

A REPORT
ON
**MARKET MAKERS:-ANALYZING QUANTITATIVE AND QUALITATIVE DATA,
ESTABLISHING, VALIDATING, AND ENHANCING RULES**

BY
Vishnu Chebolu - 2022A7PS0124P
Kshitiz Bhatta - 2022A7PS0049P

AT

algobulls®

A Practice School-I Station of



BIRLA INSTITUTE OF TECHNOLOGY & SCIENCE, PILANI

(June 2024)

A REPORT
ON
**MARKET MAKERS:-ANALYZING QUANTITATIVE AND QUALITATIVE DATA,
ESTABLISHING, VALIDATING, AND ENHANCING RULES**
(POTENTIALLY LEVERAGING MACHINE LEARNING).

BY

Vishnu Chebolu	-2022A7PS0124P -CS.
Kshitiz Bhatta	-2022A7PS0049P -CS.

Prepared in partial fulfillment of the
Practice School-I Course Nos.
BITS C221/BITS C231/BITS C241

AT

(AlgoBulls, Mumbai)

A Practice School-I Station of



BIRLA INSTITUTE OF TECHNOLOGY & SCIENCE, PILANI

(June 2024)

Acknowledgment

We would like to express our sincere gratitude to the Practice School Division, BITS Pilani for providing us with the golden opportunity to undertake this summer internship cum Practice School at AlgoBulls, Mumbai via the extremely convenient online mode. The experience that we have gained here has been invaluable in enhancing our understanding of the intersectionality between finance and IT sectors wherein we got to learn about the applications of machine learning and statistical analysis to analyzing financial data. This serves as an important introduction to the evolving quantitative trading scene in India.

We are deeply thankful to Mr. Akhil Jain in particular, our mentor at AlgoBulls , for his guidance, support, and insightful feedback throughout this period. Their expertise and willingness to share knowledge has been instrumental in shaping this report.

We are also grateful to the entire team at AlgoBulls, especially those who took the time to explain various aspects of the company's operations and answered our numerous questions patiently and satisfactorily.

We are indebted to AI and LLMs like ChatGPT and AlgoBulls platform itself that made our work easier while working in a field formerly German to us.

Finally, we would like to thank our Faculty-In-charge Prof.Pankaj Arora for his encouragement and understanding during the internship.

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE PILANI
(Rajasthan)
Practice School division

Station: Algobulls

Centre: Mumbai (Online Mode)

Duration: 2 months

Date of Start: 28th May 2024

Date of Submission: 21st July 2024

Title of the Project: MARKET MAKERS:-ANALYZING QUANTITATIVE AND QUALITATIVE DATA, ESTABLISHING, VALIDATING, AND ENHANCING RULES

Names of Students: Vishnu Chebolu, Kshitiz Bhatta

Id numbers- 2022A7PS0124P- CS, 2022A7PS0049P- CS (*in the same order*)

Name of Expert(s) - Mr. Akhil Jain

Name of the PS faculty- Prof. Pankaj arora

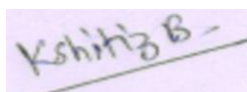
Key Words- finance, trading, python, technical analysis, stocks, returns

Project areas- Quantitative Finance, Machine Learning

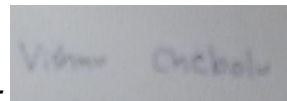
Abstract- Project on market analysis and training machine learning models for algorithmic trading. Three projects along with one elementary project are presented. The projects use the following technologies: Jupyter Notebook, Python, and Github as a hosting platform.

Signature(s) of Student(s)-

Kshitiz Bhatta:



Vishnu Chebolu:



Signature of PS Faculty -

Dr. Pankaj Arora:

Date - 19th June 2024

TABLE OF CONTENTS

	Pg. No.
Cover	1
Title Page	2
Acknowledgement	3
Abstract Sheet	4
Table of Contents	5
Introduction	6
Main Text	7-77
Conclusions and Recommendations	78
References	79

INTRODUCTION

AlgoBulls is an AI-backed algorithmic platform meant to make trading easy and intuitive for the masses. Our first meet was meant to provide a smooth segue from academia to industry. It started with an introduction to the trading scene in India and elsewhere which sensitized us towards the utility of the platform and its key features such as Python Build. This was followed by an introduction to various finance terms. Since we were new to finance, the jargon sounded a bit unfamiliar to us. Nevertheless, we were fully able to appreciate the idea and how this could be driven by the power of neural nets.

AlgoBulls' platform (much like Anaconda) has many utilities like a python environment, notebooks for statistical analysis along with ready-made trading models available. This is meant to simplify package management and deployment and eased our introduction into the field.

The purpose of this report is to highlight the work done by us during our internship stint. The first few weeks were spent in getting used to the underlying financial concepts and getting used to the interface and technologies required for the job. The last weeks was when we did the 'actual' work.

Some time was also allotted to exploring machine learning and deep learning algorithms (through Andrew NG's Coursera courses and credible YouTube channels) that could be used for this purpose.

FINANCIAL LEARNINGS

Phase-I:

We started off by watching Khan Academy's videos on financial markets to understand basic trading terms and get ourselves to terms with the jargon-heavy industry. Firstly we understood the difference between bonds and stocks.

Put in simple words, bonds are when you **lend** the company and stocks are when you are a **part owner** of the company. The age old financial adage "Buy Low, Sell High" guides us.

Then we went on to learn about balance sheets and the fundamental financing principle- assets of an enterprise being its liability taken together with its equity. We understood that the share value of a company depends directly on the equity. Shares rise when a company does well and fall when it fares poorly.

Then we went on to learn terms like market capitalization (the large-cap, mid-cap and small-cap classification) and average volume. We mostly deal with large-cap companies in the projects that follow for the media coverage related to them is greater and interesting for beginner-experimentation.

We then learnt about short trading of stocks and how it can affect the market. We understood the role of short-sellers as players who scrutinize the stock markets meticulously.

This was followed by the basics of investing and how price alone does not tell whether a stock is cheap or expensive (learning about various metrics such as Dividend Yield, P/S ratio, P/B ratio and P/E ratio).

Then came the income statement where we learnt about gross profit, operating profit, return on assets, pretax income, net income, return on equity, earnings per share, price per earnings ratio etc.

These definitions are important for a deeper understanding when one is doing algorithmic trading. In particular, we understood how the P/E ratio is important while judging whether a stock is expensive or cheap.

EBIT(Earnings before interest and taxes), depreciation and amortization were some other terms which we briefly went through.

We then learnt an interesting concept where by just looking at the P/E ratio we can not make an assumption of the assets of the company because it might

be financed completely differently. We then learnt the concept of EBITDA(Earnings before Interest, depreciation and amortization).

We learnt the basics of how the company is born and how it raises money(through angel investors, venture capitalists and finally IPO).

Phase-II:

Then we followed Zerodha's Fundamental Analysis and Technical Analysis course to make our financial-knowledge backbone further robust.

Much of FA was already covered in the Khan Academy videos. Here, we outline the key TA terms used in Algorithmic Trading: (TA is essentially analyzing charts and stats)

(Remark: ChatGPT was heavily used for exploring these definitions and thus no original intellectual property is entailed below)

- Moving Average(MA) : An average of the security's price over a specified period. We shall use this to identify trends and maybe predict future patterns.

- Momentum Oscillator: Measures the amount a security's price has changed over a given period of time. The definition is given alongside, numerator corresponding to current price and denominator denoting the previous price, it is obvious that this value

$$\text{Momentum} = \frac{\text{PriceClose}(i)}{\text{PriceClose}(i - n)} \cdot 100$$

oscillates around 100.

- Relative Strength Index(RSI): We shall use this momentum oscillator to measure the speed and change of price movements, typically used to identify overbought or oversold conditions.
- Bollinger Bands: A set of lines used plotted two standard deviations away (positively and negatively) from a simple moving average, indicating potential overbought or oversold conditions.
- Candlestick Patterns: Charts that display the OHLC (Open-High-Low-Close) prices of a security for a specific period, used to predict future price movements. We use them almost every other second.
- Support and Resistance Levels: Horizontal lines drawn on a price chart to identify levels where the price had historically had difficulty moving above(resistance) or below(support).
- Open Interest: The total number of outstanding contracts that have not been settled or closed. This indicates the liquidity and activity level in a

particular features or options market. Higher open interest suggests a more active market.

- Options Chain: A listing of all available options contracts for a particular security, showing different strike prices and expiration dates.
- Options Payoff: The profit or loss that results from an options position at expiration.
- ATM, OTM and ITM positions: These expand to “At The Money”, “Out of The Money”, and “In The Money” respectively.
- Margins: The amount of money an investor needs to deposit with their broker to cover the credit risk the broker takes on by offering the leverage. ‘Initial Margin’ is the amount required to open a leveraged position while ‘Maintenance Margin’ is the minimum amount of equity that must be maintained in a margin account.

Finally, the webinar uploaded on AlgoBulls’ YT channel helped us understand the interplay between trading and concepts like Cash Trades(Spot Trades), Future Trades and Option Trades; and also introduced us to the financial lingo- buying rights(= call option) and selling rights(=put option), which when was casually thrown upon us caused confusion.

Armed with all this knowledge, we got our hands dirty by exploring various features of the following platforms/ reading about the following topics.

- nseindia.com
- AlgoBulls (with special emphasis on its Python Build and various readymade strategies like Options Strangle and Volatility Trend ATR)
- Bloomberg Terminal (watched a video about it)
- screener.in : Stock analysis and screening tool for investors in India
- Investopedia
- India VIX
- TradingView

Implementation of the Above Ideas in the Project:

(Key Pointers)

- First we clearly specify that we understand the difference between investing and trading despite its almost interchangeable usage in common parlance. Investment corresponds to a long-term plan and is usually based on the fundamental analysis of companies. Trading is usually for a relatively short term (and can be further classified as swing-trading, short-term trading or even intraday trading). This project concerns itself with trading as is the scope of the field.
- The project heavily relies on the analysis of data obtained from trustworthy screeners for stock identification.
- Algorithmic trading is devoid of human emotions and pays strong emphasis on objectivity. But, we can still use machine learning to incorporate emotions-of-sorts and delay decisions (violating a stop-loss under certain conditions) when need be based on past trends and future predictions. We shall try to incorporate a related feature in our project. The modus operandi for the same will be outlined in a future report after we get more expertise in the pertinent domain.
- Our project will carry out a quality-check assessment to examine if the trading results in good returns (return below the FD rate {~ 6 % is considered as the FD mark for practical purposes} is unacceptable and aim for at least a 15-20% return). At the end of the day, it is obviously up to the investor to preserve his capital and seek returns at reasonable risks-reward ratio (a ~1:3 ratio is considered good).
- The quality-check assessment relies essentially on three parameters- profit made, risks and consistency.

THE CODING COMPONENT

Now we were somewhat at peace with the basic finance concepts and were ready to do our project, translating the business logic into an executable form .

To learn about the fundamentals of python packages for data manipulation, visualization and statistical analysis, we watched this video : [Algorithmic Trading Python for Beginners - FULL TUTORIAL](#)

This taught us how to use Jupyter notebooks, and do basic algorithmic trading.

In particular we used **yfinance**, a python library that can import data directly from Yahoo Finance heavily to obtain primary data.

Here is a sample chart one may obtain in this fashion:



We also revisited traditional ML-DL libraries and frameworks like Pandas (for input-output) and Numpy for dealing with arrays.

We revised Linear Regression and Gradient Descent, and thought of its utility in Algorithmic Trading. It won't be surprising to think that HFT algorithms may utilize linear regression and gradient descent for better predictions.

We strive to implement them or something similar in our project. Since many datasets are available, training should result in a good enough accuracy.

After devising a suitable strategy, we will perform a rigorous back-testing.

For that, we plan to explore AlgoBulls' youtube channel further and learn about strategy generation, strategy structure and coding, tweaking strategy parameters, etc.

Relevant developments related to the project will be posted on this GitHub repository:(since this report is submitted in online medium it makes sense to provide the link for the repository in this document.)

Link to the repository: <https://github.com/shxxtzcoder/AlgoTrading> AlgoBulls

We will use technical indicators in our project. For example, stochastic oscillator can be used as a technical indicator to time entry and exit points.
(Image Credits: Investopedia)



Image by Sabrina Jiang © Investopedia 2021

Formula for the Stochastic Oscillator

$$\%K = \left(\frac{C - L14}{H14 - L14} \right) \times 100$$

where:

C = The most recent closing price

L14 = The lowest price traded of the 14 previous trading sessions

H14 = The highest price traded during the same 14-day period

%K = The current value of the stochastic indicator

The Projects

To implement our main project idea, we started off by implementing a simple project which would familiarize us well with what was to come later. This project would first analyze stocks, create a strategy and backtest it to check how useful it is.

Firstly we understood how the finance library works with the following code snippet.

Apple=yf.download("AAPL",start="2010-01-01",end="2021-01-01")							Python
*****100%*****] 1 of 1 completed							
Apple							Python
Date	Open	High	Low	Close	Adj Close	Volume	
2010-01-04	7.622500	7.660714	7.585000	7.643214	6.461977	493729600	
2010-01-05	7.664286	7.699643	7.616071	7.656429	6.473148	601904800	
2010-01-06	7.656429	7.686786	7.526786	7.534643	6.370184	552160000	
2010-01-07	7.562500	7.571429	7.466071	7.520714	6.358408	477131200	
2010-01-08	7.510714	7.571429	7.466429	7.570714	6.400681	447610800	
...	
2020-12-24	131.320007	133.460007	131.100006	131.970001	129.339035	54930100	
2020-12-28	133.990005	137.339996	133.509995	136.690002	133.964935	124486200	
2020-12-29	138.050003	138.789993	134.339996	134.869995	132.181198	121047300	
2020-12-30	135.580002	135.990005	133.399994	133.720001	131.054169	96452100	
2020-12-31	134.080002	134.740005	131.720001	132.690002	130.044678	99116600	
769 rows x 6 columns							

So basically the data of Apple is stored in the variable "Apple" using the yfinance library.

Similarly, we downloaded the data of Coca Cola and Spy Stock for analysis purposes.

Each stock has an adj. close price, close price and a high price. For this learning project we analyzed only the close price because it's the most commonly used indicator for quantitative finance.

Here is a brief description of these terms:

Opening Price: The price at which a stock first trades upon the opening of the market on a given trading day. It is determined by the buy and sell orders placed before the market opens.

Closing Price: The last price at which a stock is traded during a regular trading session. It represents the final price of the stock for that trading day.

Adjusted Closing Price: The closing price of a stock after adjustments for all applicable splits, dividends, and distributions. It provides a more accurate reflection of the stock's value by accounting for corporate actions.

High Price: The highest price at which a stock is traded during a given trading day.

```
close=Stocks.loc[:, "Close"].copy()
```

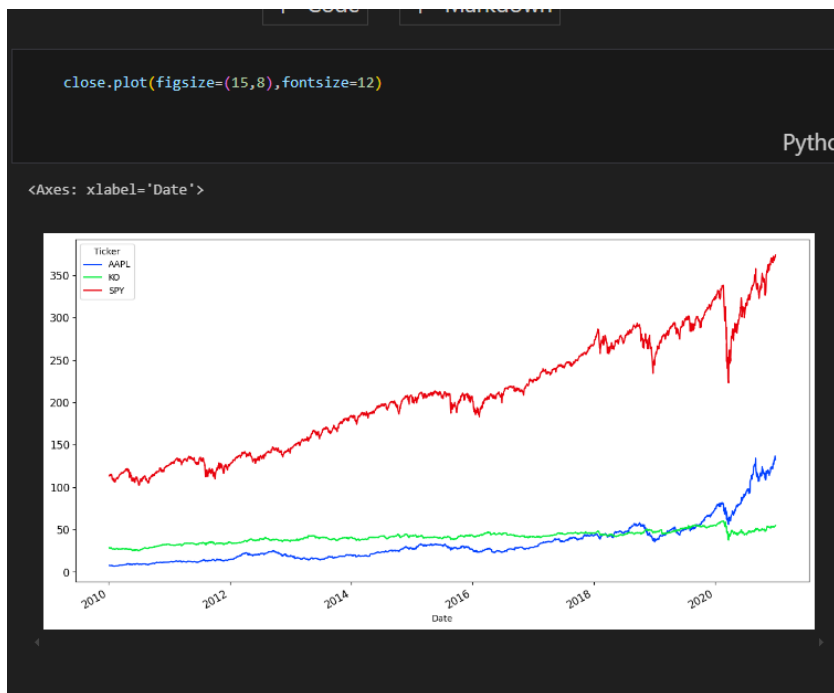
close

Ticker	AAPL	KO	SPY
Date			
2010-01-04	7.643214	28.520000	113.330002
2010-01-05	7.656429	28.174999	113.629997
2010-01-06	7.534643	28.165001	113.709999
2010-01-07	7.520714	28.094999	114.190002
2010-01-08	7.570714	27.575001	114.570000
...
2020-12-24	131.970001	53.439999	369.000000
2020-12-28	136.690002	54.160000	372.170013
2020-12-29	134.869995	54.130001	371.459991
2020-12-30	133.720001	54.439999	371.989990
2020-12-31	132.690002	54.840000	373.880005

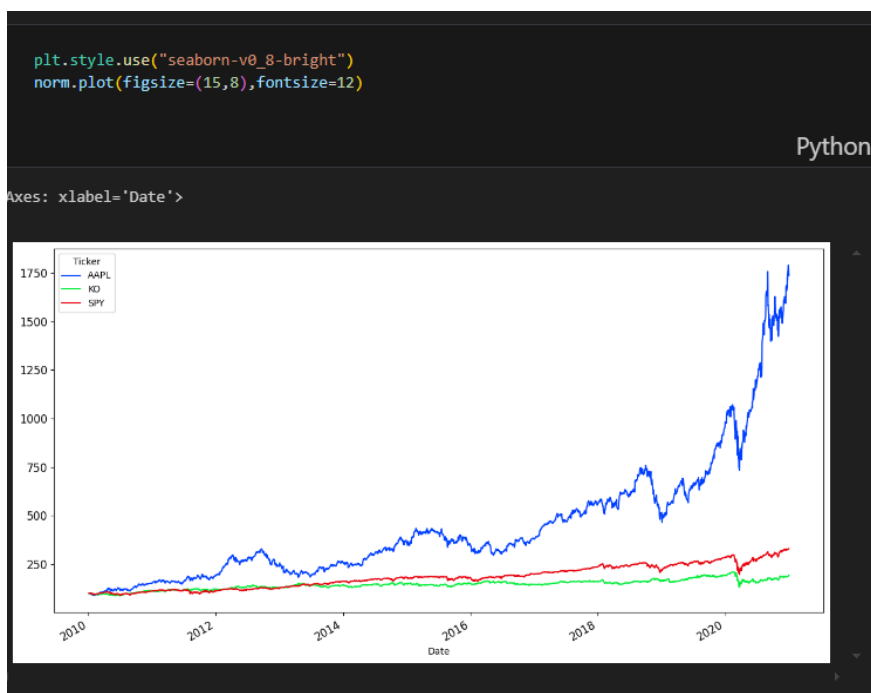
2769 rows × 3 columns

Above data shows the closing price of the 3 stocks on shown dates.

