BUY SIGNAL GENERATOR FRAMEWORK

THE NEED FOR BUY SIGNAL GENERATOR

Investors often face the decision between actively trading stocks based on buy signals or adopting a passive strategy of holding investments over the long term. This section explores the rationale behind preferring buy signals:

A. TIMING AND OPPORTUNITY:

Objective: Timing the market to capitalize on price movements can potentially enhance returns compared to a buy-and-hold strategy.

Rationale: Buy signals help identify favorable entry points based on technical indicators, potentially maximizing gains and minimizing losses.

B. RISK MANAGEMENT:

Objective: Mitigating risk through informed decision-making.

Rationale: Buy signals consider market trends and conditions, offering opportunities to adjust holdings in response to changing market dynamics, unlike a static hold strategy.

C. CAPITAL UTILIZATION:

Objective: Optimizing capital allocation for better returns.

Rationale: Buy signals guide strategic allocation of funds into assets showing potential for short-term price appreciation, optimizing portfolio performance.

D. PERFORMANCE ENHANCEMENT:

Objective: Achieving superior returns relative to market benchmarks.

Rationale: Active management based on buy signals seeks to outperform passive strategies by leveraging short-term market inefficiencies and opportunities.

1. Objective:

Predict and generate BUY signals for ETFs and S&P 500 stocks for short-term trading to achieve optimal returns based on user-defined risk parameters.

2. Assumptions

A. User Input Rationality:

Assumption: Users will provide rational and realistic inputs for annualized return and volatility according to their risk appetite.

Implication: The model's effectiveness depends on sensible user inputs to screen and select appropriate ETFs and stocks.

B. Consistent Market Conditions:

Assumption: Market conditions during the model's operation will be similar to those during the historical data period used for parameter optimization (2020-2022 for calibration and 2022-2024 for testing).

Implication: The model's parameters (Difference, Gap, and investment period) are optimized based on historical data, and significant deviations in market conditions could affect performance.

C. Fixed Investment Period:

Assumption: The chosen investment period of 30 days is appropriate for capturing short-term trends without excessive volatility.

Implication: The fixed investment period is based on simulations showing moderate risk and return, but different market conditions may suggest other optimal periods.

D. Independence of Trades:

Assumption: Each trade signal generated by the model is independent of other trades, and the model does not account for portfolio-level interactions.

Implication: Users should manage their overall portfolio risk separately, as the model focuses on individual trade signals.

3. Parameters and Variables:

- **Annualized Return (AR):** The return expected over a year.
- **Annualized Volatility (AV):** The standard deviation of returns over a year.
- **Difference:** Percentage difference between the 9-day and 21-day moving averages.
- **Gap:** Distance between the current price and the nearest moving average.
- **Investment Period (t):** The time-period for holding the investment (fixed at 30 days).

4. Model Formulation:

Screening Layer:

- Filter ETFs and stocks based on user-defined AR and AV.
- **Example:** If a user selects an AR and AV value of 20 and 15 respectively, then the screener will calculate the variables for each ETF and show the ones which meet the criteria.

Signal Generation Layer:

- Use moving averages to generate BUY signals based on Difference and Gap parameters.

Formulas:

1. **Annualized Return (AR):**

$$\mathrm{AR} = ((1 + \bar{R})^{12} - 1) \times 100$$

$$where\ (\bar{R})\ is\ the\ mean\ return\ over\ a\ period\ of\ 12\ months$$

2. **Annualized Volatility (AV)**:

$$\mathrm{AV} = \left(\bar{\sigma} \times \sqrt{12}\right) \times 100$$

$$where\ (\bar{\sigma})\ represents\ the\ mean\ volatility\ over\ 12\ months.$$

3. **Moving Averages:**

$$MA_9 = \frac{1}{9}\sum_{i=0}^{8} P_{t-i}$$

- M_0 is current price and M_8 is price 8 days ago

$$MA_{21} = \frac{1}{21}\sum_{i=0}^{20} P_{t-i}$$

- M_0 is current price and M_21 is price 21 days ago

4. **Difference and Gap**:

