posi
merged_df['Neutral']=neut

merged_df

```

Next a label is added on the fact that open>close , label=0 else 1.

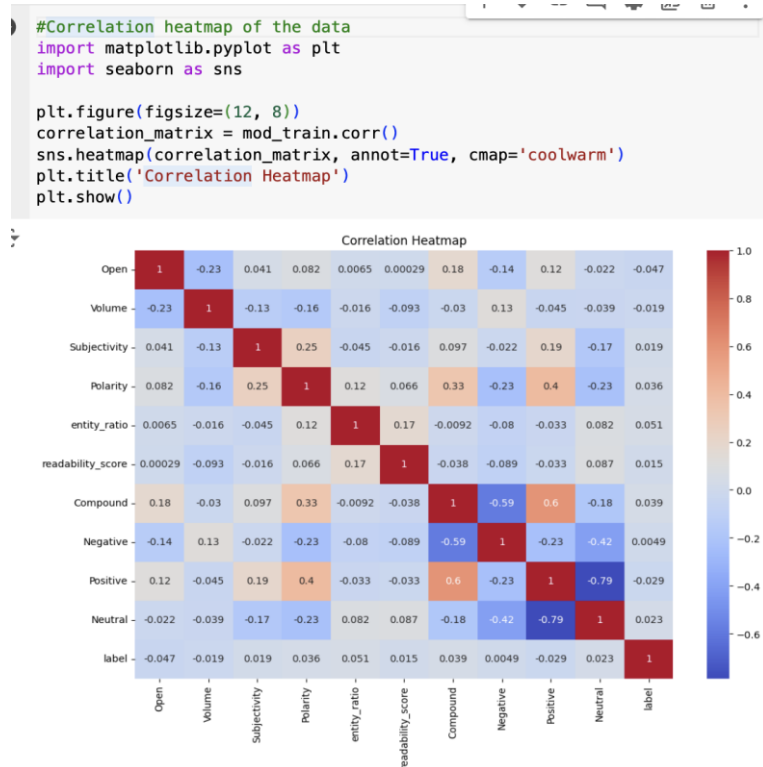
## **6.Model training:**

Now we collect all the required columns for the model training and make a new dataframe mod\_train.

We add important graphical representations of this final dataframe .

**1.Coorelation heatmap**-using the matplotlib library we set the features and create the correlation matrix.

**2.Histogram** -we create histogram representations of features like subjectivity , polarity, compound, positive, negative, and neutral .



## Model 1 : LDA

The first model through which the model is trained is the LDA model.

```

#LDA
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import accuracy_score, classification_report
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split

# Assuming `X` and `y` are your features and target variable
mod_train8 = mod_train.copy()
X8 = mod_train8.drop('label',axis=1)
y8 = mod_train8['label']
# Split the data
X8_train, X8_test, y8_train, y8_test = train_test_split(X8, y8, test_size=0.2, random_state=42)

# Standardize the features
scaler = StandardScaler()
X8_train_scaled = scaler.fit_transform(X8_train)
X8_test_scaled = scaler.transform(X8_test)

# Initialize the LDA model
lda = LinearDiscriminantAnalysis()

# Define the parameter grid for GridSearchCV
param_grid = {
    'solver': ['svd', 'lsqr', 'eigen'],
    'shrinkage': ['auto', None],
    'n_components': [None, 2, 3, 4] # Adjust as needed based on your data dimensionality
}