Then we went on to graph it:



This graph is not entirely clear as the starting points for the 3 stocks are different (and thus comparison is difficult). So we decided to normalize it for a clearer picture.



We then decided to analyze the apple stock. We created another column named lag1 which basically contains the stock price the previous day (as is suggested by the name). Using the original price and lag1 price, we created a column called "diff" which is basically how much the stock price has changed in a day.

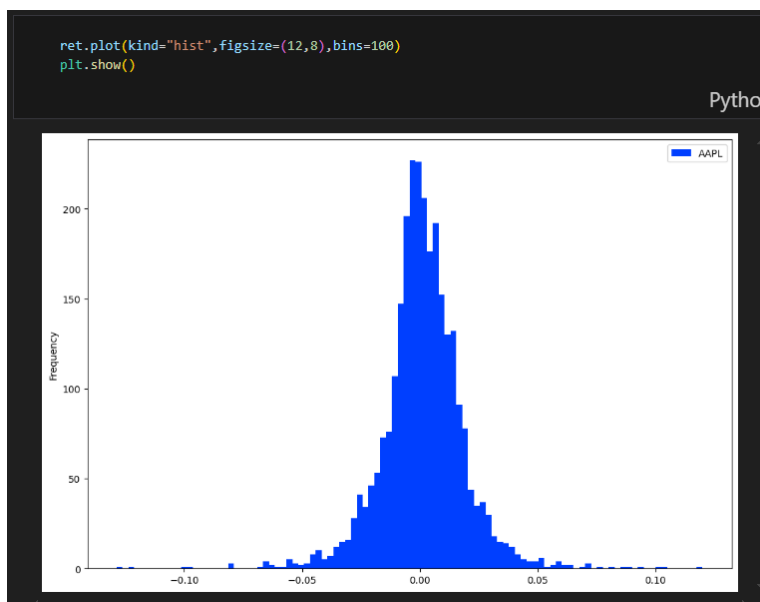
```
apple["diff"]=apple.AAPL.sub(apple.lag1)
```

apple

	AAPL	lag1	diff
Date			
2010-01-04	7.643214	NaN	NaN
2010-01-05	7.656429	7.643214	0.013215
2010-01-06	7.534643	7.656429	-0.121786
2010-01-07	7.520714	7.534643	-0.013929
2010-01-08	7.570714	7.520714	0.050000
...
2020-12-24	131.970001	130.960007	1.009995
2020-12-28	136.690002	131.970001	4.720001
2020-12-29	134.869995	136.690002	-1.820007
2020-12-30	133.720001	134.869995	-1.149994
2020-12-31	132.690002	133.720001	-1.029999

2769 rows x 3 columns

Similarly we calculated the percentage change and named the column as 'Change'. Then we dropped all the NaN (NaN=not a number, denotes undefined behavior) values and took the data of percentage change into a variable names ret, then plotted a histogram to represent this information.

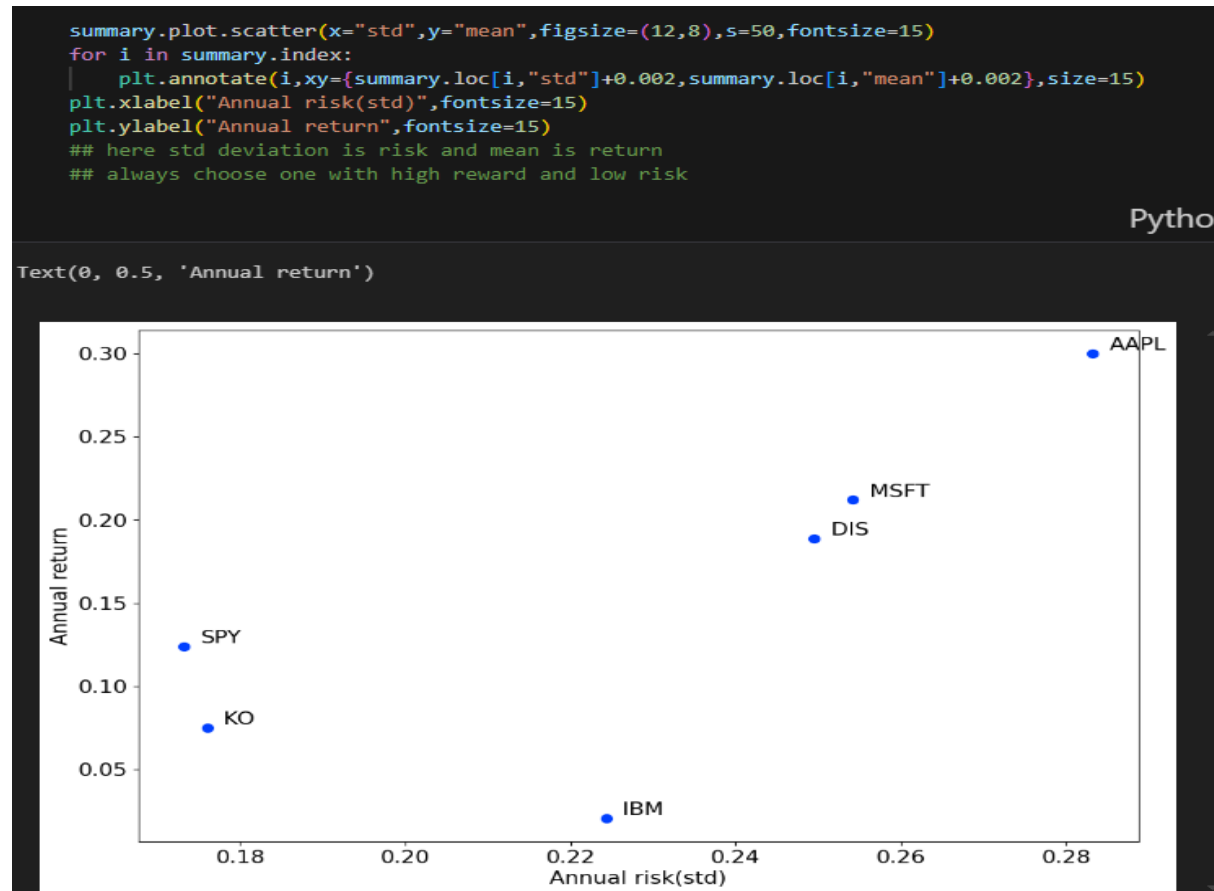


We also calculated the daily mean return, the daily variance and the standard deviation of the return values.

In quantitative finance, the mean of the returns signifies the potential of the stock and the standard deviation signifies its risk. Usually if a stock has high standard deviation of returns it also has a high mean of returns. A person is

supposed to avoid stocks with high standard deviation and low mean (since this would mean high risk with low returns).

We took six stocks and calculated the mean and standard deviation of the stocks and plotted a scatter plot.



We also calculated the correlation and covariance of the stocks. A general principle is to choose stocks which are uncorrelated. A good advice is to go for stocks with $\text{corr} < 0.5$. (The obvious reason is to hedge your investments.)

Mean returns are kind of misleading because one can't do average of percentage change (*think! that does not make sense*). Logarithms come to the rescue.

Logarithmic returns, also known as log returns, are a way of measuring the percentage change in the value of an asset over a period of time. Unlike simple returns, which measure the absolute change in the value of an asset, logarithmic returns measure the relative change in the value of an asset.

The formula for calculating logarithmic returns is:

$$\text{Logarithmic Return} = \ln(\text{Present Value} / \text{Past Value})$$

Formula:

$$L_t \equiv \frac{V_{t+1}}{V_t} - 1.$$

Finally, we were ready to implement our first strategy- Standard Moving Averages Strategy(SMA strategy).

```
data.loc["2016",["sma_s","sma_l","position"]].plot(figsize=(12,8),title="AAPLE - SMA{ } | S",
```

Pyth

```
<Axes: title={'center': 'AAPLE - SMA50 | SMA100'}, xlabel='Date'>
```

AAPLE - SMA50 | SMA100

28 -

27 -

26 -

25 -

24 -

2016-01 2016-03 2016-05 2016-07 2016-09 2016-11 2017-01

Date

sma_s

sma_l

position (right)

-1.00

-0.75

-0.50

-0.25

0.00

0.25

0.50

0.75

1.00

Checking how the strategy fares:

```
data[["returnsb&h","strategy"]].sum()

returnsb&h    6.333003
strategy      1.680623
dtype: float64
```

+ Code + Markdown

```
data[["returnsb&h","strategy"]].sum().apply(np.exp) # what 1$ will be

returnsb&h    562.844539
strategy      5.368897
dtype: float64
```

```
data[["returnsb&h","strategy"]].std()*np.sqrt(252) #annyl std

returnsb&h    0.432578
strategy      0.432734
dtype: float64
```

Clearly, the strategy is not working very well for this stock. We now adjust it for long bias wherein we go 0 when $sma_s < sma_l$ and 1 when $sma_s > sma_l$.

Here are the results:

```
data[["returnsb&h","strategy2"]].sum()

returnsb&h    6.330283
strategy2     4.004092
dtype: float64
```

```
data[["returnsb&h","strategy2"]].sum().apply(np.exp) # what 1$ will be

returnsb&h    561.315156
strategy2     54.822027
dtype: float64
```

```
data[["returnsb&h","strategy2"]].std()*np.sqrt(252)

returnsb&h    0.432604
strategy2     0.334092
dtype: float64
```

This might not be great but it is better than strategy 1. It does reduce risk and also increases reward.

Then we created a function to test the strategy in one line of code:

```
def test_strategy(stock,start,end,SMA):
    df=yf.download(stock,start=start,end=end)
    data=df.Close.to_frame()
    data["returns"]=np.log(data.Close.div(data.Close.shift(1)))
    data["SMA_S"]=data.Close.rolling(int(SMA[0])).mean()
    data["SMA_L"]=data.Close.rolling(int(SMA[1])).mean()
    data.dropna(inplace=True)

    data["position"]=np.where(data["SMA_S"]>data["SMA_L"],1,-1)
    data["strategy"]=data["returns"]*data.position.shift(1)
    data.dropna(inplace=True)
    ret=np.exp(data["strategy"].sum())
    std= data["strategy"].std()*np.sqrt(252)

    return ret,std
```

Python

```
test_strategy("SPY","2000-01-01","2020-01-01",(50,200))
##Now we can use this function whenever we want to test our strategy. It gives us the returns and
```

Python

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

```
(4.766371508372407, 0.18787947185958592)
```

Finally we were done with our preliminary project. We now had a basic understanding of how to analyze stocks, create a basic strategy and backtest it.

Background- Rolling Regression:

(Check references, we found a great article and present the ideas from there.)

Rolling regressions are one of the simplest models for analysing changing relationships among variables overtime. They use linear regression but allow the data set used to change over time. In most linear regression models, parameters are assumed to be time-invariant and thus should not change overtime. Rolling regressions estimate model parameters using a fixed window of time over the entire data set. A larger sample size, or window, used will result in fewer parameter estimates but use more observations.

When setting the width of your rolling regression you are also creating the starting position of your analysis given that it needs the a window sized amount of data begin. An expanding window can be used where instead of a constantly changing fixed window, the regression starts with a predetermined time and then continually adds in other observations until the entire data set is used.

Implementation (Python):

Before beginning with the explanation, we note that R is also a popular tool to use this technique, but here we do not concern ourselves with it.

This example will make use of the statsmodels package, and some of the description of rolling regression has benefitted from the documentation of that package. Rolling ordinary least squares applies OLS (ordinary least squares) across a fixed window of observations and then rolls (moves or slides) that window across the data set. The key parameter is window, which determines the number of observations used in each OLS regression.

First, let's import the packages we'll be using:

```
from statsmodels.regression.rolling import RollingOLS
import statsmodels.api as sm
from sklearn.datasets import make_regression
import matplotlib.pyplot as plt
import pandas as pd
```

Next we'll create some random numbers to do our regression on:

```
X,y= make_regression(n_samples=200,n_features=2,random_state=0,noise=4.0,bias=0)
df=pd.DataFrame(X).rename(columns={0:'feature0',1:'feature1'})
df['target']=y
```

Let's see the first few rows of our data to get an idea where we are heading:

```
df.head()
```

	feature0	feature1	target
0	-0.955945	-0.345982	-36.740556
1	-1.225436	0.844363	7.190031
2	-0.692050	1.536377	44.389018
3	0.010500	1.785870	57.019515
4	-0.895467	0.386902	-16.088554

Now let's fit the model using a formula and a 'window' of 25 steps.

Note that -1 just suppresses the intercept.

```
roll_reg = RollingOLS.from_formula('target ~ feature0 + feature1 -1', window=25, data=df)
model = roll_reg.fit()
```

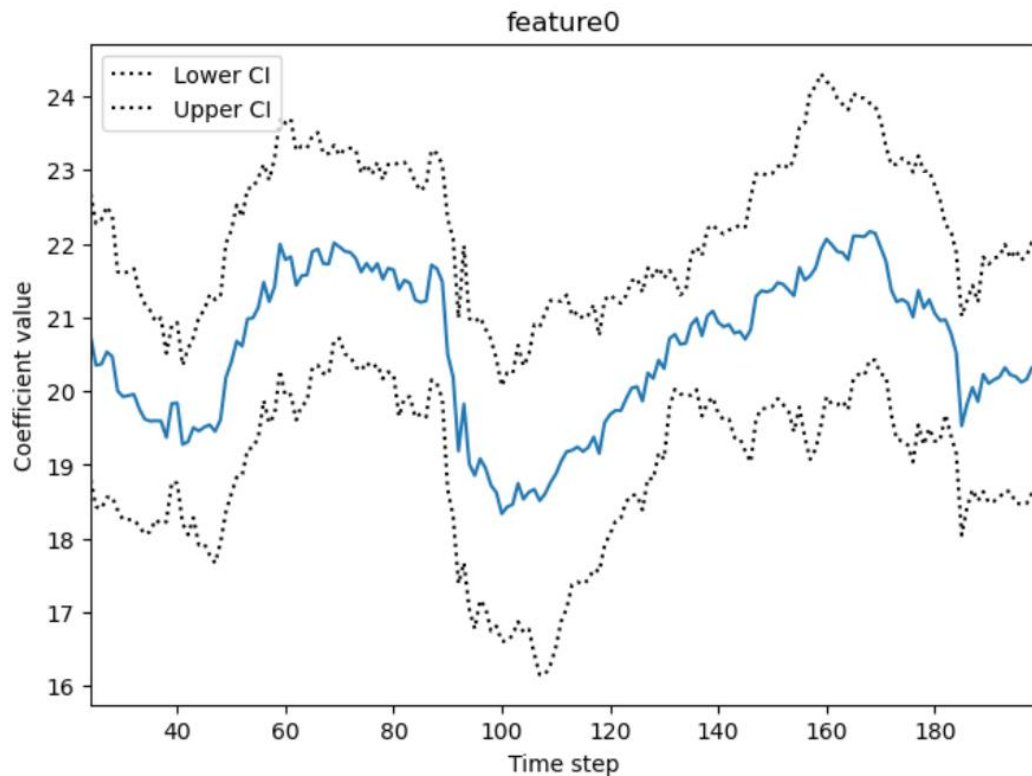
We can see the parameters using model.params. Here are the params for time steps 20 to 30:

```
model.params[20:30]
```

	feature0	feature1
20	NaN	NaN
21	NaN	NaN
22	NaN	NaN
23	NaN	NaN
24	20.736214	35.287604
25	20.351719	35.173493
26	20.368027	35.095621
27	20.532655	34.919468
28	20.470171	35.365235
29	20.002261	35.666997

Note that there aren't parameters for entries between 0 and 23 because our window is 25 steps wide. We can easily look at how any of the coefficients are changing over time. Here's an example for 'feature0'.

```
fig = model.plot_recursive_coefficient(variables=['feature0'])
plt.xlabel('Time step')
plt.ylabel('Coefficient value')
plt.show()
```



Recursive ordinary least squares (aka expanding window rolling regression):