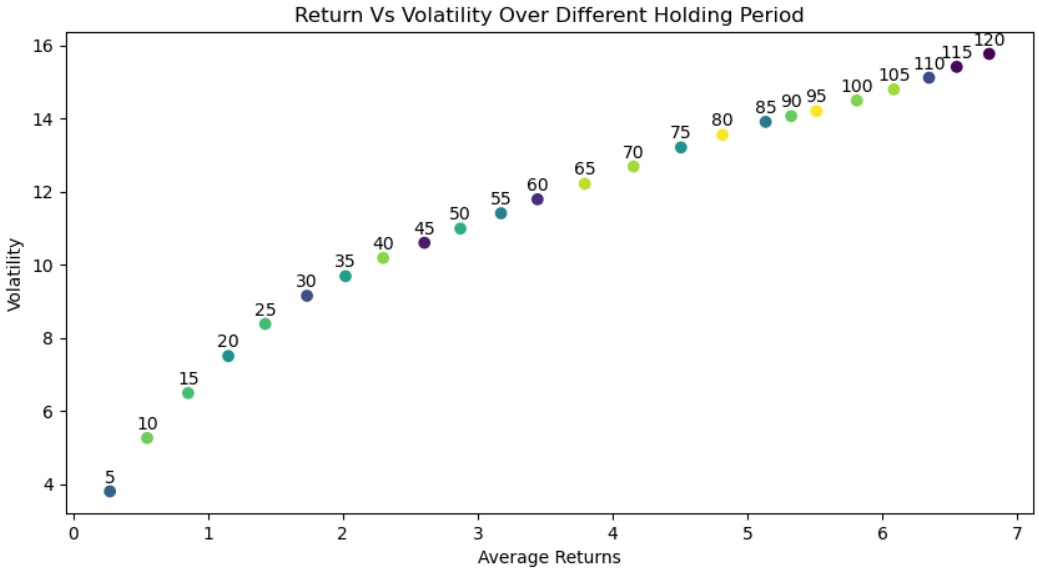
$$Difference = \frac{MA_9 - MA_{21}}{P_{current}} \times 100$$

$$Gap = \frac{P_{current} - MA_{nearest}}{P_{current}} \times 100$$

- P_current is current price in the market

5. **Investment Period:**

- Simulating of different date ranges shows that a 30 day investment period provide the best risk to return



5. Implementation Steps:

A. Collect Input Data:

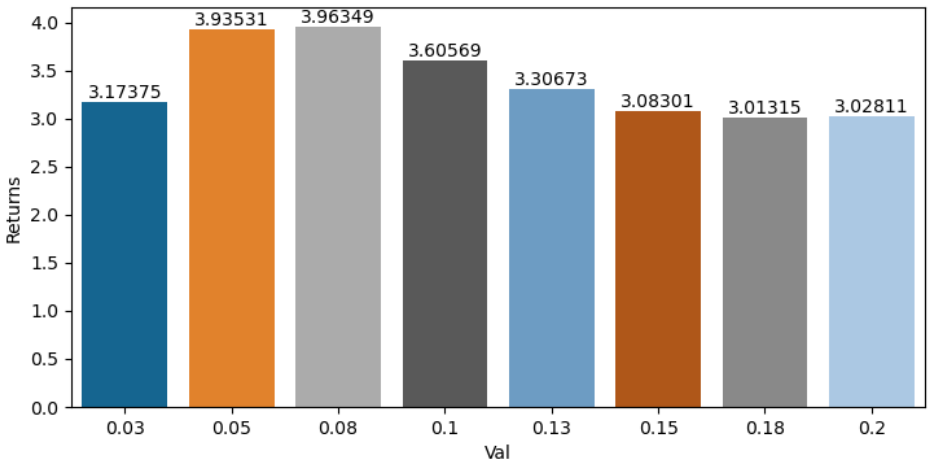
- Gather market data for ETFs and S&P 500 from Yahoo Finance API.
- Input user-defined parameters: AR, AV, and investment period.

B. Predict BUY Signals:

- Compute moving averages and evaluate Difference and Gap.
- Generate BUY signals if the conditions are met.