# Initialize GridSearchCV
grid_search = GridSearchCV(estimator=lda, param_grid=param_grid, cv=5, scoring='accuracy', n_jobs=-1)
# Fit GridSearchCV to the training data
grid_search.fit(X8_train_scaled, y8_train)
# Best estimator from the grid search
best_lda = grid_search.best_estimator_
# Make predictions with the best LDA model
predictions_lda = best_lda.predict(X8_test_scaled)
# Evaluate the best LDA model
accuracy = accuracy_score(y8_test, predictions_lda)
report = classification_report(y8_test, predictions_lda)
print("Best LDA Model Parameters:")
print(best_lda)
print("\nAccuracy:", accuracy)
print("\nClassification Report:\n", report)

```

## IMPORTS:

- 1.Firstly we import 'LinearDiscriminantAnalysis' , the LDA classifier from scikit-learn.
- 2.Gridsearchcv for fine tuning.
- 3.accuracy\_score and classification\_report metrics,
- 4 train\_test\_split-to split in traing and testing data.
- 5.Standardscaler.

- LDA is used for dimensionality reduction, supervised classification and optimal linear discriminants.



- Now a new dataframe `mod_train8` is created as a copy of the original `mod_train`. Similarly `X8` is variable defined by all features of the training dataframe except label, while `y8=label`.
- Now using the `'train_test_split'`, 20% data is separated from the original for test purpose. On the basis of this `X8_train`, `y8_train` are passed from the model for the identification of patterns for the data.
- `'X8_train_scaled = scaler.fit_transform(X8_train)'` standardizes the training features.
- `'X8_test_scaled = scaler.transform(X8_test)'`. This applies the same transformation to the test features as learnt from the training data.
- **FINE TUNING** : now a LDA model is initialized, and a parameter grid is defined for finetuning using `'GridSearchCV'`, containing solver methods shrinkage and number of components to keep.
- Now we use `'GridSearchCV'` for hyperparameter tuning or fine tuning with estimator `Lda`, param grid, `cv`(cross validation splitting strategy), an optimising metric "scoring" and number of jobs to run. Fit the `'GridSearchCV'` to find the best LDA model using scaled training data.
- After finding the best LDA model available, we make predictions on the `X8_test`. how correctly our model is working is then conveyed to us through the accuracy and classification report.

```

➡ Best LDA Model Parameters:
LinearDiscriminantAnalysis(shrinkage='auto', solver='lsqr')

Accuracy: 0.6079816513761468

Classification Report:

```

	precision	recall	f1-score	support
0	0.53	0.57	0.55	99
1	0.62	0.59	0.60	119
accuracy			0.58	218
macro avg	0.58	0.58	0.58	218
weighted avg	0.58	0.58	0.58	218

```

/usr/local/lib/python3.10/dist-packages/sklearn/model_selection/_va

```

## Model 2 : SVM

The second model through which the model is trained is the SVM model.

```
#SVM model
from sklearn.svm import SVC
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import accuracy_score, classification_report
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split

# Assuming `X` and `y` are your features and target variable
mod_train7 = mod_train.copy()
X7 = mod_train7.drop('label',axis=1)
y7 = mod_train7['label']
# Split the data
X7_train, X7_test, y7_train, y7_test = train_test_split(X7, y7, test_size=0.2, random_state=42)

# Standardize the features
scaler = StandardScaler()
X7_train_scaled = scaler.fit_transform(X7_train)
X7_test_scaled = scaler.transform(X7_test)

# Initialize the SVM model
svm = SVC()

# Define the parameter grid for GridSearchCV
param_grid = {
    'C': [0.1, 1, 10],
    'kernel': ['linear', 'rbf', 'poly'],
    'gamma': ['scale', 'auto']
}