A rolling regression with an expanding (rather than moving) window is effectively a recursive least squares model. We can perform this kind of estimation using the RecursiveLS function from statsmodels. Let's fit this to the whole dataset:

```
[20]: reg_rls = sm.RecursiveLS.from_formula(
      'target ~ feature0 + feature1 -1', df)
      model_rls = reg_rls.fit()
      print(model_rls.summary())
```

```

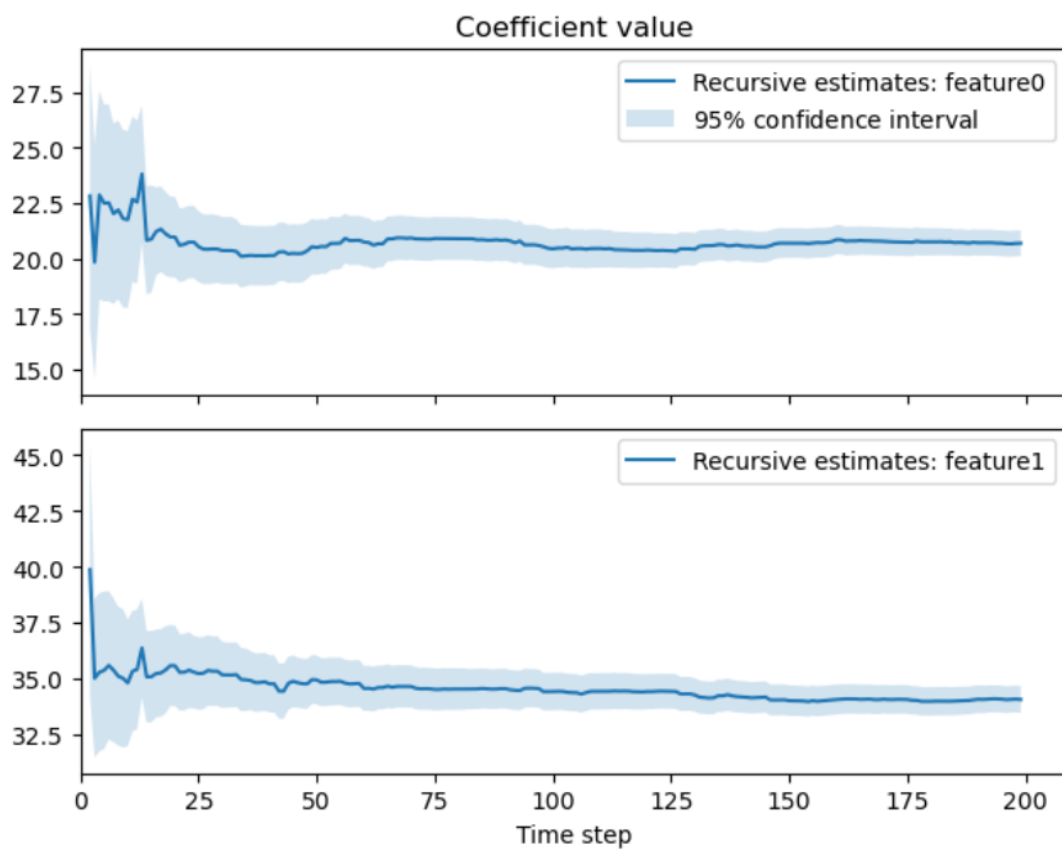
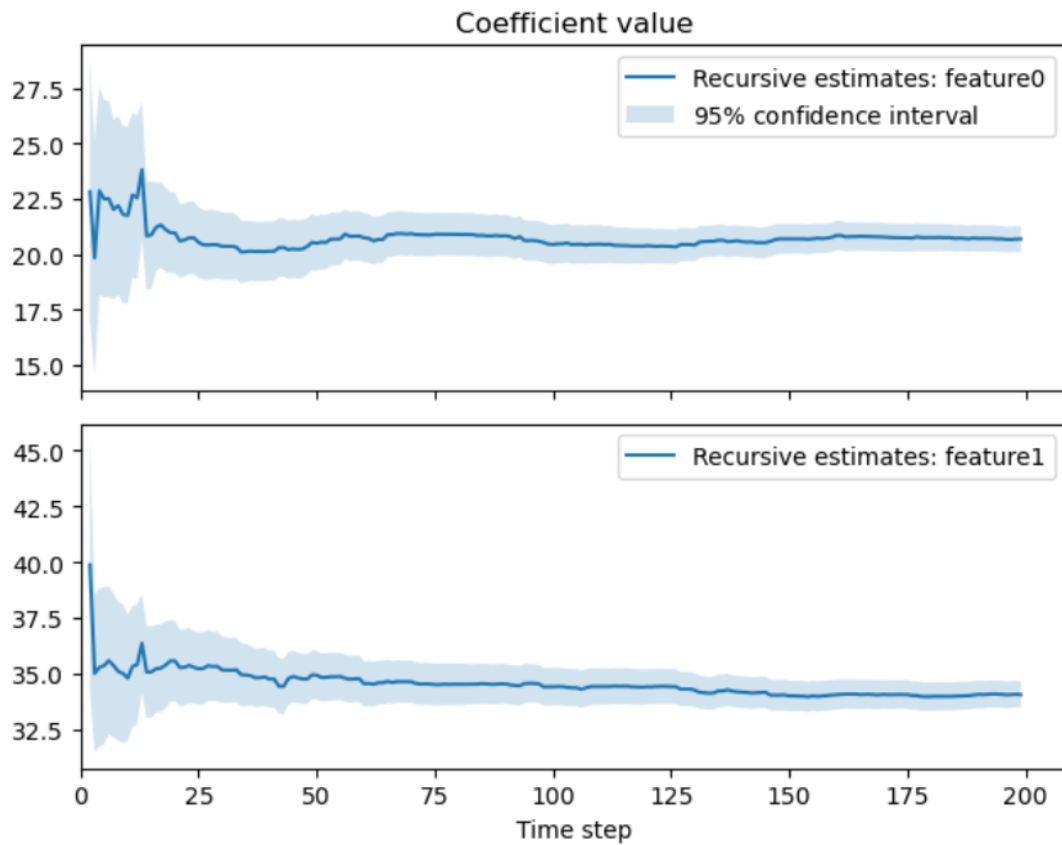
=====
Statespace Model Results
=====
Dep. Variable:          target    No. Observations:          200
Model:                RecursiveLS  Log Likelihood          -570.923
Date:                 Sat, 20 Jul 2024  R-squared:              0.988
Time:                 16:52:29    AIC                   1145.847
Sample:              0          BIC                   1152.444
                  - 200    HQIC                   1148.516
Covariance Type:      nonrobust    Scale                   17.413
=====
              coef    std err          z      P>|z|      [0.025    0.975]
-----
feature0      20.6872     0.296     69.927     0.000     20.107     21.267
feature1      34.0655     0.302    112.870     0.000     33.474     34.657
=====
Ljung-Box (L1) (Q):                2.02    Jarque-Bera (JB):                3.93
Prob(Q):                          0.16    Prob(JB):                  0.14
Heteroskedasticity (H):            1.17    Skew:                     -0.31
Prob(H) (two-sided):              0.51    Kurtosis:                  3.31
=====
```

Warnings:

[1] Parameters and covariance matrix estimates are RLS estimates conditional on the entire sample.

But now we can look back at how the values of the coefficients changed in real time:

```
fig = model_rls.plot_recursive_coefficient(range(reg_rls.k_exog), legend_loc='upper right')
ax_list = fig.axes
for ax in ax_list:
    ax.set_xlim(0, None)
ax_list[-1].set_xlabel('Time step')
ax_list[0].set_title('Coefficient value');
```

Background- K-Means Clustering:

(Note: Check references for the sources used)

K-means clustering is one of the simplest and popular unsupervised machine learning algorithms. Typically, unsupervised algorithms make inferences from datasets using only input vectors without referring to known, or labelled, outcomes.

A cluster refers to a collection of data points aggregated together because of certain similarities. You'll define a target number k , which refers to the number of centroids you need in the dataset. A centroid is the imaginary or real location representing the center of the cluster.

Every data point is allocated to each of the clusters through reducing the in-cluster sum of squares. In other words, the K-means algorithm identifies k number of centroids, and then allocates every data point to the nearest cluster, while keeping the variance of individual entries from centroids as small as possible. The 'means' in the K-means refers to averaging of the data; that is, finding the centroid.

Working:

To process the learning data, the K-means algorithm in data mining starts with a first group of randomly selected centroids, which are used as the beginning points for every cluster, and then performs iterative (repetitive) calculations to optimize the positions of the centroids

It halts creating and optimizing clusters when either:

- (i) The centroids have stabilized — there is no change in their values because the clustering has been successful, or,
- (ii) The defined number of iterations has been achieved.

An Example:

Let's see the steps how the K-means machine learning algorithm works using the Python programming language.

We'll use the Scikit-learn library and some random data to illustrate a K-means clustering simple explanation

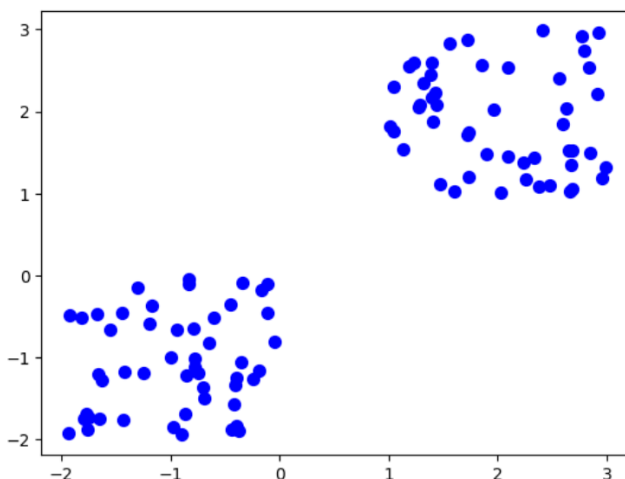
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
%matplotlib inline
```

As you can see from the above code, we'll import the following libraries in our project: Pandas for reading and writing spreadsheets, Numpy for carrying out efficient computations, Matplotlib for visualization of data.

Now we proceed to generate random data:

```
X=-2*np.random.rand(100,2)
X1=1+2*np.random.rand(50,2)
X[50:100, : ]=X1
plt.scatter(X[:,0],X[:,1],s=50,c='b')
plt.show()
```

A total of 100 data points has been generated and divided into two groups, of 50 points each. Here is how the data is displayed on a two-dimensional space:



Now, we'll use some of the available functions in the Scikit-learn library to process the randomly generated data. In this case, we arbitrarily gave k (`n_clusters`) an arbitrary value of two.

```
Kmean=KMeans(n_clusters=2,n_init='auto')
Kmean.fit(X)
```

▼

```
KMeans(n_clusters=2, n_init='auto')
```

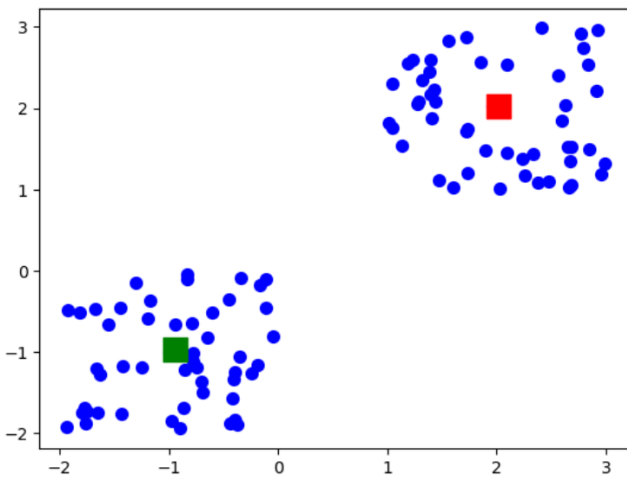
At this stage, we can check the spatial location of the centroid of the clusters:

```
Kmean.cluster_centers_
```

```
array([[ 2.01388777,  1.90650922],
       [-0.95041339, -1.06047671]])
```

Let's display the cluster centroids (using green and red colors):

```
plt.scatter(X[ : , 0], X[ : , 1], s =50, c='b')
plt.scatter(-0.94665068, -0.97138368, s=200, c='g', marker='s')
plt.scatter(2.01559419, 2.02597093, s=200, c='r', marker='s')
plt.show()
```



With everything in place, now is the time to test the algorithm:

Let's fetch the labels property of the K-means clustering example dataset; that is, how the data points are categorized into the two clusters.

Kmean.labels

```
array([1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

As you can see above, 50 data points belong to cluster-1 while the rest belong to cluster-0.

Let's use the code below for predicting the cluster of a data point:

```
sample_test=np.array([-3.0,-3.0])  
second_test=sample_test.reshape(1, -1)  
Kmean.predict(second_test)
```

```
array([1])
```

It shows that the test data point belongs to the (green centroid) cluster-1.

K-means clustering is an extensively used technique for data cluster analysis. It is easy to understand, especially if you accelerate your learning using a K-means clustering tutorial. Furthermore, it delivers training results quickly.

However, its performance is usually not as competitive as those of the other sophisticated clustering techniques because slight variations in the data could lead to high variance. Furthermore, clusters are assumed to be spherical and evenly sized, something which may reduce the accuracy of the K-means clustering Python results.

PUTTING OUR KNOWLEDGE TO USE:

Now we finally make strategies utilizing machine learning principles.

Here are the brief outlines of what we do. This section shall provide a smooth transition to a detailed description of our work.

First, we begin describing the intersectionality between machine learning and finance/trading.

Few Use cases of Machine Learning in Trading:

Supervised Learning:

- Signal Generation through prediction. For example, buy or sell signals, based on predicted return or direction.
- Risk management through prediction. For example, determining position sizing and stop-loss levels to have more optimized risk.

Unsupervised Learning:

- Extract insights from the data. For example: discover patterns, relationships and structures.

Challenges:

- Main problem is reflexivity. (The phenomenon of others happening to use the same strategy as yours, diluting the optimality of your strategy.)
- Unpredictable to predict factors like non-farm payrolls and weekly joblessness claims.
- Very hard to predict returns or prices.
- Quite hard to predict return signals.
- Hard to predict technical indicators.
- Volatility is not so hard to predict compared to returns. Volatility is easy to predict because it mean reverts, meaning it goes back to its average while price is random. This however leads to arbitrage and market manipulation which may challenge the learning model.
- Furthermore, some technical challenges are overfitting and generalization, non-stationarity and regime shifts, as well as interpretation of black-box models.

(1) Unsupervised Learning Trading Strategy

Flow:

- I. Download/Load SP500 stocks prices data.
- II. Calculate different features and indicators on each stock.

- III. Aggregate on monthly level and filter top 150 most liquid stocks.
- IV. Calculate Monthly Returns for different time-horizons.
- V. Download Fama-French Factors and Calculate Rolling Factor Betas.
- VI. For each month fit a K-Means Clustering Algorithm to group similar assets based on their features.
- VII. For each month select assets based on the cluster and form a portfolio based on Efficient Frontier max sharpe ratio optimization.
- VIII. Visualize Portfolio returns and compare to SP500 returns.

Limitation: We are going to use the most recent SP500 Stocks list , which means that there may be a survivorship bias in the list.

(2) Twitter Sentiment Investing Strategy

This approach focuses on analyzing how people feel about certain stocks, industries, or the overall market. It assumes that public sentiment can impact stock prices and markets. For example, if many people are positive about a particular company on Twitter, it might indicate that potential for that company's stock to perform well.

Flow:

- I. Load NASDAQ stocks twitter sentiment data.
- II. Calculate a quantitative feature of the engagement ratio in Twitter of each stock.
- III. Rank all stocks every month and construct an equal-weight portfolio.(An equal-weight portfolio is constructed to ensure diversification, reduce the impact of any single asset's performance on the overall portfolio, and to potentially enhance returns by avoiding overconcentration in larger or more popular assets.)
- IV. Compare it against NASDAQ Performance.

(3) Intraday Strategy using GARCH Model

This approach involves buying and selling financial assets within the same trading day to profit for short-term price movements. Intraday traders use technical analysis, real-time data, and risk management techniques to make quick decisions, aiming to capitalize on market volatility.

Flow:

- I. Load simulated daily data and simulated 5-minute data.
- II. Define function to fit GARCH model and predicted volatility 1-day ahead in a rolling window.
- III. Calculate prediction premium and form a daily signal on it.

- IV. Merge with intraday data and calculate intraday indicators to form the intraday signal.
- V. Generate the position entry and hold until the end of the day.
- VI. Calculate final strategy returns.

(1)Unsupervised Learning Trading Strategy:

Various libraries are imported, including *RollingOLS* from *statsmodels* for rolling regression, *web* from *pandas_datareader* for data fetching, *matplotlib.pyplot* for plotting, *statsmodels.api*, *pandas*, *numpy*, *datetime* for date manipulation, *yfinance* for financial data, *pandas_ta* for technical analysis, *warnings* and to suppress warnings.

1. Download/Load SP500 stocks prices data.

```
165]: from statsmodels.regression.rolling import RollingOLS
import pandas_datareader.data as web
import matplotlib.pyplot as plt
import statsmodels.api as sm
import pandas as pd
import numpy as np
import datetime as dt
import yfinance as yf
import pandas_ta
import warnings
warnings.filterwarnings('ignore')

sp500=pd.read_html('https://en.wikipedia.org/wiki/List_of_S%26P_500_companies')[0]
```

The S&P 500 company list is fetched from Wikipedia using `pd.read_html`. The 'Symbol' column in the dataframe is cleaned to replace periods ('.') with hyphens ('-') to avoid errors with *yfinance*.

```
sp500=pd.read_html('https://en.wikipedia.org/wiki/List_of_S%26P_500_companies')[0]

#Do up some cleaning, '.' in the symbol name gives an error with yfinance, so replace '.' with '-'
#Note the limitation that this list of stocks is not free of survivorship bias
sp500['Symbol']=sp500['Symbol'].str.replace('.', '-')
```

A unique list of stock symbols from the S&P 500 is created.

```
symbols_list=sp500['Symbol'].unique().tolist()
```

The end date for data fetching is set to '2023-09-27'. The start date is calculated to be 8 years before the end date using `pd.DateOffset`

Now, the code which follows calculates various technical indicators (Garman-Klass volatility, RSI, Bollinger Bands, ATR, MACD) and dollar volume for each stock in the S&P 500, normalizes the indicators where necessary, and stores them in new columns within the dataframe `df`. These indicators are commonly used in financial analysis and machine learning models for trading strategies.