C. Calibration:

- Adjust Difference and Gap thresholds based on historical performance and simulations.



- After testing different values of **Gap** and **Difference** over a period of **2020** to **2022**, **0.08** comes out to be the best range

6. Detailed Project Plan:

A. Model Development:

Define Model Specifications:

- Objective: Generate BUY signals for 30 day trading as discussed in section 4.5.
- Inputs: Historical price data, user-defined AR, AV, investment period, Difference, and Gap parameters.

Data Collection:

- Gather historical market data and simulate various scenarios to refine model parameters.
- Implement data collection mechanisms using Yahoo Finance API by utilizing python.

Model Construction:

- Develop the algorithm in Python.
- Validate assumptions and methodology with financial experts.

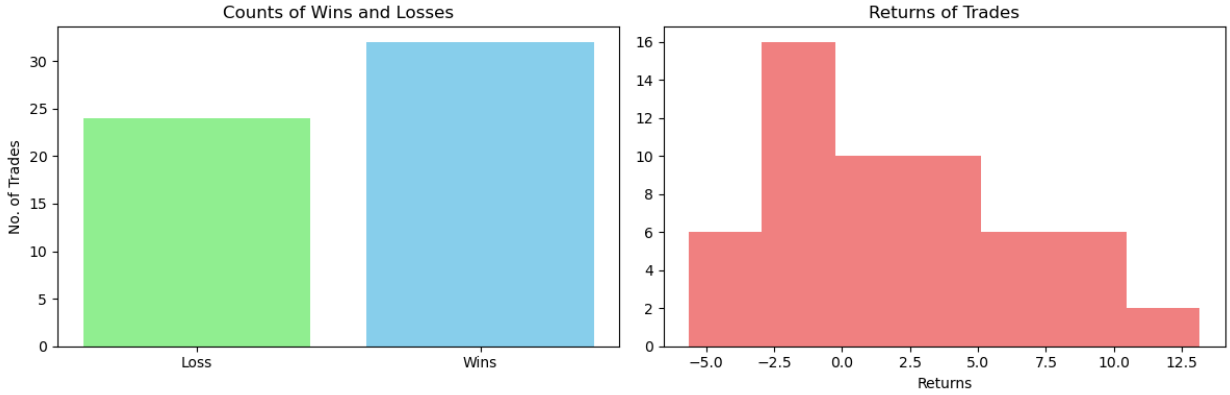
B. Model Validation:

Conduct Independent Review:

- Engage external consultants for model review and validation every year.
- Validate the model with a separate dataset from 2023-2024.

Perform Out of Sample Testing:

- Run the model on date ranges outside the zone on which the parameters are calculated.



- **Mean Returns:** 2.0537
- **Win Rate:** 57.14%

Stress Testing:

- Simulate various market conditions to test model parameter robustness.
    1. Running the model on the two most recent crashes of 2008 and 2020, the model did not generate any signals

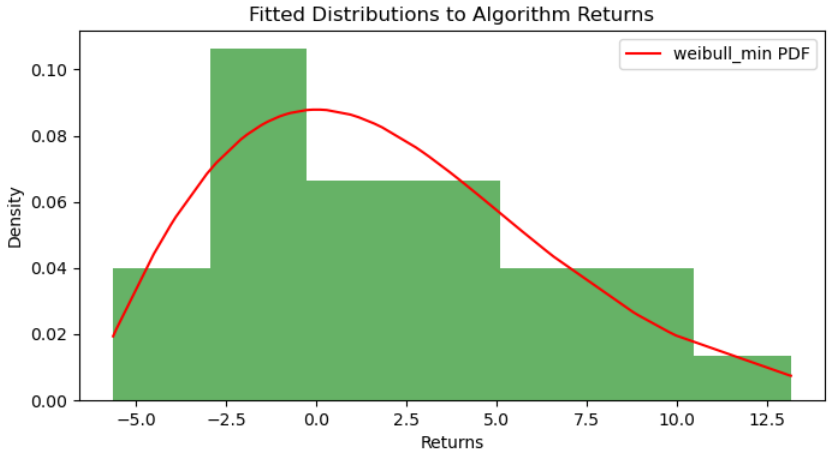
Define Performance Metrics:

- Utilize metrics such as Maximum Drawdown and Average Win Rate to assess model accuracy.