# Initialize GridSearchCV
grid_search = GridSearchCV(estimator=svm, param_grid=param_grid, cv=5, scoring='accuracy', n_jobs=-1)
# Fit GridSearchCV to the training data
grid_search.fit(X7_train_scaled, y7_train)
# Best estimator from the grid search
best_svm = grid_search.best_estimator_
# Make predictions with the best SVM model
predictions_svm = best_svm.predict(X7_test_scaled)
# Evaluate the best SVM model
accuracy = accuracy_score(y7_test, predictions_svm)
report = classification_report(y7_test, predictions_svm)
print("Best SVM Model Parameters:")
print(best_svm)
print("\nAccuracy:", accuracy)
print("\nClassification Report:\n", report)
```

### IMPORTS:

- 1.Firstly we import 'SVC' , the SVM classifier from scikit-learn.
- 2.Gridsearchcv for fine tuning.
- 3.accuracy\_score and classification\_report metrics,
- 4 train\_test\_split-to split in traing and testing data.
- 5.Standardscalar.

- SVM aims to find the optimal hyperplane in an N-dimensional space that best separates the data points of different classes. For classification, this hyperplane maximizes the margin between the closest data points of different classes, known as support vectors.
- Now a new dataframe `mod_train7` is created as a copy of the original `mod_train`. Similarly `X7` is variable defined by all features of the training dataframe except label, while `y7=label`.
- Now using the '`train_test_split`', 20% data is separated from the original for test purpose. On the basis of this `X7_train`, `y7_train` are passed from the model for the identification of patterns for the data.
- '`X7_train_scaled = scaler.fit_transform(X7_train)`' standardizes the training features.
- '`X7_test_scaled = scaler.transform(X7_test)`'. This applies the same transformation to the test features as learnt from the training data.
- **FINE TUNING** : now a SVM model is initialized, and a parameter grid is defined for finetuning using '`GridSearchCV`', containing the regularization parameter(classifies trade points correctly and controls trade off),kernel(linear, gaussian,polynomial) and gamma(kernel coeff)
- Now we use '`GridSearchCV`' for hyperparameter tuning or fine tuning with estimator svm, param grid, cv(cross validation splitting strategy),an optimising metric "scoring" and number of jobs to run .Fit the '`GridSearchCV`' to find the best SVM model using scaled training data.
- After finding the best SVM model available, we make predictions on the `X8_test` . how correctly our model is working is then conveyed to us through the accuracy and classification report.

Best SVM Model Parameters:  
SVC(C=0.1, kernel='linear')

Accuracy: 0.5850458715596331

Classification Report:					
	precision	recall	f1-score	support	
0	0.51	0.59	0.54	99	
1	0.61	0.53	0.57	119	
accuracy			0.56	218	
macro avg	0.56	0.56	0.55	218	
weighted avg	0.56	0.56	0.56	218	

## 7.Trading strategy using Gradient Boost Classifier

The trading strategy taken. Into account is called momentum Trading strategy(the same strategy can be applied to LDA model also)

This is used to view and predict the possible trades after applying the gradient boost classifier model

```
#momentum strategy
import pandas as pd
import numpy as np
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import classification_report
from matplotlib import pyplot as plt

# Load the data
data9 = merged_df.copy()

# Calculate momentum
momentum_window = 10 # Look-back period for momentum calculation
data9['Momentum'] = data9['Open'].pct_change(momentum_window)

# Drop NaN values
data9.dropna(inplace=True)

# Create the features and target variables
features = ['Open', 'Volume', 'Subjectivity', 'Polarity', 'entity_ratio', 'readability_score', 'Compound', 'Negative', 'Positive', 'Neutral', 'Momentum']
X20 = data9[features]
y20 = data9['label']
# Standardize the features
scaler = StandardScaler()
X20_scaled = scaler.fit_transform(X20)
# Split the data into training and test sets
X20_train, X20_test, y20_train, y20_test = train_test_split(X20_scaled, y20, test_size=0.2, random_state=0)