2. Calculate features and technical indicators for each stock.

- Garman-Klass Volatility
- RSI
- Bollinger Bands
- ATR
- MACD
- Dollar Volume

$$\text{Garman-Klass Volatility} = \frac{(\ln(\text{High}) - \ln(\text{Low}))^2}{2} - (2\ln(2) - 1)(\ln(\text{Adj Close}) - \ln(\text{Open}))^2$$

This line calculates the Garman-Klass volatility estimator, a measure of price volatility, and stores it in a new column `garman_klass_vol`.

```
df['garman_klass_vol'] = ((np.log(df['high'])-np.log(df['low']))**2)/2-(2*np.log(2)-1)*((np.log(df['adj_close'])-np.log(df['open']))**2)
```

This calculates the 20-period RSI for each stock and adds it as a new column `rsi`.

```
df['rsi'] = df.groupby(level=1)['adj_close'].transform(lambda x: pandas_ta.rsi(close=x, length=20))
```

These lines calculate the lower, middle, and upper Bollinger Bands for each stock using the log-transformed adjusted closing price over a 20-period window and store them in `'bb_low'`, `'bb_mid'`, and `'bb_high'`, respectively.

```
#This function would supply all 3 bands at once
# pandas_ta.bbands(close=df.xs('AAPL', level=1)['adj_close'], length=20)

df['bb_low'] = df.groupby(level=1)['adj_close'].transform(lambda x: pandas_ta.bbands(close=np.log1p(x), length=20).iloc[:,0])
df['bb_mid'] = df.groupby(level=1)['adj_close'].transform(lambda x: pandas_ta.bbands(close=np.log1p(x), length=20).iloc[:,1])
df['bb_high'] = df.groupby(level=1)['adj_close'].transform(lambda x: pandas_ta.bbands(close=np.log1p(x), length=20).iloc[:,2])

#Note that we normalize data (except RSI) since we are going to use it in a ML model
```

This function computes the 14-period ATR for each stock and normalizes it by subtracting the mean and dividing by the standard deviation. The normalized ATR is stored in a new column `'atr'`.

```
def compute_atr(stock_data):
    atr = pandas_ta.atr(high=stock_data['high'],
                        low=stock_data['low'],
                        close=stock_data['close'],
                        length=14)
    return atr.sub(atr.mean()).div(atr.std())    #normalize data

df['atr'] = df.groupby(level=1, group_keys=False).apply(compute_atr)
```

This function computes the MACD for each stock's adjusted closing price, normalizes it, and stores it in a new column `'macd'`.

```
def compute_macd(close):
    macd = pandas_ta.macd(close=close, length=20).iloc[:,0]
    return macd.sub(macd.mean()).div(macd.std())    #normalize data

df['macd'] = df.groupby(level=1, group_keys=False)['adj close'].apply(compute_macd)
```

This calculates the dollar volume of trading by multiplying the adjusted closing price by the trading volume, then dividing by one million to convert to millions of dollars, and stores it in a new column 'dollar_volume'.

```
df['dollar_volume'] = (df['adj close']*df['volume'])/1e6    #divided by a million since millions of shares are traded each day and it makes sense to
                                                         #do this to get nice numbers
```

Finally, data frame 'df' is displayed to perform checks.

(Note that all columns are not visible in this image and there is a horizontal scrollbar at the bottom)

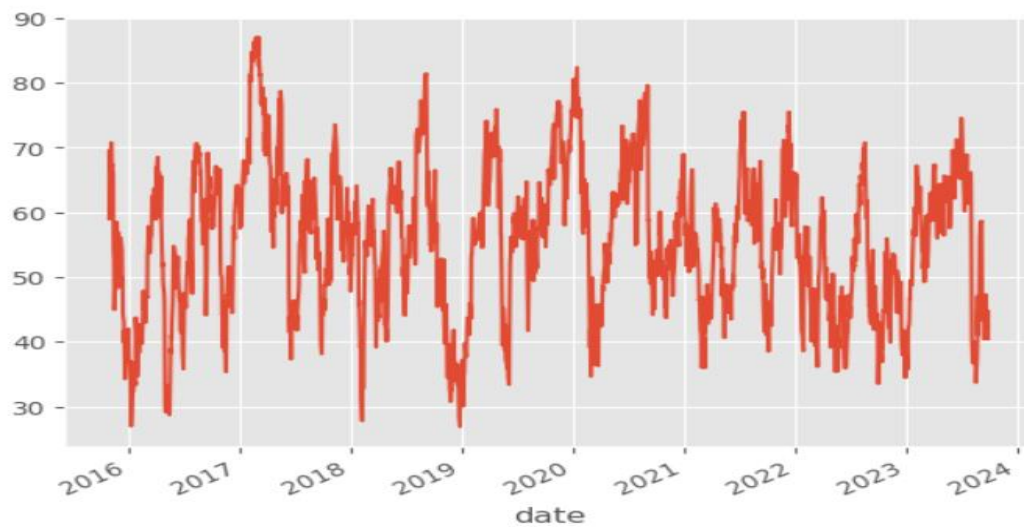
OT		Price	adj close	close	high	low	open	volume	garman_klass_vol	rsi	bb_low	bb_mid	bb_high	atr	macd
date	ticker														
2015-09-29	A	31.425232	33.740002	34.060001	33.240002	33.360001	2252400.0	-0.001082	NaN	NaN	NaN	NaN	NaN	NaN	NaN
	AAL	37.361629	39.180000	39.770000	38.790001	39.049999	7478800.0	-0.000443	NaN	NaN	NaN	NaN	NaN	NaN	NaN
	AAPL	24.651136	27.264999	28.377501	26.965000	28.207500	293461600.0	-0.005712	NaN	NaN	NaN	NaN	NaN	NaN	NaN
	ABBV	36.004154	52.790001	54.189999	51.880001	53.099998	12842800.0	-0.057368	NaN	NaN	NaN	NaN	NaN	NaN	NaN
	ABT	33.302032	39.500000	40.150002	39.029999	39.259998	12287500.0	-0.010064	NaN	NaN	NaN	NaN	NaN	NaN	NaN
...
2023-09-26	XYL	88.736298	89.519997	90.849998	89.500000	90.379997	1322400.0	-0.000018	26.146729	4.485761	4.567684	4.649607	0.033800	-2.159189	
	YUM	122.211006	124.010002	124.739998	123.449997	124.239998	1500600.0	-0.000051	36.057170	4.811707	4.841672	4.871637	0.142547	-1.363696	
	ZBH	111.534821	112.459999	117.110001	112.419998	116.769997	3610500.0	0.000022	31.893229	4.745884	4.785551	4.825217	-0.381708	-0.881067	
	ZBRA	223.960007	223.960007	226.649994	222.580002	225.970001	355400.0	0.000133	29.494977	5.400991	5.539167	5.677342	-0.057389	-1.600791	
	ZTS	175.131119	176.869995	178.449997	176.270004	176.580002	1463200.0	0.000049	42.623459	5.153746	5.212559	5.271371	0.651515	-1.188278	

985955 rows x 14 columns

We check the parameters for stocks of Apple Inc. to ensure consistency.

```
[169]: #Checkers
df.xs('AAPL', level=1)['rsi'].plot()
```

```
[169]: <Axes: xlabel='date'>
```



3. Aggregate to monthly level and filter top 150 most liquid stocks for each month.

- To reduce training time and experiment with features and strategies, we convert the business-daily data to month-end frequency.

```
[290]: last_cols = [c for c in df.columns.unique(0) if c not in ['dollar_volume', 'volume', 'open',
                                                             'high', 'low', 'close']]

data = (pd.concat([df.unstack('ticker')['dollar_volume'].resample('M').mean().stack('ticker').to_frame('dollar_volume'),
                  df.unstack()[last_cols].resample('M').last().stack('ticker')],
                 axis=1)).dropna()
```

The above code segment Creates a list 'last_cols' of column names from df, excluding 'dollar_volume', 'volume', 'open', 'high', 'low', and 'close'. This list will include the technical indicators and other relevant features.

It then unstacks df by 'ticker', resamples the 'dollar_volume' column to a monthly frequency using the mean, restacks it by 'ticker', and converts it to a dataframe with a single column 'dollar_volume'.

Unstacks df, resamples the columns in last_cols to a monthly frequency using the last value of each month, and restacks by 'ticker'.

Unstacks df, resamples the columns in last_cols to a monthly frequency using the last value of each month, and restacks by 'ticker'.

The result is a cleaner dataset suitable for further analysis or modeling, with a focus on monthly time periods.

This is the resultant 'data' obtained as a result:

data										
[290]:										
		dollar_volume	adj close	garman_klass_vol	rsi	bb_low	bb_mid	bb_high	atr	macd
date	ticker									
2015-11-30	A	135.740907	38.950901	-0.002098	73.421446	3.544191	3.616636	3.689081	-1.033887	0.567158
	AAL	287.915810	39.429939	-0.000966	40.719013	3.672028	3.749832	3.827636	0.190822	-0.418770
	AAPL	4023.983928	26.854136	-0.003307	55.537341	3.281679	3.324991	3.368302	-0.967900	-0.142789
	ABBV	334.491202	40.025845	-0.062372	49.376883	3.717771	3.766009	3.814247	-0.526809	0.145677
	ABT	210.542072	38.091488	-0.011927	56.962578	3.650901	3.672752	3.694603	-1.064842	0.335557
...
2023-09-30	OTIS	154.361755	78.356499	-0.000097	33.116229	4.370137	4.415425	4.460712	-1.028320	-1.534536
	ABNB	1633.500725	132.279999	0.000213	44.494127	4.857047	4.940924	5.024801	-1.006939	-0.037854
	CEG	196.670368	107.862022	0.000131	55.245466	4.652147	4.692320	4.732493	-0.436215	0.366876
	GEHC	212.275853	66.130211	0.000185	40.922300	4.155436	4.212972	4.270508	-0.893478	-1.116463
	KVUE	670.804280	20.006195	-0.000159	35.706336	3.014275	3.089221	3.164168	-0.899746	-1.435620

46553 rows × 9 columns

The code below calculates a 5-year rolling average of dollar volume for each stock, ranks the stocks by this rolling average within each month, filters the dataframe to keep only the top 150 stocks by dollar volume, and drops the intermediate columns used for this filtering.

This line below unstacks data by 'ticker' to have a wide format, applies a rolling window of 5 years (60 months, as 5*12) to compute the mean of the dollar volume for each stock, requiring at least 12 months of data to compute the mean, then restacks the dataframe by 'ticker' to return to a multi-index format. Finally, it stores the result back in the data dataframe under the 'dollar_volume' column.

```
data['dollar_volume'] = (data.loc[:, 'dollar_volume'].unstack('ticker').rolling(5*12, min_periods=12).mean().stack())

data['dollar_vol_rank'] = (data.groupby('date')['dollar_volume'].rank(ascending=False))
```

In the above lines of code Groups data by 'date'.

Ranks the stocks within each date group by their dollar volume in descending order (highest volume gets the lowest rank number).

Stores these ranks in a new column 'dollar_vol_rank'.

Now, Filters the dataframe to include only rows where the 'dollar_vol_rank' is less than 150, effectively keeping only the top 150 stocks by dollar volume for each date.

Drops the 'dollar_volume' and 'dollar_vol_rank' columns as they are no longer needed.

```
data = data[data['dollar_vol_rank'] < 150].drop(['dollar_volume', 'dollar_vol_rank'], axis=1)
```

This is how 'data' looks after the above operations:

data									
[215]:		adj close	garman_klass_vol	rsi	bb_low	bb_mid	bb_high	atr	macd
date	ticker								
2016-10-31	AAL	39.134327	-0.000176	62.203539	3.604673	3.655493	3.706314	0.402199	1.131595
	AAPL	26.212475	-0.002468	49.891093	3.294237	3.323117	3.351997	-1.038688	-0.195978
	ABBV	39.878780	-0.049190	27.477797	3.744517	3.798670	3.852823	-0.893132	-0.760593
	ABT	34.112476	-0.008074	38.008736	3.549492	3.599959	3.650426	-1.035224	-0.650888
	ACN	103.117416	-0.005023	53.823725	4.633009	4.644646	4.656283	-0.996806	-0.135456
...
2023-09-30	XOM	113.372101	-0.000065	59.440186	4.687091	4.727187	4.767283	0.601335	1.400623
	MRNA	98.120003	0.000146	38.747314	4.582514	4.685332	4.788149	-0.529511	-0.376899
	UBER	44.270000	0.000441	45.005268	3.806654	3.862227	3.917801	-0.746098	-0.133973
	CRWD	160.479996	0.000144	51.534803	5.026187	5.103696	5.181204	-0.744862	0.245950
	ABNB	132.279999	0.000213	44.494127	4.857047	4.940924	5.024801	-1.006939	-0.037854

12516 rows × 8 columns

- Calculate 5-year rolling average of dollar volume for each stocks before filtering.

```
]: data['dollar_volume'] = (data.loc[:, 'dollar_volume'].unstack('ticker').rolling(5*12, min_periods=12).mean().stack())
data['dollar_vol_rank'] = (data.groupby('date')['dollar_volume'].rank(ascending=False))
data = data[data['dollar_vol_rank']<150].drop(['dollar_volume', 'dollar_vol_rank'], axis=1)
```

data									
[215]:		adj close	garman_klass_vol	rsi	bb_low	bb_mid	bb_high	atr	macd
date	ticker								
2016-10-31	AAL	39.134327	-0.000176	62.203539	3.604673	3.655493	3.706314	0.402199	1.131595
	AAPL	26.212475	-0.002468	49.891093	3.294237	3.323117	3.351997	-1.038688	-0.195978
	ABBV	39.878780	-0.049190	27.477797	3.744517	3.798670	3.852823	-0.893132	-0.760593
	ABT	34.112476	-0.008074	38.008736	3.549492	3.599959	3.650426	-1.035224	-0.650888
	ACN	103.117416	-0.005023	53.823725	4.633009	4.644646	4.656283	-0.996806	-0.135456
...
2023-09-30	XOM	113.372101	-0.000065	59.440186	4.687091	4.727187	4.767283	0.601335	1.400623
	MRNA	98.120003	0.000146	38.747314	4.582514	4.685332	4.788149	-0.529511	-0.376899
	UBER	44.270000	0.000441	45.005268	3.806654	3.862227	3.917801	-0.746098	-0.133973
	CRWD	160.479996	0.000144	51.534803	5.026187	5.103696	5.181204	-0.744862	0.245950
	ABNB	132.279999	0.000213	44.494127	4.857047	4.940924	5.024801	-1.006939	-0.037854

12516 rows × 8 columns

The function `calculate_returns` computes the returns over different time lags for a given DataFrame of adjusted close prices (`adj close`), while also handling

outliers. 'outlier_cutoff = 0.005' is used to set the cutoff for the outlier handling to the 0.5th percentile.

lags = [1, 2, 3, 6, 9, 12] is a list of time lags (in months) over which the returns will be calculated. We loop through each lag and calculate corresponding returns.

df[f'return_{lag}m'] creates a new column in the DataFrame to store the returns for the current lag.

.pct_change(lag) calculates the percentage change over the specified lag.

.pipe(lambda x: x.clip(lower=x.quantile(outlier_cutoff), upper=x.quantile(1-outlier_cutoff))) clips the percentage changes to remove outliers by keeping only the values between the 0.5th and 99.5th percentiles.

.add(1) adds 1 to the clipped percentage changes.

.pow(1/lag) takes the root of the adjusted percentage change to annualize it over the specified lag.

.sub(1) subtracts 1 to get the final adjusted return for the lag. Finally, the updated data frame is returned.

Thus, this function ensures that the calculated returns for different lags are adjusted for outliers, providing more robust return estimates.

▼ 4. Calculate Monthly Returns for different time horizons as features.

- To capture time series dynamics that reflect, for example, momentum patterns, we compute historical returns using the method .pct_change(lag), that is, returns over various monthly periods as identified by lags.

```
[217]: def calculate_returns(df):  
  
    outlier_cutoff = 0.005  
  
    lags = [1, 2, 3, 6, 9, 12]  
  
    for lag in lags:  
  
        df[f'return_{lag}m'] = (df['adj_close']  
                                .pct_change(lag)  
                                .pipe(lambda x: x.clip(lower=x.quantile(outlier_cutoff),  
                                                         upper=x.quantile(1-outlier_cutoff)))  
                                .add(1)  
                                .pow(1/lag)  
                                .sub(1))  
  
    return df
```

Let's check 'data' at this point:

```
data = data.groupby(level=1, group_keys=False).apply(calculate_returns).dropna()
data
```

[217]:

		adj close	garman_klass_vol	rsi	bb_low	bb_mid	bb_high	atr	macd	return_1m	return_2m	return_3m	return_6m	return_9m	re
date	ticker														
2017-10-31	AAL	45.534168	-0.000363	41.051782	3.849110	3.921750	3.994389	1.011062	-0.018698	-0.014108	0.022981	-0.023860	0.016495	0.007008	
	AAPL	39.713894	-0.001055	69.196613	3.594730	3.641603	3.688475	-0.906642	-0.039275	0.096808	0.015249	0.044955	0.028875	0.038941	
	ABBV	66.876801	-0.036142	55.247871	4.187696	4.234050	4.280405	0.375557	0.473815	0.022728	0.098590	0.091379	0.056495	0.047273	
	ABT	48.237461	-0.005677	53.844890	3.887384	3.910952	3.934519	-1.040044	0.276133	0.021276	0.034308	0.034801	0.038672	0.031320	
	ACN	128.834717	-0.004274	69.365295	4.798335	4.838013	4.877691	-0.986514	0.352343	0.064180	0.048455	0.037203	0.028692	0.027398	
...
2023-09-30	XOM	113.372101	-0.000065	59.440186	4.687091	4.727187	4.767283	0.601335	1.400623	0.046947	0.046139	0.030496	0.012838	0.008747	
	MRNA	98.120003	0.000146	38.747314	4.582514	4.685332	4.788149	-0.529511	-0.376899	-0.132219	-0.086803	-0.068763	-0.071952	-0.064976	
	UBER	44.270000	0.000441	45.005268	3.806654	3.862227	3.917801	-0.746098	-0.133973	-0.062672	-0.053920	0.008422	0.057244	0.066838	
	CRWD	160.479996	0.000144	51.534803	5.026187	5.103696	5.181204	-0.744862	0.245950	-0.015641	-0.003656	0.029981	0.026391	0.047942	
	ABNB	132.279999	0.000213	44.494127	4.857047	4.940924	5.024801	-1.006939	-0.037854	0.005549	-0.067704	0.010603	0.010289	0.049124	

10343 rows × 14 columns

This code snippet retrieves and processes financial factor data from the Fama-French dataset and combines it with monthly return data.

`factor_data = web.DataReader('F-F_Research_Data_5_Factors_2x3', 'famafrench', start='2010')[0].drop('RF', axis=1)` is used to retrieve the Fama-French data.

`web.DataReader('F-F_Research_Data_5_Factors_2x3', 'famafrench', start='2010')` retrieves the data: This line uses the `DataReader` function from `pandas_datareader` to fetch the Fama-French 5-factor data starting from the year 2010.

`[0]` accesses the first DataFrame in the retrieved data.

`.drop('RF', axis=1)` removes the 'RF' (risk-free rate) column from the DataFrame.

`factor_data.index = factor_data.index.to_timestamp()` : is used to convert index to timestamps.