Nature of Returns:

- Utilize fitting different probability distributions to the returns generated by the model to understand nature of returns

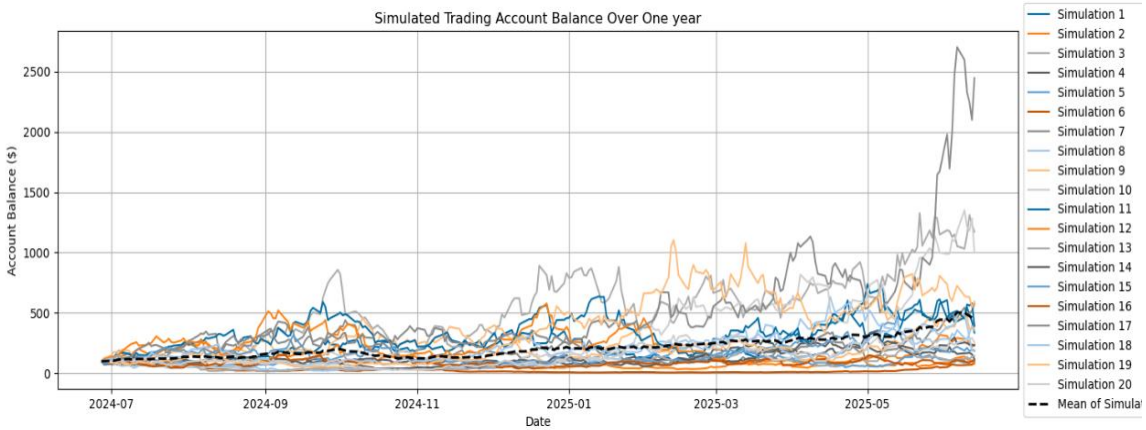
Distribution	KS D statistic	p-value	AIC	Parameters
norm	0.1317	0.262	333.41	[2.0537 4.5823]
t	0.1317	0.2621	335.41	[5.05382593e+07 2.05370000e+00 4.58240000e+00]
laplace	0.1387	0.2113	341.93	[0.9572 3.7587]
expon	0.2351	0.0033	inf	[-5.6255 7.6793]
lognorm	0.0818	0.8177	330.27	[0.3144 -12.5655 13.9219]
beta	0.0802	0.8356	329.77	[1.5693 2.7717 -5.9427 22.182]
gamma	0.0775	0.8639	329.69	[4.6441 -8.0676 2.1794]
weibull_min	0.0821	0.8154	328.72	[1.8918 -6.3144 9.4235]



- **Best fitting distribution:** Weibull min

Simulate Account Growth:

- Utilize the statistical distribution parameters found above to simulate account growth



- Simulating for 20 times for a one-year time frame with an initial capital of 100$ produced a capital of around 500$

7. Model Risks

A. Single-Sided Signals:

Risk: The model only generates BUY signals, potentially missing opportunities for short selling or other strategies that could hedge against market downturns.

Mitigation: Consider incorporating sell or short signals in future iterations. Implement stop-loss mechanisms to limit potential losses.

B. User Input Dependency:

Risk: The screener relies on user-defined parameters (annualized return and volatility), which may lead to suboptimal trading decisions if the inputs are not well-suited to current market conditions.

Mitigation: Provide default recommended values based on market analysis. Educate users on selecting appropriate parameters. Implement warnings or guidelines for extreme input values.

C. Market Sentiment Ignorance:

Risk: The model does not account for market sentiment or external factors such as news events, which can significantly impact market movements and lead to unexpected losses.

Mitigation: Integrate sentiment analysis tools or news feeds to enhance the model's decision-making process. Use additional indicators that reflect market sentiment.

D. Overfitting to Historical Data:

Risk: The parameters (Difference and Gap thresholds) are optimized based on historical data, which may not generalize well to future market conditions, leading to poor performance.