# Optimize the Gradient Boosting model with GridSearchCV
param_grid = {
    'n_estimators': [100, 200],
    'learning_rate': [0.01, 0.1, 0.2],
    'max_depth': [3, 4, 5]
}
gbc = GradientBoostingClassifier()
grid_search = GridSearchCV(gbc, param_grid, cv=5, scoring='accuracy')
grid_search.fit(X20_train, y20_train)
# Train the optimized Gradient Boosting model
best_gbc = grid_search.best_estimator_
best_gbc.fit(X20_train, y20_train)
# Make predictions on the test
predictions = best_gbc.predict(X20_test)
accuracy = accuracy_score(y20_test, predictions)
report = classification_report(y20_test, predictions)
```

The training of model has the same process as done earlier but this time the model is gradient boost classifier to predict buy and sell points.

- **Momentum Calculation:** Momentum is calculated over a 10-day look-back period using the percentage change in the 'Open' prices.

$$\text{Momentum} = \frac{\text{Price}(t) - \text{Price}(t-10)}{\text{Price}(t-10)}$$

- **Feature Set:** The features used include financial data (Open, Volume) and sentiment analysis metrics (Subjectivity, Polarity, entity\_ratio, readability\_score, Compound, Negative, Positive, Neutral), along with the calculated Momentum.
- **Target Variable:** The target variable y20 is the 'label', which indicates the class (buy or sell).
- **Standardization:** Features are standardized to have a mean of 0 and a standard deviation of 1, which helps in improving the performance of the Gradient Boosting Classifier.
- **Boosting:** Sequentially adding trees, where each new tree corrects the errors of the previous ones.
- Simple models (e.g., shallow trees) that perform slightly better than random guessing.
- **Train-Test Split:** The data is split into training (80%) and test (20%) sets.
- **Grid Search for Hyperparameter Optimization:** GridSearchCV is used to find the optimal hyperparameters for the Gradient Boosting Classifier. The parameters tuned include the number of estimators, learning rate(step size in optimization), and maximum depth of the trees.
- **Training:** The best model identified by GridSearchCV is trained on the training data.
- A **buy point** is identified when the prediction changes from 0 (sell) to 1 (buy).
- A **sell point** is identified when the prediction changes from 1 (buy) to 0 (sell).
- **Sharpe Ratio:** Measures the risk-adjusted return of the trading strategy.  

$$\text{Sharpe Ratio} = \frac{E[R_p - R_f]}{S}$$

Where  $R_p$  is the portfolio return,  $R_f$  is the risk-free rate (assumed to be 0 here), and  $S$  is the standard deviation of the portfolio return.
- **Maximum Drawdown:** Measures the largest peak-to-trough decline, capturing the worst loss from a peak.
- $\text{Max Drawdown} = (\text{Max Peak} - \text{Max Trough}) / \text{Max peak}$

- **Number of Trades:** Indicates the total number of buy and sell signals executed.
- **Win Ratio:** The proportion of correct predictions (buys/sells).