Converts the index of the DataFrame to timestamps, which changes the frequency from daily to monthly.

`factor_data = factor_data.resample('M').last().div(100)` is used to resample the data to a monthly frequency, taking the last value of each month (`resample('M').last()`) and then divides by 100 to convert into percentage.

`factor_data.index.name = 'date'` renames the index to 'date' for clarity.

`factor_data = factor_data.join(data['return_1m']).sort_index()` is used to combine with return data.

Joins the `factor_data` DataFrame with the monthly return data (`data['return_1m']`), aligning them by the 'date' index.

Sorts the combined DataFrame by the index (`sort_index()`).

Overall, this code fetches and processes Fama-French factor data, converting it to a monthly frequency, normalizing it, and then merging it with existing monthly return data.

5. Download Fama-French Factors and Calculate Rolling Factor Betas.

- We will introduce the Fama—French data to estimate the exposure of assets to common risk factors using linear regression.
- The five Fama—French factors, namely market risk, size, value, operating profitability, and investment have been shown empirically to explain asset returns and are commonly used to assess the risk/return profile of portfolios. Hence, it is natural to include past factor exposures as financial features in models.
- We can access the historical factor returns using the pandas-datareader and estimate historical exposures using the RollingOLS rolling linear regression.

```
[219]: factor_data = web.DataReader('F-F_Research_Data_5_Factors_2x3',
                                   'famafrench',
                                   start='2010')[0].drop('RF', axis=1)

factor_data.index = factor_data.index.to_timestamp()

factor_data = factor_data.resample('M').last().div(100)

factor_data.index.name = 'date'

factor_data = factor_data.join(data['return_1m']).sort_index()
```

Here is a display of factor_data at the end of this:

factor_data

[219]:

		Mkt-RF	SMB	HML	RMW	CMA	return_1m
date ticker							
2017-10-31	AAL	0.0225	-0.0194	0.0020	0.0093	-0.0325	-0.014108
	AAPL	0.0225	-0.0194	0.0020	0.0093	-0.0325	0.096808
	ABBV	0.0225	-0.0194	0.0020	0.0093	-0.0325	0.022728
	ABT	0.0225	-0.0194	0.0020	0.0093	-0.0325	0.021276
	ACN	0.0225	-0.0194	0.0020	0.0093	-0.0325	0.064180
...
2023-09-30	VRTX	-0.0524	-0.0181	0.0151	0.0187	-0.0082	0.009617
	VZ	-0.0524	-0.0181	0.0151	0.0187	-0.0082	-0.056890
	WFC	-0.0524	-0.0181	0.0151	0.0187	-0.0082	-0.015500
	WMT	-0.0524	-0.0181	0.0151	0.0187	-0.0082	-0.000676
	XOM	-0.0524	-0.0181	0.0151	0.0187	-0.0082	0.046947

10343 rows × 6 columns

We perform checks for Apple and Microsoft at this stage to capture what is happening.

```
[221]: #checker

factor_data.xs('AAPL', level=1).head()
```

```
[221]:
```

	Mkt-RF	SMB	HML	RMW	CMA	return_1m
date						
2017-10-31	0.0225	-0.0194	0.0020	0.0093	-0.0325	0.096808
2017-11-30	0.0312	-0.0033	-0.0003	0.0316	-0.0005	0.020278
2017-12-31	0.0106	-0.0107	0.0006	0.0074	0.0169	-0.015246
2018-01-31	0.0557	-0.0318	-0.0129	-0.0076	-0.0096	-0.010636
2018-02-28	-0.0365	0.0032	-0.0104	0.0052	-0.0237	0.068185

```
[223]: #checker

factor_data.xs('MSFT', level=1).head()
```

```
[223]:
```

	Mkt-RF	SMB	HML	RMW	CMA	return_1m
date						
2017-10-31	0.0225	-0.0194	0.0020	0.0093	-0.0325	0.116660
2017-11-30	0.0312	-0.0033	-0.0003	0.0316	-0.0005	0.016984
2017-12-31	0.0106	-0.0107	0.0006	0.0074	0.0169	0.016277
2018-01-31	0.0557	-0.0318	-0.0129	-0.0076	-0.0096	0.110708
2018-02-28	-0.0365	0.0032	-0.0104	0.0052	-0.0237	-0.008415

This code snippet filters the factor_data DataFrame to retain only those stocks that have at least 10 observations.

observations = factor_data.groupby(level=1).size() groups the factor_data by the second level of the index, which is assumed to be the 'ticker' level, and counts the number of observations for each stock (size()).

valid_stocks = observations[observations >= 10] filters the observations Series to include only those stocks that have at least 10 observations.

factor_data =
factor_data[factor_data.index.get_level_values('ticker').isin(valid_stocks.index)] filters the factor_data DataFrame to retain only the rows where the 'ticker' value is in the index of valid_stocks.

This process ensures that the factor_data DataFrame only includes stocks with a sufficient number of observations, enhancing the robustness of any subsequent analysis.

- Filter out stocks with less than 10 months of data.(This is done since we are gonna use a rolling window for 2 years and entries which don't have enough data would break our code)

```
[225]: observations = factor_data.groupby(level=1).size()
valid_stocks = observations[observations >= 10]
factor_data = factor_data[factor_data.index.get_level_values('ticker').isin(valid_stocks.index)]
```

One may check the present state of 'factor_data' at this point.

```
factor_data
```

```
[225]:
```

		Mkt-RF	SMB	HML	RMW	CMA	return_1m
	date ticker						
2017-10-31	AAL	0.0225	-0.0194	0.0020	0.0093	-0.0325	-0.014108
	AAPL	0.0225	-0.0194	0.0020	0.0093	-0.0325	0.096808
	ABBV	0.0225	-0.0194	0.0020	0.0093	-0.0325	0.022728
	ABT	0.0225	-0.0194	0.0020	0.0093	-0.0325	0.021276
	ACN	0.0225	-0.0194	0.0020	0.0093	-0.0325	0.064180
...
2023-09-30	VRTX	-0.0524	-0.0181	0.0151	0.0187	-0.0082	0.009617
	VZ	-0.0524	-0.0181	0.0151	0.0187	-0.0082	-0.056890
	WFC	-0.0524	-0.0181	0.0151	0.0187	-0.0082	-0.015500
	WMT	-0.0524	-0.0181	0.0151	0.0187	-0.0082	-0.000676
	XOM	-0.0524	-0.0181	0.0151	0.0187	-0.0082	0.046947

10313 rows × 6 columns

This code calculates rolling regression betas for each stock in the factor_data DataFrame using a Rolling Ordinary Least Squares (OLS) model.

factor_data.groupby(level=1, group_keys=False) groups the factor_data DataFrame by the second level of the index, which is assumed to be the

'ticker' level. `group_keys=False` ensures the original index is preserved in the result.

The lines of code that follow apply a lambda function to each group (x represents each stock's data).

Endogenous Variable (endog): The dependent variable is the monthly return (`return_1m`).

Exogenous Variables (exog): The independent variables are the factor data, with a constant term added using `sm.add_constant`.

Rolling OLS: `RollingOLS` is used to fit a rolling regression model:

`window=min(24, x.shape[0])`: Sets the rolling window size to the smaller of 24 or the number of observations for the stock.

`min_nobs=len(x.columns)+1`: Ensures the minimum number of observations for a valid regression is at least the number of independent variables plus one.

Fit Model: Fits the rolling OLS model with `params_only=True`, meaning only the parameters (betas) are returned.

Extract Parameters: Extracts the `params` attribute, which contains the regression coefficients (betas).

Drop Constant Term: Drops the constant term ('const') from the betas.

Store Betas:

The resulting DataFrame `betas` contains the rolling betas for each stock, with one row per stock and columns corresponding to the factor betas.

Thus, this process calculates the rolling betas for each stock with respect to the Fama-French factors over a rolling window, excluding the intercept term. The resulting betas DataFrame provides the time-varying sensitivity of each stock's returns to the different factors.

- Calculate Rolling Factor Betas.

```
[227]: betas = (factor_data.groupby(level=1,
                                group_keys=False)
        .apply(lambda x: RollingOLS(endog=x['return_1m'],
                                    exog=sm.add_constant(x.drop('return_1m', axis=1)),
                                    window=min(24, x.shape[0]),
                                    min_nobs=len(x.columns)+1)
        .fit(params_only=True)
        .params
        .drop('const', axis=1)))
```

Here is the 'betas' table after performing the above operations:

betas

[227]:

		Mkt-RF	SMB	HML	RMW	CMA
date	ticker					
2017-10-31	AAL	NaN	NaN	NaN	NaN	NaN
	AAPL	NaN	NaN	NaN	NaN	NaN
	ABBV	NaN	NaN	NaN	NaN	NaN
	ABT	NaN	NaN	NaN	NaN	NaN
	ACN	NaN	NaN	NaN	NaN	NaN
...
2023-09-30	VRTX	0.456835	-0.444629	-0.314191	-0.077989	0.802008
	VZ	0.332722	-0.166039	0.265929	0.311102	0.108623
	WFC	1.120621	0.297483	2.062606	-0.441341	-1.519516
	WMT	0.700774	-0.313572	-0.413679	-0.141574	0.508836
	XOM	0.983098	-1.094429	1.756406	-0.640017	-0.368886

10313 rows × 5 columns

The provided code performs a series of data manipulations to prepare a dataset that includes factor betas for further analysis.

First, the code initializes a list of Fama-French factor names: 'Mkt-RF', 'SMB', 'HML', 'RMW', 'CMA'. These represent the market risk premium, size premium, value premium, profitability premium, and investment premium, respectively.

Next, the code joins the calculated rolling betas with the main data DataFrame. Specifically, it groups the betas DataFrame by the 'ticker' column and shifts the betas forward by one period using `.shift()`. This ensures that the beta values used in the dataset correspond to the previous period's betas, aligning with the typical use of betas as predictors.

After merging the betas into the main data DataFrame, the code addresses any missing values (NaNs) in the factor columns. It does this by grouping the data by 'ticker' and filling NaNs with the mean value of the corresponding factor for that ticker. This step ensures that any missing beta values are replaced with the mean beta for that stock, preventing gaps in the data.

The code then drops the 'adj close' column from the data DataFrame using `.drop('adj close', axis=1)`. This column represents the adjusted close price of the stocks, which is no longer needed for the subsequent analysis.

Finally, the code removes any remaining rows with missing values by using `.dropna()`. This ensures that the final dataset is free of NaNs, which is crucial for reliable statistical analysis or modeling.

In summary, the code prepares the dataset by incorporating rolling factor betas, handling missing values, and cleaning up unnecessary columns, resulting in a well-structured dataset ready for further analysis.

- Join the rolling factors data to the main features dataframe.

```
[229]: factors = ['Mkt-RF', 'SMB', 'HML', 'RMW', 'CMA']

data = data.join(betas.groupby('ticker').shift())

data.loc[:, factors] = data.groupby('ticker', group_keys=False)[factors].apply(lambda x: x.fillna(x.mean()))

data = data.drop('adj close', axis=1)

data = data.dropna()
```

```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
MultiIndex: 10092 entries, (Timestamp('2017-10-31 00:00:00'), 'AAL') to (Timestamp('2023-09-30 00:00:00'), 'CRWD')
Data columns (total 18 columns):
#   Column                Non-Null Count  Dtype
---  -
0   garman_klass_vol      10092 non-null  float64
1   rsi                   10092 non-null  float64
2   bb_low               10092 non-null  float64
3   bb_mid               10092 non-null  float64
4   bb_high              10092 non-null  float64
5   atr                  10092 non-null  float64
6   macd                 10092 non-null  float64
7   return_1m            10092 non-null  float64
8   return_2m            10092 non-null  float64
9   return_3m            10092 non-null  float64
10  return_6m            10092 non-null  float64
11  return_9m            10092 non-null  float64
12  return_12m           10092 non-null  float64
13  Mkt-RF                10092 non-null  float64
14  SMB                   10092 non-null  float64
15  HML                   10092 non-null  float64
16  RMW                   10092 non-null  float64
17  CMA                   10092 non-null  float64
dtypes: float64(18)
memory usage: 1.4+ MB
```

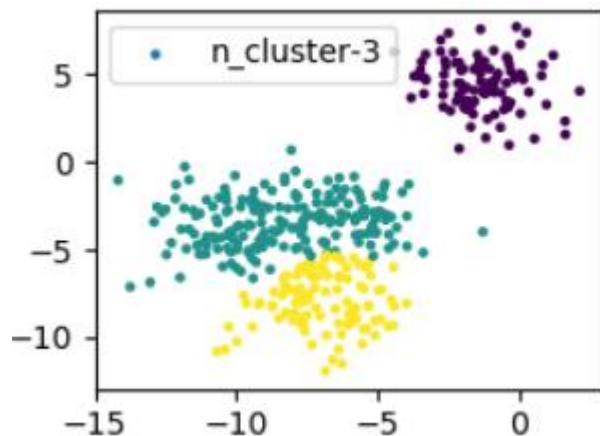
At this point we have to decide on what ML model and approach to use for predictions etc.

- ▼ 6. For each month fit a K-Means Clustering Algorithm to group similar assets based on their features.

K-Means Clustering

- You may want to initialize predefined centroids for each cluster based on your research.
- For visualization purpose we will initially rely on the 'k-means++' initialization.
- Then we will pre-define our centroids for each cluster.

Here is a sample of what a 3-clustered 2-D dataset might look like, the parameter used here is Euclidean distance between the points.



This code applies k-means clustering to the data in a time series context, grouping similar data points into clusters. Here's a step-by-step explanation in paragraphs:

First, the `get_clusters` function is defined. It uses the `KMeans` algorithm from the `sklearn.cluster` module to perform clustering on the given `DataFrame` `df`. The function initializes the `KMeans` model with four clusters (`n_clusters=4`), a fixed random state (`random_state=0`) for reproducibility, and random initialization of centroids (`init='random'`). The `fit` method of the `KMeans` model is then applied to the `DataFrame` `df`, and the resulting cluster labels are assigned to a new column in `df` named 'cluster'. This function returns the modified `DataFrame` with the new cluster labels.

Next, the code prepares the data `DataFrame` for clustering. It ensures that there are no missing values by calling `.dropna()` on the `DataFrame`. This step is crucial as k-means clustering cannot handle NaNs.

The cleaned data `DataFrame` is then grouped by the 'date' column using `.groupby('date', group_keys=False)`. This groups the data by each date, effectively treating each date as a separate subset of the data. The `apply(get_clusters)` method is then used to apply the `get_clusters` function to each subset of data grouped by date. This means that clustering is performed independently for each date, grouping the data points within each date into four clusters.

In summary, the code performs k-means clustering on the data `DataFrame` by date. It defines a function to assign cluster labels to each data point, cleans the data to remove NaNs, groups the data by date, and applies the clustering function to each date-specific group. The result is a `DataFrame` where each data point is assigned to one of four clusters, with clustering performed separately for each date.

```
[320]: from sklearn.cluster import KMeans

def get_clusters(df):
    df['cluster']=KMeans(n_clusters=4,
                        random_state=0,
                        init='random').fit(df).labels_

    return df

data=data.dropna().groupby('date',group_keys=False).apply(get_clusters)
```

```
[328]: df
```

		Price	adj close	close	high	low	open	volume	garman_klass_vol	rsi	bb_low	bb_mid	bb_high	atr	macd
	date	ticker													
2015-09-29		A	31.425232	33.740002	34.060001	33.240002	33.360001	2252400.0	-0.001082	NaN	NaN	NaN	NaN	NaN	NaN
		AAL	37.361629	39.180000	39.770000	38.790001	39.049999	7478800.0	-0.000443	NaN	NaN	NaN	NaN	NaN	NaN
		AAPL	24.651136	27.264999	28.377501	26.965000	28.207500	293461600.0	-0.005712	NaN	NaN	NaN	NaN	NaN	NaN
		ABBV	36.004154	52.790001	54.189999	51.880001	53.099998	12842800.0	-0.057368	NaN	NaN	NaN	NaN	NaN	NaN
		ABT	33.302032	39.500000	40.150002	39.029999	39.259998	12287500.0	-0.010064	NaN	NaN	NaN	NaN	NaN	NaN
...	
2023-09-26		XYL	88.736298	89.519997	90.849998	89.500000	90.379997	1322400.0	-0.000018	26.146729	4.485761	4.567684	4.649607	0.033800	-2.159189
		YUM	122.211006	124.010002	124.739998	123.449997	124.239998	1500600.0	-0.000051	36.057170	4.811707	4.841672	4.871637	0.142547	-1.363696
		ZBH	111.534821	112.459999	117.110001	112.419998	116.769997	3610500.0	0.000022	31.893229	4.745884	4.785551	4.825217	-0.381708	-0.881067
		ZBRA	223.960007	223.960007	226.649994	222.580002	225.970001	355400.0	0.000133	29.494977	5.400991	5.539167	5.677342	-0.057389	-1.600791
		ZTS	175.131119	176.869995	178.449997	176.270004	176.580002	1463200.0	0.000049	42.623459	5.153746	5.212559	5.271371	0.651515	-1.188278

The plot_clusters function visualizes the clusters created by the k-means algorithm. Here's a detailed explanation of the function in paragraphs:

First, the function filters the data DataFrame to separate the rows belonging to each cluster. It does this by creating four separate DataFrames, one for each cluster (0, 1, 2, and 3). This is achieved using boolean indexing:

```
cluster_0 = data[data['cluster'] == 0]
```

```
cluster_1 = data[data['cluster'] == 1]
```

```
cluster_2 = data[data['cluster'] == 2]
```

```
cluster_3 = data[data['cluster'] == 3]
```

Next, the function creates a scatter plot for each cluster using matplotlib.pyplot. Each scatter plot uses two columns from the data DataFrame for the x and y coordinates. The x-coordinate for each point is taken from the 8th column (iloc[:, 7]), and the y-coordinate is taken from the 4th column (iloc[:, 3]). Each cluster is plotted with a distinct color and label:

```
plt.scatter(cluster_0.iloc[:, 7], cluster_0.iloc[:, 3], color='red', label='cluster 0')
```

```
plt.scatter(cluster_1.iloc[:, 7], cluster_1.iloc[:, 3], color='green', label='cluster 1')
```



```
plt.scatter(cluster_2.iloc[:, 7], cluster_2.iloc[:, 3], color='blue', label='cluster 2')
plt.scatter(cluster_3.iloc[:, 7], cluster_3.iloc[:, 3], color='black', label='cluster 3')
```

The function then adds a legend to the plot to indicate which color corresponds to which cluster using `plt.legend()`. Finally, it displays the plot using `plt.show()`. The function does not return any value, as its primary purpose is to visualize the clusters.