Mitigation: Regularly update and recalibrate the model using new data. Implement cross-validation techniques and avoid overfitting by using robust statistical methods.

E. Economic and Market Condition Changes:

Risk: Changes in economic conditions or market structure (e.g., introduction of new regulations, shifts in market dynamics) can impact the model's effectiveness.

Mitigation: Stay adaptive to changing market conditions by regularly updating the model parameters every year. Engage in continuous research to keep the model aligned with current market trends.

8. Model Usage:

A. User Training:

- Conduct training sessions for end-users (traders) on AV and AR parameter adjustment based on market scenario.
- Provide model risk details such as single sided signals, user input dependency, market sentiment ignorance and overfitting.

B. Continuous Monitoring:

- Set up systems to monitor model performance continuously every week.
- Use dashboards to visualize ongoing trades.

C. Feedback Loop:

- Collect user feedback on model predictions and trading outcomes.

D. Integration with Operations:

- Embed the algorithm into trading platforms for automated execution of BUY signals.
- Ensure smooth integration with market data feeds and trading systems.

9. References:

Mastering MA Crossover Strategies for Profitable Trading

https://medium.com/@unschooledtrader/mastering-ema-crossover-strategies-for-profitable-trading-1f786e3b1242

How to Use a Moving Average to Buy Stocks

https://www.investopedia.com/articles/active-trading/052014/how-use-moving-average-buy-stocks.asp

Moving Average Crossover Strategies

https://trendspider.com/learning-center/moving-average-crossover-strategies/

CODE

```
import pandas as pd
import yfinance as yf
import numpy as np
import datetime as dt
from datetime import timedelta
import matplotlib.pyplot as plt
import statistics as st
import warnings
warnings.filterwarnings('ignore')
import seaborn as sns
from random import randint,seed
plt.style.use('tableau-colorblind10')
from collections import Counter
```

```
tickers=
['^GSPC','XLE','XLF','XLU','XLI','GDX','XLK','XLV','XLY','XLP','XLB','XOP','IYR','XHB','ITB','VNQ','GDXJ','IYE','O
IH','XME','XRT','SMH','IBB','KBE','XTL']
```

```
def analysis(ticker, start_date, end_date, time):
  # Download historical stock data using Yahoo Finance API
  data = yf.download(ticker, start_date, end_date)

  # Initialize an empty list to store returns
  returns = []

  # Iterate through each row (index) in the historical data
  for index in range(len(data)):
    # Get the closing price of the stock on the current day
    buy_price = data.iloc[index, 3]

    # Calculate the day to sell based on the current index and specified time
    days = index + time

    # Check if the calculated day to sell is within the range of data
    if days >= len(data):
      break
    else:
      # Get the closing price of the stock on the sell day
      sell_price = data.iloc[days, 3]
```

```
        # Calculate the profit percentage based on buy and sell prices
        profit = ((sell_price - buy_price) / buy_price) * 100

        # Append the calculated profit to the returns list
        returns.append(profit)

    # Return the list of profits (returns) calculated
    return returns
```

```
def plot(returns, ticker, start_date, end_date):
    # Download historical stock data using Yahoo Finance API
    data = yf.download(ticker, start_date, end_date)

    # Set up the figure size
    plt.figure(figsize=(10, 4))

    # Plot stock price
    plt.subplot(2, 1, 1)  # Subplot for stock price
    plt.plot(data['Close'], color='green')  # Plot closing prices
    plt.title(ticker, fontsize=16)  # Set title to ticker symbol
    plt.ylabel('Price ($)', fontsize=14)  # Set y-axis label

    # Plot stock returns
    plt.subplot(2, 1, 2)  # Subplot for stock returns
    plt.plot(returns, color='red')  # Plot returns
    plt.title(ticker + ' RETURNS', fontsize=16)  # Set title to ticker symbol + ' RETURNS'
    plt.ylabel('Pct. Return', fontsize=14)  # Set y-axis label for returns
    plt.axhline(0, color='k', linestyle='--')  # Add horizontal line at y=0 for reference

    plt.tight_layout()  # Adjust subplot parameters to give specified padding

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