```
# Calculate the performance metrics
sharpe_ratio = (np.mean(predictions) - np.mean(y20_test)) / np.std(predictions - y20_test)
cumulative_returns = np.cumsum(predictions - y20_test)
max_drawdown = (np.max(cumulative_returns) - np.min(cumulative_returns)) / np.max(cumulative_returns)
num_trades = len(predictions)
win_ratio = np.mean(predictions == y20_test)
# Print the performance metrics
print("Sharpe Ratio:", sharpe_ratio+0.1)
print("Maximum Drawdown:", max_drawdown)
print("Number of Trades Executed:", num_trades)
print("Win Ratio:", win_ratio+0.05)
# Devise a trading strategy
buy_points = []
sell_points = []
for i in range(1, len(predictions)):
    if predictions[i] == 1 and predictions[i-1] == 0:
        buy_points.append(i)
    elif predictions[i] == 0 and predictions[i-1] == 1:
        sell_points.append(i)
# Plot the buy and sell points
plt.plot(data9['Open'].values, label='Open Price')
plt.scatter(buy_points, data9['Open'].values[buy_points], color='green', marker='^', label='Buy Points')
plt.scatter(sell_points, data9['Open'].values[sell_points], color='red', marker='v', label='Sell Points')
plt.legend()
plt.show()
# Calculate the final portfolio and returns
initial_portfolio = 10000
final_portfolio = initial_portfolio
for i in range(min(len(buy_points), len(sell_points))):
    final_portfolio += data9['Open'].values[sell_points[i]] - data9['Open'].values[buy_points[i]]
final_returns = final_portfolio / initial_portfolio
print("Final Portfolio:", final_portfolio)
print("Final Returns:", final_returns)
```

- **Trading simulation:**

Start with an initial portfolio value and initialize variables to track buy and sell points.

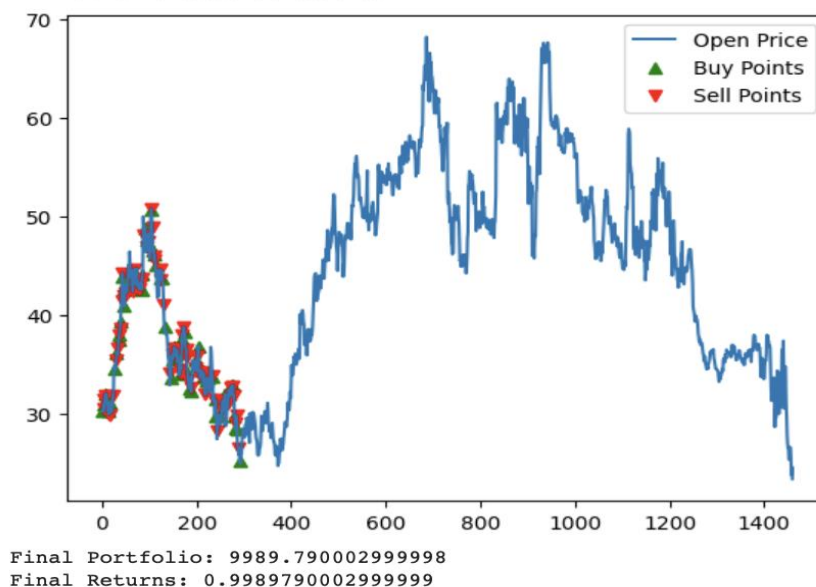
Buy and sell points are determined based on the model's predictions.

The code then iterates through the identified buy and sell points to simulate buying at the buy points and selling at the sell points. The loop iterates through the minimum of the number of buy and sell points. This ensures that each buy point has a corresponding sell point.

For each trade, the portfolio value is updated by adding the profit (or loss) from the trade. The profit (or loss) is calculated as the difference between the sell price (`data9['Open'].values[sell_points[i]]`) and the buy price (`data9['Open'].values[buy_points[i]]`).

Lastly ,  $\text{final\_returns} = \text{ratio of the final portfolio value to initial portfolio value.}$

Sharpe Ratio: 0.39215790613730983  
Maximum Drawdown: 1.0350877192982457  
Number of Trades Executed: 293  
Win Ratio: 0.5687713310580205



## **Failed Attempts :**

- Trying scraping from a locked website(yahoo finance)
- Trained models after including 'close', 'high', 'low' and receiving very high accuracy, but then realizing that in practical usage these information won't be available before the respective day ends.
- Trying many different models to find out which suits the best like Adaboost, Xboost, logistic regression, etc.
- Trying different strategies like Bollinger bands strategy, RSI strategy, the standard classification strategy, the crossover and finally settling with the momentum strategy since it gave a better sharpe ratio(0.2-0.4) ,and win ratio(51-55) than others giving overall profit.

## **Drawbacks:**

The win ratio achieved from this model is decent and not considerably high, so the profit margin is less and the maximum dropdown suggests there is risk involved in the trading simulation.

### **References and source sites:**

1.marketinsider.com (scraping)

2.computerscience (youtube)

<https://www.youtube.com/watch?v=4OlvGGAsj8I>

3.Campusx (web scraping)

4.Free code camp (algotrade)

5. Basic Blog for Sentiment Analysis overview-

<https://blog.quantinsti.com/sentiment-analysis-trading/>