In summary, the `plot_clusters` function separates the data DataFrame into different clusters based on the cluster labels, and then visualizes these clusters on a scatter plot, using specific columns for the x and y coordinates. Each cluster is plotted in a different color for better visualization and differentiation.

```
#The reason we didn't normalize RSI was for better visualization at this point!!

def plot_clusters(data):

    cluster_0 = data[data['cluster']==0]
    cluster_1 = data[data['cluster']==1]
    cluster_2 = data[data['cluster']==2]
    cluster_3 = data[data['cluster']==3]

    plt.scatter(cluster_0.iloc[:,7] , cluster_0.iloc[:,3] , color = 'red', label='cluster 0')
    plt.scatter(cluster_1.iloc[:,7] , cluster_1.iloc[:,3] , color = 'green', label='cluster 1')
    plt.scatter(cluster_2.iloc[:,7] , cluster_2.iloc[:,3] , color = 'blue', label='cluster 2')
    plt.scatter(cluster_3.iloc[:,7] , cluster_3.iloc[:,3] , color = 'black', label='cluster 3')

    plt.legend()
    plt.show()
    return
```

The provided code iterates through unique dates in the data DataFrame, plotting clusters for each date separately. Here's a step-by-step explanation of how it works:

First, the code iterates over each unique date in the data DataFrame:

```
for i in data.index.get_level_values('date').unique().tolist():
```

This line retrieves all unique dates from the 'date' level of the DataFrame index and converts them into a list. The loop then iterates over each date in this list.

Within the loop, the code extracts the subset of the DataFrame corresponding to the current date:

```
g = data.xs(i, level=0)
```

The `.xs(i, level=0)` method extracts the rows corresponding to the current date `i` from the data DataFrame. This creates a new DataFrame `g` that contains only the data for that specific date.

Next, a title is set for the plot to indicate the current date:

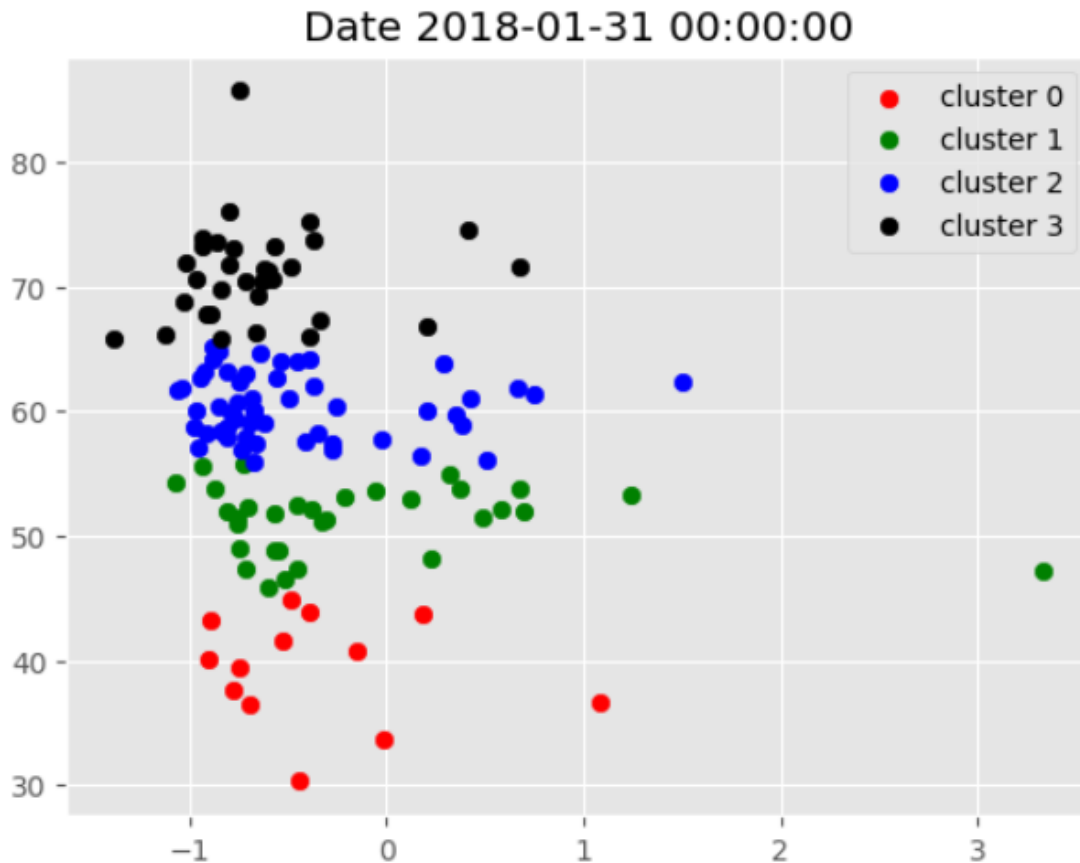
```
plt.title(f'Date {i}')
```

This sets the title of the plot to "Date {i}", where {i} is replaced with the actual date.

The `plot_clusters` function is then called to create and display the scatter plot for the extracted data:

```
for i in data.index.get_level_values('date').unique().tolist():  
  
    g = data.xs(i, level=0)  
  
    plt.title(f'Date {i}')  
    plot_clusters(g)  
  
#IDEA->Focus on stocks clustered around RSI of 65-70
```

Here is a sample of the plot on an arbitrarily chosen date:



The provided code snippet initializes centroids for a clustering algorithm, specifically targeting a particular feature—presumably the RSI (Relative Strength Index). Here's a detailed explanation:

The `target_rsi_values` list specifies the RSI values that are of particular interest. These values are `[30, 45, 55, 70]`, which are typical RSI thresholds used to assess different market conditions. For instance, an RSI of 30 is often considered an oversold condition, while an RSI of 70 is considered overbought.

The `initial_centroids` array is created with a shape of `(len(target_rsi_values), 18)`, where `len(target_rsi_values)` is 4, indicating that there will be four centroids, and 18 represents the number of features or columns in the data. This array is initialized with zeros, meaning that all centroid values are initially set to zero.

The line `initial_centroids[:, 6] = target_rsi_values` assigns the `target_rsi_values` to the 7th column (index 6) of the `initial_centroids` array. This means that the centroids' RSI feature is set to the specified target RSI values. In other words, each centroid is initialized with an RSI value corresponding to one of the target values while the other features (columns) are set to zero.

In summary, this code snippet sets up initial centroids for a clustering algorithm with specific RSI values in one of the feature columns. This setup helps guide the clustering process to focus on these target RSI levels, potentially aiming to cluster data points around these RSI values.

▼ Applying predefined centroids

```
[375]: target_rsi_values = [30, 45, 55, 70]

initial_centroids = np.zeros((len(target_rsi_values), 18))

initial_centroids[:, 6] = target_rsi_values

initial_centroids

[375]: array([[ 0.,  0.,  0.,  0.,  0.,  0., 30.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
                0.,  0.,  0.,  0.],
               [ 0.,  0.,  0.,  0.,  0.,  0., 45.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
                0.,  0.,  0.,  0.],
               [ 0.,  0.,  0.,  0.,  0.,  0., 55.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
                0.,  0.,  0.,  0.],
               [ 0.,  0.,  0.,  0.,  0.,  0., 70.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
                0.,  0.,  0.,  0.]])
```

The code filters stocks based on their cluster, selecting only those in cluster 3, which corresponds to a hypothesis that stocks near an RSI 70 centroid will continue to outperform. It then adjusts the dates by one day, resets the index, and sets a multi-level index with date and ticker. The code collects unique dates from the filtered data and creates a dictionary (fixed_dates) where each date (in 'YYYY-MM-DD' format) maps to a list of tickers for that date, facilitating the selection of stocks for each month to form a portfolio based on the Efficient Frontier's max Sharpe ratio optimization.

▼ 7. For each month select assets based on the cluster and form a portfolio based on Efficient Frontier max sharpe ratio optimization

- First we will filter only stocks corresponding to the cluster we choose based on our hypothesis.
- Momentum is persistent and my idea would be that stocks clustered around RSI 70 centroid should continue to outperform in the following month - thus I would select stocks corresponding to cluster 3.

```

: filtered_df = data[data['cluster']==3].copy()

filtered_df = filtered_df.reset_index(level=1)

filtered_df.index = filtered_df.index+pd.DateOffset(1)

filtered_df = filtered_df.reset_index().set_index(['date', 'ticker'])

dates = filtered_df.index.get_level_values('date').unique().tolist()

fixed_dates = {}

for d in dates:

    fixed_dates[d.strftime('%Y-%m-%d')] = filtered_df.xs(d, level=0).index.tolist()

fixed_dates

```

```

{'2017-11-01': ['ABBV',
               'ABT',
               'AZO',
               'BA',
               'BKNG',
               'BMY',
               'BRK-B',
               'C',
               'CCL',
               (sample)

```

The 'optimize_weights' function optimizes portfolio weights using the PyPortfolioOpt package's EfficientFrontier optimizer to maximize the Sharpe ratio. It takes the last year's stock prices as input, calculates the mean historical returns and sample covariance matrix, and sets the weight bounds for diversification (minimum half of an equal weight and maximum 10% of the portfolio). The function then uses the EfficientFrontier optimizer to find the weights that maximize the Sharpe ratio, and returns the cleaned weights, ensuring a diversified and optimized portfolio.

▼ Define portfolio optimization function



- We will define a function which optimizes portfolio weights using PyPortfolioOpt package and EfficientFrontier optimizer to maximize the sharpe ratio.
- To optimize the weights of a given portfolio we would need to supply last 1 year prices to the function.
- Apply single stock weight bounds constraint for diversification (minimum half of equally weight and maximum 10% of portfolio).

```

from pypfopt.efficient_frontier import EfficientFrontier
from pypfopt import risk_models
from pypfopt import expected_returns

```

```

def optimize_weights(prices, lower_bound=0):

    returns = expected_returns.mean_historical_return(prices=prices,
                                                    frequency=252)

    cov = risk_models.sample_cov(prices=prices,
                                frequency=252)

    ef = EfficientFrontier(expected_returns=returns,
                           cov_matrix=cov,
                           weight_bounds=(lower_bound, .1),
                           solver='SCS')

    weights = ef.max_sharpe()

    return ef.clean_weights()

```

The code snippet identifies unique stock tickers from the dataset and downloads fresh daily prices for these stocks from Yahoo Finance. The data range spans from 12 months before the earliest date in the dataset to the most recent date, ensuring up-to-date and relevant price information for further analysis and portfolio optimization.

- Download Fresh Daily Prices Data only for short listed stocks.

```

|: stocks = data.index.get_level_values('ticker').unique().tolist()

new_df = yf.download(tickers=stocks,
                     start=data.index.get_level_values('date').unique()[0]-pd.DateOffset(months=12),
                     end=data.index.get_level_values('date').unique()[-1])

```

[380]:

new_df																	
[*****100%*****] 499 of 499 completed																	
Price	Adj Close ...																
Ticker	A	AAL	AAPL	ABBV	ABNB	ABT	ACGL	ACN	ADBE	ADI	...	WTW	WY	WYNN	XEL		
Date																	
2014-12-01	38.339569	45.366467	25.683325	46.056957	NaN	36.802006	19.003332	73.201912	73.750000	44.426800	...	198791	2620000	1498700	5969000	27	
2014-12-02	38.699100	45.347511	25.585115	46.163593	NaN	37.549343	19.203333	73.184875	73.470001	44.882038	...	197772	1589500	1167100	3752000	20	
2014-12-03	38.929554	45.707573	25.875278	45.650452	NaN	37.931328	19.350000	73.644524	73.180000	46.248787	...	194299	2227400	2201300	3324100	16	
2014-12-04	38.966438	47.043552	25.777067	46.363518	NaN	37.773567	19.426666	73.491325	73.029999	46.338814	...	144809	2150500	3638400	2614200	12	
2014-12-05	38.994102	48.332161	25.667698	46.456818	NaN	37.615776	19.476667	73.363640	72.400002	46.928085	...	159229	2673500	1531500	2330300	11	
...

First, calculate the daily returns for each stock that could potentially be included in the portfolio. Then, for each month start, select the relevant stocks for that month and determine their optimal weights for the next month using the maximum Sharpe ratio optimization. If the optimization fails, assign equally-weighted weights instead. Finally, compute the daily portfolio return based on these weights.

- Calculate daily returns for each stock which could land up in our portfolio.
- Then loop over each month start, select the stocks for the month and calculate their weights for the next month.
- If the maximum sharpe ratio optimization fails for a given month, apply equally-weighted weights.
- Calculated each day portfolio return.

```
for start_date in fixed_dates.keys():

    try:

        end_date = (pd.to_datetime(start_date)+pd.offsets.MonthEnd(0)).strftime('%Y-%m-%d')

        cols = fixed_dates[start_date]

        optimization_start_date = (pd.to_datetime(start_date)-pd.DateOffset(months=12)).strftime('%Y-%m-%d')

        optimization_end_date = (pd.to_datetime(start_date)-pd.DateOffset(days=1)).strftime('%Y-%m-%d')

        optimization_df = new_df[optimization_start_date:optimization_end_date]['Adj Close'][cols]

        success = False
        try:
            weights = optimize_weights(prices=optimization_df,
                                       lower_bound=round(1/(len(optimization_df.columns)*2),3))

            weights = pd.DataFrame(weights, index=pd.Series(0))

            success = True
        except:
            print(f'Max Sharpe Optimization failed for {start_date}, Continuing with Equal-Weights')
```

```

if success==False:
    weights = pd.DataFrame([1/len(optimization_df.columns) for i in range(len(optimization_df.columns))],
                           index=optimization_df.columns.tolist(),
                           columns=pd.Series(0)).T

temp_df = returns_dataframe[start_date:end_date]

temp_df = temp_df.stack().to_frame('return').reset_index(level=0)\
    .merge(weights.stack().to_frame('weight').reset_index(level=0, drop=True),
           left_index=True,
           right_index=True)\
    .reset_index().set_index(['Date', 'index']).unstack().stack()

temp_df.index.names = ['date', 'ticker']

temp_df['weighted_return'] = temp_df['return']*temp_df['weight']

temp_df = temp_df.groupby(level=0)['weighted_return'].sum().to_frame('Strategy Return')

portfolio_df = pd.concat([portfolio_df, temp_df], axis=0)

except Exception as e:
    print(e)

portfolio_df = portfolio_df.drop_duplicates()

```



```
portfolio_df
```

```
Max Sharpe Optimization failed for 2018-04-01, Continuing with Equal-Weights
Max Sharpe Optimization failed for 2018-05-01, Continuing with Equal-Weights
Max Sharpe Optimization failed for 2020-03-01, Continuing with Equal-Weights
Max Sharpe Optimization failed for 2020-04-01, Continuing with Equal-Weights
Max Sharpe Optimization failed for 2021-02-01, Continuing with Equal-Weights
Max Sharpe Optimization failed for 2021-10-01, Continuing with Equal-Weights
Max Sharpe Optimization failed for 2022-09-01, Continuing with Equal-Weights
Max Sharpe Optimization failed for 2022-10-01, Continuing with Equal-Weights
'return'
```

Strategy Return

date	
2017-11-01	0.001481
2017-11-02	0.002892
2017-11-03	0.006382
2017-11-06	0.002932
2017-11-07	0.002917
...	...
2023-09-25	0.003587
2023-09-26	-0.011112
2023-09-27	0.004989
2023-09-28	0.007681
2023-09-29	-0.007398

1487 rows × 1 columns

The below code downloads SPY (S&P 500 ETF) price data from Yahoo Finance starting from January 1, 2015, to the current date, calculates its daily log returns, and renames the column to 'SPY Buy&Hold'. It then merges these SPY returns with the previously calculated portfolio returns, aligning them by date for comparison. This enables a direct comparison of the portfolio's performance against the S&P 500 benchmark over the same period.

8. Visualize Portfolio returns and compare to SP500 returns. ¶

```
spy = yf.download(tickers='SPY',
                  start='2015-01-01',
                  end=dt.date.today())

spy_ret = np.log(spy[['Adj Close']]).diff().dropna().rename({'Adj Close': 'SPY Buy&Hold'}, axis=1)

portfolio_df = portfolio_df.merge(spy_ret,
                                  left_index=True,
                                  right_index=True)

portfolio_df
```

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

	Strategy Return	SPY Buy&Hold
2017-11-01	0.001481	0.001321
2017-11-02	0.002892	0.000388
2017-11-03	0.006382	0.003333
2017-11-06	0.002932	0.001547
2017-11-07	0.002917	-0.000696
...
2023-09-25	0.003587	0.004196
2023-09-26	-0.011112	-0.014800
2023-09-27	0.004989	0.000399
2023-09-28	0.007681	0.005781
2023-09-29	-0.007398	-0.002430

1487 rows × 2 columns

The below code visualizes the cumulative returns of the unsupervised learning trading strategy over time. Using a ggplot style, it calculates cumulative returns by taking the exponential of the cumulative sum of log returns. The plot, spanning from the start date to September 29, 2023, displays the portfolio's performance. The y-axis is formatted to show percentages, and the plot is titled "Unsupervised Learning Trading Strategy Returns Over Time," providing a clear visual comparison of returns.

```

import matplotlib.ticker as mtick

plt.style.use('ggplot')

portfolio_cumulative_return = np.exp(np.log1p(portfolio_df).cumsum())-1

portfolio_cumulative_return[:'2023-09-29'].plot(figsize=(16,6))

plt.title('Unsupervised Learning Trading Strategy Returns Over Time')

plt.gca().yaxis.set_major_formatter(mtick.PercentFormatter(1))

plt.ylabel('Return')

plt.show()

```



Footnotes:

Alpha: Alpha represents the excess return of an investment relative to the return of a benchmark index. It is often seen as a measure of an investment's ability to beat the market. Positive Alpha indicates that the investment has outperformed the benchmark index. Negative Alpha indicates that the investment has underperformed the benchmark index. In algorithmic trading, strategies aim to generate positive alpha by identifying and exploiting market inefficiencies. High alpha indicates successful strategies.