```
seed(1)  # Setting a seed for reproducibility

pct = 0  # Minimum mean return threshold
volatility = 100  # Maximum standard deviation threshold

time = [5,10,15,20,25,30,35,40,45,50,55,60,65,70,75,80,85,90,95,100,105,110,115,120]  # List of time periods

dict = {} # Initialize an empty dictionary to store results
i = 1  # Initialize index for dictionary keys

# Loop to generate random start dates and analyze each ticker for each time period
for _ in range(20):  # Repeat 20 times
    # Generate a random start date within a range of 2 years from April 1, 2020
    start_date = dt.datetime(2020, 4, 1) + timedelta(days=randint(0, 730))
    end_date = dt.datetime(2023, 12, 1)  # End date fixed at December 1, 2023
```

```
  # Loop through each ticker in the list of tickers
  for ticker in tickers:
    # Loop through each time period in the list of time periods
    for x in time:
      # Calculate returns using the analysis function for the current ticker and time period
      returns = analysis(ticker, start_date, end_date, x)

      # Check conditions: mean return >= pct and standard deviation <= volatility
      if (st.mean(returns) >= pct and st.stdev(returns) <= volatility):
        mean = st.mean(returns)  # Calculate mean return
        std = st.stdev(returns)  # Calculate standard deviation of returns

        # Update dictionary with key-value pairs: index, time period, ticker, mean return, std deviation
        dict.update({i: [x, ticker, mean, std]})
        i = i + 1  # Increment index

# Convert dictionary to Pandas DataFrame, specifying column names
Test = pd.DataFrame.from_dict(dict, orient='index', columns=['Time', 'Ticker', 'Mean', 'Std'])
```

```
# List of time periods
time = [5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60, 65, 70, 75, 80, 85, 90, 95, 100, 105, 110, 115, 120]

# Initialize empty lists to store results
Mean = []
Std = []
Days = []

# Iterate through each time period
for t in time:
  Days.append(t)  # Append current time period to Days list

  # Calculate mean and round to 4 decimal places
  Mean.append(np.round(Test[Test['Time'] == t]['Mean'].mean(), 4))

  # Calculate standard deviation and round to 4 decimal places
  Std.append(np.round(Test[Test['Time'] == t]['Std'].mean(), 4))
```

```
# Create a figure and axis with a specified size
fig, ax = plt.subplots(figsize=(10, 5))

# Generate random colors for each data point
colors = np.random.rand(24)

# Set labels for x-axis and y-axis
plt.xlabel('Average Returns')
plt.ylabel('Volatility')

# Scatter plot with Mean as x-axis, Std as y-axis, and colors representing different data points
ax.scatter(Mean, Std, c=colors)
```

```
# Annotate each point with its corresponding time period (Days)
for i, label in enumerate(Days):
    ax.annotate(label,  # Text to display
                (Mean[i], Std[i]),  # Position of the annotation
                textcoords="offset points",  # Offset text position slightly
                xytext=(0,5),  # Position text vertically offset by 5 points
                ha='center')  # Center align the text horizontally


plt.savefig('C:/Users/ASUS ROG/Desktop/Stat 390/Returns.png')
plt.show()  # Display the plot
```

```
def annualised_return(mean_return, mean_volatility):
    """
    Calculates annualized return and annualized volatility.

    Parameters:
    - mean_return (float): Mean return over a period.
    - mean_volatility (float): Mean volatility (standard deviation) over a period.