Beta: Beta measures the volatility or systematic risk of a security or portfolio in comparison to the market as a whole. It indicates how much the price of the security is expected to move relative to the market. Beta is used to understand the risk associated with a trading strategy. A high beta strategy is more volatile and riskier, while a low beta strategy is more stable.

Beta = 1: The security's price moves with the market.

Beta > 1: The security's price is more volatile than the market.

Beta < 1: The security's price is less volatile than the market.

(2)Twitter Sentiment Investing Strategy:

Importing the libraries:

▼ Twitter Sentiment Investing Strategy

1. Load Twitter Sentiment Data

- Load the twitter sentiment dataset, set the index, calculate engagement ratio and filter out stocks with no significant twitter activity.

```
[19]: import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import datetime as dt
import yfinance as yf
import os
plt.style.use('ggplot')
```

```
data_folder="D:\\shxxtzcoders Computing Arsenal\\Quant\\AlgorithmicTrading\\MachineLearningQuantStrategies\\Project2_TwitterSentimentInvestingStrategy"
sentiment_df= pd.read_csv(os.path.join(data_folder, 'sentiment_data.csv'))
```

This code snippet reads a CSV file named sentiment_data.csv from a folder located at the specified location(Note the use of double quotes for string to avoid conflict with the single-quote in shxxtzcoders) and loads its contents into a DataFrame called sentiment_df. The os.path.join function is used to create the full file path, and pd.read_csv is used to read the CSV file into the DataFrame.

Convert the 'date' column to datetime objects,Set the multi-index using 'date' and 'symbol':

```
sentiment_df['date']=pd.to_datetime(sentiment_df['date'])

sentiment_df=sentiment_df.set_index(['date','symbol'])
```

This line of code will compute the engagement ratio for each row and add it as a new column to your DataFrame. Make sure that the columns twitterComments and twitterLikes exist and have valid numeric data to avoid any errors.

```
sentiment_df['engagement_ratio']=sentiment_df['twitterComments']/sentiment_df['twitterLikes']
```

filters the sentiment_df DataFrame to include only rows where twitterLikes are greater than 20 and twitterComments are greater than 10.

```
sentiment_df=sentiment_df[(sentiment_df['twitterLikes']>20)&(sentiment_df['twitterComments']>10)]
```

The sentiment data frame now:

```
sentiment_df
```

```
[19]:
```

		twitterPosts	twitterComments	twitterLikes	twitterImpressions	twitterSentiment	engagement_ratio
date symbol							
2021-11-18	AAPL	811.0	2592.0	21674.0	7981808.0	NaN	0.119590
	AMD	150.0	675.0	2949.0	1645270.0	NaN	0.228891
	AMZN	557.0	1315.0	12969.0	5590695.0	NaN	0.101396
	ATVI	82.0	36.0	131.0	1310715.0	NaN	0.274809
	BA	61.0	55.0	342.0	425847.0	NaN	0.160819
...
2023-01-04	TMO	21.0	2.0	32.0	30857.0	0.610020	0.062500
	TSLA	6767.0	540711.0	3810688.0	55464921.0	0.543057	0.141893
	TSN	35.0	168.0	460.0	57207.0	0.561900	0.365217
	V	132.0	1008.0	5943.0	139835.0	0.567286	0.169611
	XOM	212.0	374.0	2071.0	483389.0	0.588914	0.180589

26112 rows × 6 columns

Now we calculate on a monthly level and calculate the average monthly metric

2. Aggregate Monthly and calculate average sentiment for the month

- Aggregate on a monthly level and calculate average monthly metric, for the one we choose.

```
[22]: aggregated_df = (sentiment_df.reset_index('symbol').groupby([pd.Grouper(freq='M'), 'symbol'])  
                        [['engagement_ratio']].mean())
```

```
aggregated_df['rank'] = (aggregated_df.groupby(level=0)['engagement_ratio']  
                        .transform(lambda x: x.rank(ascending=False)))
```

aggragated_df

[22]:

		engagement_ratio	rank
date	symbol		
2021-11-30	AAL	0.203835	38.0
	AAPL	0.256318	23.0
	ABBV	0.244677	26.0
	ABT	0.285456	17.0
	AES	0.864613	2.0
...
2023-01-31	TMO	0.243042	39.0
	TSLA	0.151992	73.0
	TSN	0.280553	27.0
	V	0.194045	61.0
	XOM	0.217904	52.0

1112 rows × 4 columns

Choosing based on their cross sectional ranking for each month

3. Select Top 5 Stocks based on their cross-sectional ranking for each month

- Select top 5 stocks by rank for each month and fix the date to start at beginning of next month.

```
[27]: filtered_df = aggregated_df[aggregated_df['rank'] < 6].copy()
filtered_df = filtered_df.reset_index(level=1)
filtered_df.index = filtered_df.index + pd.DateOffset(1)
filtered_df = filtered_df.reset_index().set_index(['date', 'symbol'])
```

```
filtered_df.head(20)
```

```
[27]:
```

		engagement_ratio	rank
	date	symbol	
	2021-12-01	AES	0.864613
		FCX	0.626323
		MNST	0.699721
		OXY	2.147741
		SLB	0.647374
	2022-01-01	FCX	0.841220
		ILMN	0.741935
		L	6.507246
		LUV	1.303215
		MA	0.883401
	2022-02-01	AMD	0.715556
		D	1.037446
		FCX	0.655237
		LUV	1.035258
		MA	0.729063

(only a portion of table is visible in the image)

What follows are code segments which are self-explanatory owing to their simplicity.

▼ 4. Extract the stocks to form portfolios with at the start of each new month ¶

- Create a dictionary containing start of month and corresponded selected stocks.

```
[32]: dates = filtered_df.index.get_level_values('date').unique().tolist()
fixed_dates = {}
for d in dates:
    fixed_dates[d.strftime('%Y-%m-%d')] = filtered_df.xs(d, level=0).index.tolist()
```

```
fixed_dates
```

```
[32]: {'2021-12-01': ['AES', 'FCX', 'MNST', 'OXY', 'SLB'],
      '2022-01-01': ['FCX', 'ILMN', 'L', 'LUV', 'MA'],
      '2022-02-01': ['AMD', 'D', 'FCX', 'LUV', 'MA'],
      '2022-03-01': ['FCX', 'GILD', 'LUV', 'MRO', 'OXY'],
      '2022-04-01': ['A', 'CRM', 'PFE', 'PM', 'STZ'],
      '2022-05-01': ['AMD', 'CRM', 'CVX', 'J', 'KEY'],
      '2022-06-01': ['AMD', 'DD', 'FCX', 'KEY', 'LMT'],
      '2022-07-01': ['CB', 'CRM', 'DD', 'FCX', 'STZ'],
      '2022-08-01': ['DD', 'JPM', 'META', 'REGN', 'STZ'],
      '2022-09-01': ['ABT', 'DIS', 'L', 'META', 'MRNA'],
      '2022-10-01': ['J', 'KEY', 'L', 'META', 'MU'],
      '2022-11-01': ['A', 'DD', 'FCX', 'J', 'META'],
      '2022-12-01': ['AEP', 'AES', 'DAL', 'J', 'STZ'],
      '2023-01-01': ['A', 'AES', 'DAL', 'J', 'KEY'],
      '2023-02-01': ['AES', 'BIIB', 'FCX', 'GILD', 'MDT']}
```

▼ 5. Download fresh stock prices for only selected/shortlisted stocks ¶

```
[35]: stocks_list = sentiment_df.index.get_level_values('symbol').unique().tolist()

prices_df = yf.download(tickers=stocks_list,
                        start='2021-01-01',
                        end='2023-03-01')

[*****100%*****] 85 of 85 completed

1 Failed download:
['ATVI']: YFTzMissingError('%%ticker%: possibly delisted; No timezone found')
```

(1 download above failed due to survivorship bias)

‘returns_df’ is created by calculating the logarithmic returns of the adjusted closing prices ('Adj Close') from prices_df. This is done using the log difference of prices, and missing values are dropped. An empty DataFrame portfolio_df is initialized to store the portfolio returns for each period.

6. Calculate Portfolio Returns with monthly rebalancing

```
[42]: returns_df = np.log(prices_df['Adj Close']).diff().dropna()

portfolio_df = pd.DataFrame()
```

For each start date in fixed_dates, the corresponding end date is calculated as the last day of the same month. The columns (symbols) to be considered for that period are retrieved from fixed_dates. For each period, the mean returns of the selected columns (symbols) are calculated, and a temporary DataFrame temp_df is created to store these portfolio returns. The temporary DataFrame temp_df is concatenated to portfolio_df, building the full portfolio returns DataFrame over time.


```

for start_date in fixed_dates.keys():

    end_date = (pd.to_datetime(start_date)+pd.offsets.MonthEnd()).strftime('%Y-%m-%d')

    cols = fixed_dates[start_date]

    temp_df = returns_df[start_date:end_date][cols].mean(axis=1).to_frame('portfolio_return')

    portfolio_df = pd.concat([portfolio_df, temp_df], axis=0)

```

The final `portfolio_df` contains the portfolio returns, calculated as the mean of the selected stocks' returns for each period, with monthly rebalancing based on the dates and symbols specified in `fixed_dates`.

```
portfolio_df
```

portfolio_return	
Date	
2021-12-01	-0.016417
2021-12-02	0.024872
2021-12-03	-0.007711
2021-12-06	0.023926
2021-12-07	0.030547
...	...
2023-02-22	-0.007870
2023-02-23	-0.007323
2023-02-24	-0.009463
2023-02-27	-0.001871
2023-02-28	0.001403

312 rows × 1 columns

Before we move on further, we define **QQQ**.

QQQ:QQQ is an exchange-traded fund (ETF) that tracks the Nasdaq-100 Index, which includes 100 of the largest non-financial companies listed on the Nasdaq stock market.

The historical adjusted closing prices for the QQQ ETF (representing the NASDAQ-100) are downloaded using yfinance for the specified date range. Logarithmic returns for QQQ are calculated from the adjusted closing prices, and the result is stored in a DataFrame `qqq_ret` with the column named 'nasdaq_return'. The QQQ returns DataFrame `qqq_ret` is merged with the previously calculated `portfolio_df` DataFrame on the index (dates). This combines the portfolio returns and the NASDAQ/QQQ returns into a single DataFrame. The resulting `portfolio_df` now contains both the portfolio returns and the NASDAQ/QQQ returns, allowing for comparison between one's strategy and the benchmark.

7. Download NASDAQ/QQQ prices and calculate returns to compare to our strategy

```
[45]: qqq_df = yf.download(tickers='QQQ',
                        start='2021-01-01',
                        end='2023-03-01')

qqq_ret = np.log(qqq_df['Adj Close']).diff().to_frame('nasdaq_return')
```

```
portfolio_df = portfolio_df.merge(qqq_ret,
                                left_index=True,
                                right_index=True)
```

```
portfolio_df
```

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

	portfolio_return	nasdaq_return
Date		
2021-12-01	-0.016417	-0.017159
2021-12-02	0.024872	0.007181
2021-12-03	-0.007711	-0.017542
2021-12-06	0.023926	0.007981
2021-12-07	0.030547	0.029669
...
2023-02-22	-0.007870	0.000748
2023-02-23	-0.007323	0.008696
2023-02-24	-0.009463	-0.016886
2023-02-27	-0.001871	0.007136
2023-02-28	0.001403	-0.001294

312 rows × 2 columns

Convert the portfolio and NASDAQ returns to cumulative returns using the formula for cumulative log returns.

Plot Cumulative Returns:

- Plot the cumulative returns using the plot function with a specified figure size.
- Set the title and labels for the plot.
- Format the y-axis to display percentages.
- Display the plot.

```

portfolios_cumulative_return = np.exp(np.log1p(portfolio_df).cumsum()).sub(1)

portfolios_cumulative_return.plot(figsize=(16,6))

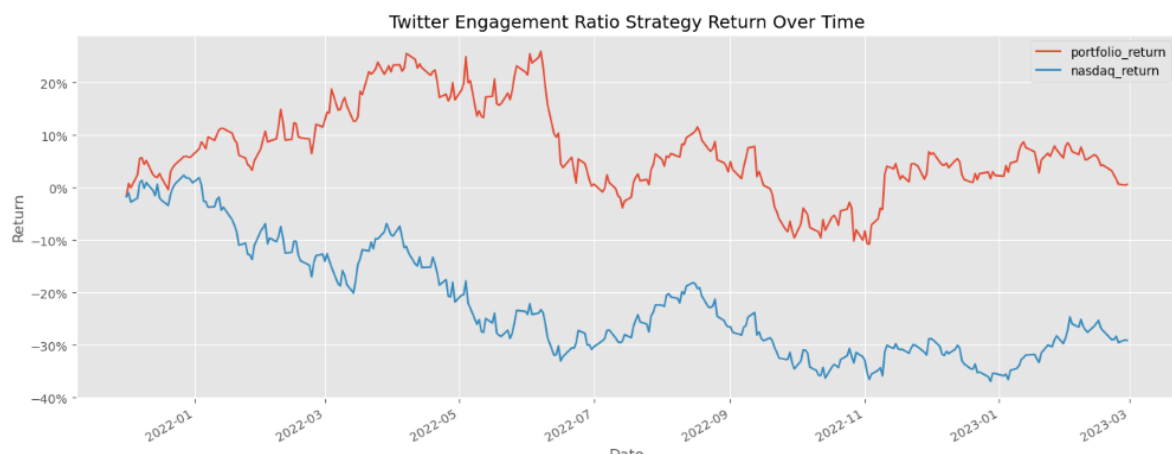
plt.title('Twitter Engagement Ratio Strategy Return Over Time')

plt.gca().yaxis.set_major_formatter(mtick.PercentFormatter(1))

plt.ylabel('Return')

plt.show()

```



(3) Intraday Strategy Using GARCH Model:

Loading the required libraries:

1. Load Simulated Daily and Simulated 5-minute data.

- We are loading both datasets, set the indexes and calculate daily log returns.

```

8]: import matplotlib.pyplot as plt
from arch import arch_model
import pandas_ta
import pandas as pd
import numpy as np
import os

```

Now we set the `data_folder` which specifies the directory where the CSV files are stored. We read the daily data into a DataFrame named `daily_df`. Then, we remove the column 'Unnamed: 7' which is not needed. Finally, we change the 'Date' column to datetime format.

Now, set the 'Date' column as the index of the DataFrame.

```
data_folder = "D:\shxxtzcode's Computing Arsenal\Quant\AlgorithmicTrading\MachineLearningQuantStrategies\Project3_IntradayStrategyUsingGARCHmodel"
daily_df = pd.read_csv(os.path.join(data_folder, 'simulated_daily_data.csv'))
```

```
daily_df = daily_df.drop('Unnamed: 7', axis=1)

daily_df['Date'] = pd.to_datetime(daily_df['Date'])

daily_df = daily_df.set_index('Date')
```

Load the 5-minute interval data into a DataFrame named `intraday_5min_df`. Remove the column 'Unnamed: 6' that is not needed. Change the 'datetime' column to datetime format. Set the 'datetime' column as the index of the DataFrame. Finally, create a new 'date' column by extracting the date part from the datetime index.

```
intraday_5min_df = pd.read_csv(os.path.join(data_folder, 'simulated_5min_data.csv'))
```

```
intraday_5min_df = intraday_5min_df.drop('Unnamed: 6', axis=1)
```

```
intraday_5min_df['datetime'] = pd.to_datetime(intraday_5min_df['datetime'])

intraday_5min_df = intraday_5min_df.set_index('datetime')

intraday_5min_df['date'] = pd.to_datetime(intraday_5min_df.index.date)
```

'intraday_5min_df' is printed to check the loaded and processed data.

```
intraday_5min_df
```

```
[8]:
```

	open	low	high	close	volume	date
datetime						
2021-09-29 20:00:00	10379.7775	10364.5950	10398.7025	10370.9575	46	2021-09-29
2021-09-29 20:05:00	10370.9425	10352.4175	10380.2500	10371.1450	53	2021-09-29
2021-09-29 20:10:00	10372.8150	10357.8250	10388.7500	10384.3125	116	2021-09-29
2021-09-29 20:15:00	10385.8275	10384.0825	10457.2000	10442.5175	266	2021-09-29
2021-09-29 20:20:00	10442.5225	10426.2375	10448.0000	10440.9950	65	2021-09-29
...
2023-09-20 10:40:00	6792.1025	6783.6000	6796.2500	6790.1375	41	2023-09-20
2023-09-20 10:45:00	6790.5575	6779.1000	6795.7500	6781.9175	42	2023-09-20
2023-09-20 10:50:00	6781.5475	6779.3750	6790.2500	6783.9050	44	2023-09-20
2023-09-20 10:55:00	6783.9025	6779.9000	6793.2500	6782.0900	95	2023-09-20
2023-09-20 11:00:00	6783.7750	6774.3500	6787.5000	6778.6375	54	2023-09-20

```
177877 rows × 6 columns
```

Compute the logarithmic returns of the adjusted closing prices. This is done using the `np.log` of the adjusted closing prices' differences.

Compute the rolling variance of the logarithmic returns over a 6-month window (approximately 180 trading days). This provides a baseline measure of volatility.

Slice the DataFrame to include only data from the year 2020 and onwards.

2. Define function to fit GARCH model and predict 1-day ahead volatility in a rolling window.

- We are first calculating the 6-month rolling variance and then we are creating a function in a 6-month rolling window to fit a garch model and predict the next day variance.

```
11]: daily_df['log_ret'] = np.log(daily_df['Adj Close']).diff()

daily_df['variance'] = daily_df['log_ret'].rolling(180).var()

daily_df = daily_df['2020:']
```

`predict_volatility(x)` fits a GARCH(1, 3) model to the input series `x` and forecasts the variance for the next day (horizon=1).

The GARCH model is fitted with `arch_model` from the `arch` library, and the forecasted variance is extracted from the model.

```
def predict_volatility(x):

    best_model = arch_model(y=x,
                            p=1,
                            q=3).fit(update_freq=5,
                                    disp='off')

    variance_forecast = best_model.forecast(horizon=1).variance.iloc[-1,0]

    print(x.index[-1])

    return variance_forecast
```

Use 'rolling(180).apply(lambda x: predict_volatility(x))' to apply the predict_volatility function over a 180-day rolling window. This computes the 1-day ahead volatility prediction for each window.