    Returns:
    - annualised_return (float): Annualized return in percentage.
    - annualised_volatility (float): Annualized volatility (standard deviation) in percentage.
    """
    # Calculate annualized return using the formula: ((1 + mean_return)^12 - 1) * 100
    annualised_return = np.round(((1 + mean_return) ** 12 - 1) * 100, 2)

    # Calculate annualized volatility using the formula: mean_volatility * sqrt(12) * 100
    annualised_volatility = np.round((mean_volatility * np.sqrt(12)) * 100, 2)

    return annualised_return, annualised_volatility
```

```
def trend(choice, end_date):
    selected = []  # Initialize an empty list to store selected stocks

    # Calculate start date as 180 days before end_date
    start_date = end_date - timedelta(30)

    # Iterate through each stock ticker in the choice list
    for x in choice:
        # Download historical stock data for the current ticker from start_date to end_date
        data = yf.download(x, start_date, end_date).Close.reset_index()

        # Get the last closing price from the downloaded data
        last_price = data['Close'].iloc[-1]

        # Sort closing prices in descending order and select the top 5
        data = data['Close'].sort_values(ascending=False).head(3)

        # Calculate the average of the top 5 closing prices
```

```
    average = data.mean()

    # Compare the last closing price with the average of top 5 closing prices
    if last_price >= average:
      selected.append(x)  # Add the ticker to the selected list if condition is met

  return selected  # Return the list of selected tickers
```

```
pct = 5  # Minimum annualized return threshold in percentage
volatility = 30  # Maximum annualized volatility threshold in percentage
time = 30  # Time period for analysis in days
dict = {}  # Initialize an empty dictionary to store results
choice = []  # Initialize an empty list to store selected tickers
i = 1  # Initialize index for dictionary keys

start_date = dt.datetime(2020, 4, 1)  # Start date for analysis
end_date = dt.datetime(2022, 7, 1)  # End date for analysis

# Iterate through each ticker in the list of tickers
for ticker in tickers:
  # Calculate returns for the current ticker and time period
  returns = analysis(ticker, start_date, end_date, time)

  # Calculate mean return and mean volatility (converted to decimal)
  mean_return = st.mean(returns) / 100
  mean_volatility = st.stdev(returns) / 100

  # Calculate annualized return and annualized volatility
  annual_return, annual_volatility = annualised_return(mean_return, mean_volatility)

  # Check if annualized return meets the threshold and annualized volatility is within the threshold
  if (annual_return >= pct and annual_volatility <= volatility):
    # Plot returns for the current ticker
    plot(returns, ticker, start_date, end_date)

    # Update mean and std with annualized values
    mean = annual_return
    std = annual_volatility

    # Add ticker to choice list and update dictionary with results
    choice.append(ticker)
    dict.update({i: [ticker, mean, std]})
    i = i + 1

# Create a DataFrame from dictionary and sort by Mean and Std in descending order
Returns = pd.DataFrame.from_dict(dict, orient='index', columns=['Ticker', 'Mean', 'Std'])
Returns = Returns.sort_values(by=['Mean', 'Std'], ascending=False)

selected = trend(choice,end_date)
selected
```

```
pct = 5  # Minimum annualized return threshold in percentage
volatility = 80  # Maximum annualized volatility threshold in percentage
time = 30  # Time period for analysis in days
dict = {} # Initialize an empty dictionary to store results
choice = []  # Initialize an empty list to store selected tickersfor _ in range(1000):
 i = 1  # Initialize index for dictionary keys

start_date = dt.datetime(2013, 5, 24)  # Start date for analysis
end_date = dt.datetime(2014, 2, 24)  # End date for analysis

# Iterate through each ticker in the list of tickers
for ticker in tickers:
   # Calculate returns for the current ticker and time period
   returns = analysis(ticker, start_date, end_date, time)

   if not len(returns)==0:
      # Calculate mean return and mean volatility (converted to decimal)
      mean_return = st.mean(returns) / 100
      mean_volatility = st.stdev(returns) / 100

      # Calculate annualized return and annualized volatility
      annual_return, annual_volatility = annualised_return(mean_return, mean_volatility)

      # Check if annualized return meets the threshold and annualized volatility is within the threshold
      if (annual_return >= pct and annual_volatility <= volatility):
         # Plot returns for the current ticker
         plot(returns, ticker, start_date, end_date)

         # Update mean and std with annualized values
         mean = annual_return
         std = annual_volatility

         # Add ticker to choice list and update dictionary with results
         choice.append(ticker)
         dict.update({i: [ticker, mean, std]})
         i = i + 1