Remove rows with missing values from daily_df that result from the rolling calculations.

```
daily_df['predictions'] = daily_df['log_ret'].rolling(180).apply(lambda x: predict_volatility(x))

daily_df = daily_df.dropna()
```

```
2020-06-28 00:00:00
2020-06-29 00:00:00
2020-06-30 00:00:00
2020-07-01 00:00:00
2020-07-02 00:00:00
2020-07-03 00:00:00
2020-07-04 00:00:00
```

This gives a very long warning as well(not shown here). We Chat-GPTed this warning to dig into it and briefly touch upon it here. The large warning is due to the use of the arch_model function within a rolling window. The arch_model fitting process can be computationally expensive and generates numerous warnings when applied to a large dataset with many rolling windows. Common warnings may include convergence warnings, fitting issues, or numerical stability problems.

Possible Causes of Warnings

1. Model Convergence Issues:

- The GARCH model might have trouble converging for certain rolling windows, especially with many parameters (like p=1, q=3) or on smaller datasets.

2. Numerical Instability:

- With many rolling windows, the fitting process can encounter numerical instability or precision issues, especially with volatile data.

3. Large Number of Model Fits:

- If your dataset is large, applying the model across many rolling windows will result in many individual fits, which can generate warnings.

4. Data Sufficiency:

- In the initial rolling windows, there may not be enough data to fit the model properly, which can cause warnings or errors.

(however at the end we get this table :)

Out[23]:

	Open	High	Low	Close	Adj Close	Volume	log_ret	variance	predictions
Date									
2020-06-28	2262.115234	2299.386719	2243.881348	2285.895508	2285.895508	1.456087e+10	0.010797	0.002473	0.000728
2020-06-29	2285.007324	2309.393311	2260.468994	2297.713623	2297.713623	1.646055e+10	0.005157	0.002473	0.000651
2020-06-30	2296.395264	2304.458984	2271.209473	2284.498291	2284.498291	1.573580e+10	-0.005768	0.002468	0.000877
2020-07-01	2286.496338	2327.438721	2276.183838	2307.081299	2307.081299	1.597155e+10	0.009837	0.002455	0.000618
2020-07-02	2307.784912	2318.740723	2259.155762	2280.852539	2280.852539	1.633892e+10	-0.011434	0.002456	0.000703
...
2023-09-14	6557.069336	6693.655762	6542.862793	6634.918457	6634.918457	1.381136e+10	0.011801	0.000393	0.000334
2023-09-15	6633.454590	6710.124512	6560.175293	6652.173340	6652.173340	1.147974e+10	0.002597	0.000385	0.000292
2023-09-16	6651.549805	6688.692383	6618.472656	6642.070313	6642.070313	7.402031e+09	-0.001520	0.000384	0.000286
2023-09-17	6641.981934	6654.499512	6611.268555	6633.546875	6633.546875	6.774211e+09	-0.001284	0.000383	0.000280
2023-09-18	6633.248535	6853.683594	6603.878906	6688.570313	6688.570313	1.561534e+10	0.008261	0.000378	0.000298

1178 rows × 9 columns

The prediction premium is computed as the difference between the GARCH model's predicted variance and the actual rolling variance, normalized by the rolling variance.

Compute the rolling standard deviation of the prediction premium over a 6-month (180-day) window. This measures the volatility of the prediction premium.

3. Calculate prediction premium and form a daily signal from it.

- We are calculating the prediction premium. And calculate its 6-month rolling standard deviation.
- From this we are creating our daily signal.

```
daily_df['prediction_premium'] = (daily_df['predictions']-daily_df['variance'])/daily_df['variance']
```

```
daily_df['premium_std'] = daily_df['prediction_premium'].rolling(180).std()
```

Create a daily trading signal based on the prediction premium and its rolling standard deviation. If the prediction premium is greater than the rolling

standard deviation, assign a signal of 1. If the prediction premium is less than the negative rolling standard deviation, assign a signal of -1. If the prediction premium is between the negative and positive rolling standard deviations, assign NaN.

Shift the daily signal by one day to align with the prediction (i.e., the signal for today should be applied to tomorrow).

```
daily_df['signal_daily'] = daily_df.apply(lambda x: 1 if (x['prediction_premium'] > x['premium_std'])
                                         else (-1 if (x['prediction_premium'] < x['premium_std'] * -1) else np.nan),
                                         axis=1)

daily_df['signal_daily'] = daily_df['signal_daily'].shift()
```

The resulting DataFrame `daily_df` will now include the `prediction_premium`, `premium_std`, and `signal_daily` columns.

daily_df												
	Open	High	Low	Close	Adj Close	Volume	log_ret	variance	predictions	prediction_premium	premium_std	signal_daily
Date												
2020-06-28	2262.115234	2299.386719	2243.881348	2285.895508	2285.895508	1.456087e+10	0.010797	0.002473	0.000728	-0.705556	NaN	NaN
2020-06-29	2285.007324	2309.393311	2260.468994	2297.713623	2297.713623	1.646055e+10	0.005157	0.002473	0.000651	-0.736678	NaN	NaN
2020-06-30	2296.395264	2304.458984	2271.209473	2284.498291	2284.498291	1.573580e+10	-0.005768	0.002468	0.000877	-0.644670	NaN	NaN
2020-07-01	2286.496338	2327.438721	2276.183838	2307.081299	2307.081299	1.597155e+10	0.009837	0.002455	0.000618	-0.748302	NaN	NaN
2020-07-02	2307.784912	2318.740723	2259.155762	2280.852539	2280.852539	1.633892e+10	-0.011434	0.002456	0.000703	-0.713818	NaN	NaN
...
2023-09-14	6557.069336	6693.655762	6542.862793	6634.918457	6634.918457	1.381136e+10	0.011801	0.000393	0.000334	-0.151792	0.544319	NaN
2023-09-15	6633.454590	6710.124512	6560.175293	6652.173340	6652.173340	1.147974e+10	0.002597	0.000385	0.000292	-0.240810	0.541657	NaN
2023-09-16	6651.549805	6688.692383	6618.472656	6642.070313	6642.070313	7.402031e+09	-0.001520	0.000384	0.000286	-0.256793	0.540672	NaN

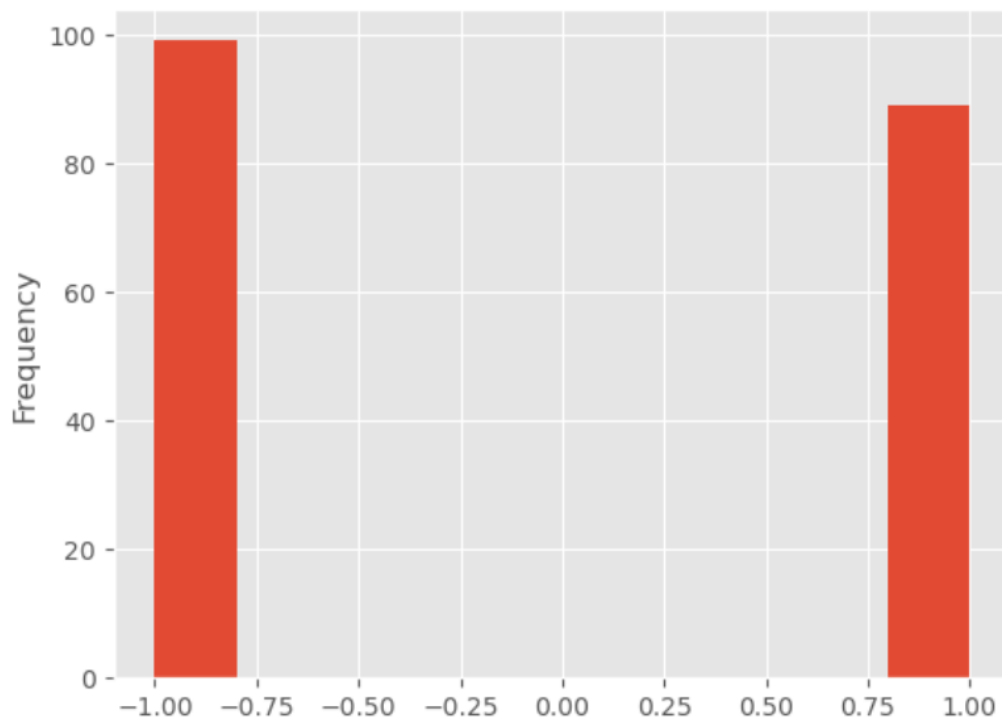
(partial table shown above)

Plotting the `signal_daily` field:

```
plt.style.use('ggplot')

daily_df['signal_daily'].plot(kind='hist')

plt.show()
```



Merge the intraday data (intraday_5min_df) with the daily trading signal (daily_df), aligning on the date.

Drop redundant date columns and set the datetime index.

4. Merge with intraday data and calculate intraday indicators to form the intraday signal.

- Calculate all intraday indicators and intraday signal.

```
final_df = intraday_5min_df.reset_index()\
    .merge(daily_df[['signal_daily']].reset_index(),
           left_on='date',
           right_on='Date')\
    .drop(['date', 'Date'], axis=1)\
    .set_index('datetime')
```

Calculate the 20-period RSI to gauge overbought or oversold conditions.

```
final_df['rsi'] = pandas_ta.rsi(close=final_df['close'],
                                length=20)
```

Calculate the lower Bollinger Band('lband') for 20 periods. Calculate the upper Bollinger Band('uband') for 20 periods.

```
final_df['lband'] = pandas_ta.bbands(close=final_df['close'],
                                     length=20).iloc[:,0]

final_df['uband'] = pandas_ta.bbands(close=final_df['close'],
                                     length=20).iloc[:,2]
```

Now, form the intraday trading signal:

- **Buy Signal:** If RSI > 70 and the close price is above the upper Bollinger Band.
- **Sell Signal:** If RSI < 30 and the close price is below the lower Bollinger Band.
- **Neutral:** If neither condition is met.

Calculate Intraday Returns:

Compute the log returns of the close price for intraday performance analysis.

```
final_df['signal_intraday'] = final_df.apply(lambda x: 1 if (x['rsi']>70)&
                                             (x['close']>x['uband'])
                                             else (-1 if (x['rsi']<30)&
                                             (x['close']<x['lband']) else np.nan),
                                             axis=1)
```

```
final_df['return'] = np.log(final_df['close']).diff()
```

final_df											
[21]:	open	low	high	close	volume	signal_daily	rsi	lband	uband	signal_intraday	return
datetime											
2021-09-29 20:00:00	10379.7775	10364.5950	10398.7025	10370.9575	46	NaN	NaN	NaN	NaN	NaN	NaN
2021-09-29 20:05:00	10370.9425	10352.4175	10380.2500	10371.1450	53	NaN	NaN	NaN	NaN	NaN	0.000018
2021-09-29 20:10:00	10372.8150	10357.8250	10388.7500	10384.3125	116	NaN	NaN	NaN	NaN	NaN	0.001269
2021-09-29 20:15:00	10385.8275	10384.0825	10457.2000	10442.5175	266	NaN	NaN	NaN	NaN	NaN	0.005589
2021-09-29 20:20:00	10442.5225	10426.2375	10448.0000	10440.9950	65	NaN	NaN	NaN	NaN	NaN	-0.000146
...
2023-09-18 23:35:00	6708.0025	6705.5175	6712.0000	6708.8675	11	NaN	62.861699	6677.365065	6715.855435	NaN	0.000128
2023-09-18 23:40:00	6708.8650	6707.1800	6714.2500	6709.3350	22	NaN	63.165869	6679.544207	6716.559793	NaN	0.000070
2023-09-18 23:45:00	6709.9250	6704.3600	6713.5000	6708.2750	25	NaN	61.954798	6682.379916	6716.534084	NaN	-0.000158
2023-09-18 23:50:00	6708.2750	6705.7700	6712.7500	6705.8050	11	NaN	59.172065	6685.549735	6715.801265	NaN	-0.000368
2023-09-18 23:55:00	6705.8025	6705.3875	6712.0000	6706.8875	6	NaN	60.000874	6687.480314	6715.772186	NaN	0.000161

177456 rows × 11 columns

The code below processes a DataFrame `final_df` to compute and analyze trading strategy returns. First, it assigns a `return_sign` based on conditions involving `signal_daily` and `signal_intraday`, where -1 or 1 is assigned based on matching signals, and NaN otherwise. Then, it fills forward missing values in `return_sign` grouped by date. The `forward_return` is calculated by shifting the `return` column one step backward, and the `strategy_return` is computed by multiplying `forward_return` by `return_sign`. Finally, it aggregates the `strategy_return` by day, summing the values to produce `daily_return_df`.

5. Generate the position entry and hold until the end of the day.

```
[26]: final_df['return_sign'] = final_df.apply(lambda x: -1 if (x['signal_daily']==1)&(x['signal_intraday']==1)
                                             else (1 if (x['signal_daily']==-1)&(x['signal_intraday']==-1) else np.nan),
                                             axis=1)

final_df['return_sign'] = final_df.groupby(pd.Grouper(freq='D'))['return_sign']\
    .transform(lambda x: x.ffill())

final_df['forward_return'] = final_df['return'].shift(-1)

final_df['strategy_return'] = final_df['forward_return']*final_df['return_sign']

daily_return_df = final_df.groupby(pd.Grouper(freq='D'))['strategy_return'].sum()
```

6. Calculate final strategy returns.

```
] : import matplotlib.ticker as mtick

strategy_cumulative_return = np.exp(np.log1p(daily_return_df).cumsum()).sub(1)

strategy_cumulative_return.plot(figsize=(16,6))
```

Now, we move to cumulative strategy returns.

1. Calculate Cumulative Strategy Returns:

- Convert the daily strategy returns into cumulative returns. This involves computing the cumulative sum of the log returns, then exponentiating to get the cumulative return.

2. Plot the Cumulative Strategy Returns:

- Use matplotlib to plot the cumulative returns and format the y-axis to show percentages.

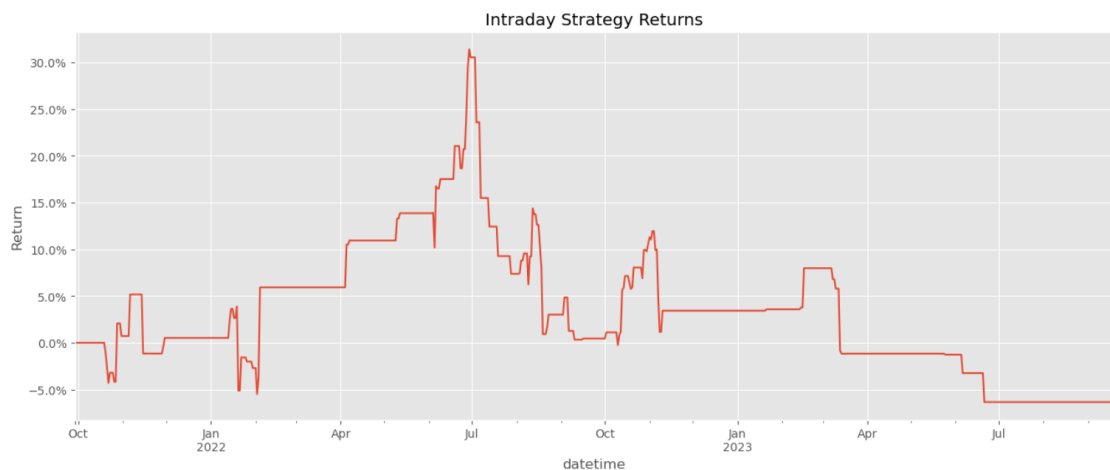
```
plt.title('Intraday Strategy Returns')

plt.gca().yaxis.set_major_formatter(mtick.PercentFormatter(1))

plt.ylabel('Return')

plt.show()
```

The `np.log1p()` function computes the natural logarithm of $1 + x$ for each daily return (used with smaller values of x for accuracy), which helps in handling compounding returns. `np.exp()` then exponentiates the cumulative sum to get the cumulative return.) Use matplotlib to create a plot. The `PercentFormatter` helps in formatting the y-axis to display the returns as percentages.



CONCLUSIONS AND RECOMMENDATIONS:

We believe that HFT is a booming field in itself and though it might have relied on traditional algorithms up to this point, with ML and DL becoming research hotbeds, the potential to exploit them in HFTs/ MFTs is huge. We believe that we are truly naive to pass a consequential educated remark on the subject, but empirical data suggests that the move to ML-oriented trading is already on. ¹

Thus, it is highly beneficial for the trading community if ML and DL, along with skills like competitive programming that encourage algorithmic thinking and boost problem solving skills be encouraged from college level.

Apart from this, the following remark is exclusively for the Indian educational scene. Quant culture should be encouraged at the undergrad level. AlgoTradingNinjas, a slack space created by AlgoBulls represents one such step in the positive direction.

Footnotes:

¹: GenAI and HFT: A Competitive Edge?

<https://www.watertechnology.com/emerging-technologies/7951483/genai-and-hft-a-competitive-edge>

REFERENCES:

OpenAI. (n.d.). *ChatGPT*. Retrieved June 19, 2024, from <https://www.openai.com/chatgpt>

Wikipedia: *The Free Encyclopedia*. Retrieved June 19, 2024, from <https://en.wikipedia.org/wiki/Article>

AlgoBulls. (2024, February). *QuantQuest: PreEvent Webinar at IIT Kanpur* [Video]. YouTube. <https://www.youtube.com/watch?v=keNaYuKZMhc>

Investopedia. Retrieved June 19, 2024, from <https://www.investopedia.com/articles>

Lost Stat. (n.d.). *Rolling regression*. Retrieved from https://lost-stats.github.io/Time_Series/Rolling_Regression.html

Lentz, D. (2019, July 24). *Understanding K-Means clustering in machine learning*. Towards Data Science. Retrieved from <https://towardsdatascience.com/understanding-k-means-clustering-in-machine-learning-6a6e67336aa1>

StatQuest with Josh Starmer. (2020, April 10). *K-Means clustering: The math of the algorithm* [Video]. YouTube. [StatQuest: K-means clustering](#)