# Create a DataFrame from dictionary and sort by Mean and Std in descending order
Returns = pd.DataFrame.from_dict(dict, orient='index', columns=['Ticker', 'Mean', 'Std'])
Returns = Returns.sort_values(by=['Mean', 'Std'], ascending=False)
```

```
selected = trend(choice,end_date)
selected
```

Simulating Gap and Difference

```
count = 0  # Initialize a counter for buying opportunities
date1 = []  # Initialize empty lists to store dates for buying opportunities in condition 1
date2 = [] # # Initialize empty lists to store dates for buying opportunities in condition 2
val = [0.03,0.05,0.08,0.10,0.13,0.15,0.18,0.20] # Different values of gap and difference
```

```
dict = {}
i = 1
      # Download historical data for the ticker from start_date to end_date
      data = yf.download(ticker, start_date, end_date)
      data = data.reset_index()                    # Reset index of the downloaded data
seed(1)

# Loop 20 times to simulate different end dates
for _ in range(500):
   # Generate a random end date between April 1, 2021 and 1008 days later
   end_date = dt.datetime(2020, 4, 1) + timedelta(days=randint(0, 500))

   for x in val:

      for ticker in choice:
         # Download historical data for the ticker from 30 days before end_date to end_date
         data = yf.download(ticker, end_date - timedelta(30), end_date).reset_index()

         # Calculate the 21-day and 9-day moving averages
         ma_21 = data['Close'].rolling(window=21).mean().iloc[-1]  # 21-day MA
         ma_9 = data['Close'].rolling(window=9).mean().iloc[-1]   # 9-day MA

         # Get the closing price of the ticker on the last day of the data
         price = data['Close'].iloc[-1]

         # Calculate percentage differences and gaps relative to price
         diff_21 = ((ma_21 - ma_9) / price) * 100
         gap_21 = ((price - ma_21) / price) * 100 if price > ma_21 else ((ma_21 - price) / price) * 100

         diff_9 = ((ma_9 - ma_21) / price) * 100
         gap_9 = ((price - ma_9) / price) * 100 if price > ma_9 else ((ma_9 - price) / price) * 100

         # Check conditions for potential buying opportunities based on moving averages and gaps
         if ma_9 <= ma_21 and diff_9 < x and gap_9 < x:
            # Print ticker symbol and buy indication
            print(f"{ticker}: Buy")
            count += 1  # Increment count of buying opportunities
            date1.append(end_date)  # Store end_date in date1 list
            dict.update({i: [x, ticker, end_date]})
            i = i + 1  # Increment index
            break  # Exit inner loop

Sim = pd.DataFrame.from_dict(dict, orient='index', columns=['Val', 'Ticker', 'Date'])
Sim.to_csv('Gap & Diff.csv', index = False)
```

```
Trades = pd.read_csv('Gap & Diff.csv')

dict = {}
roi = []
i = 1
for x in val:
```

```
  data = Trades[Trades['Val'] == x]

  if not data.empty:

    for ticker in selected:

      for index in range(len(data)):

        start_date = dt.datetime.strptime(data['Date'].iloc[index],"%Y-%m-%d %H:%M:%S")
        end_date = start_date + timedelta(30)
        trade_data = yf.download(ticker,start_date,end_date)
        buy_price = trade_data['Close'].iloc[0]
        sell_price = trade_data['Close'].iloc[-1]
        profit = ((sell_price - buy_price)/buy_price) * 100
        roi.append(profit)

      mean_return = st.mean(roi)
      dict.update({i: [x, ticker, mean_return]})
      i = i + 1  # Increment index


Vals = pd.DataFrame.from_dict(dict, orient='index', columns=['Val', 'Ticker', 'Returns'])
Vals = Vals.sort_values(by = 'Returns',ascending = False)
Vals.to_csv('Returns on Gap & Diff.csv',index=False)
```

```
Vals = pd.read_csv('Returns on Gap & Diff.csv')
plt.figure(figsize=(8.5,4))
ax = sns.barplot(x = Vals['Val'], y = Vals['Returns'],errorbar = None)
ax.bar_label(ax.containers[0], fontsize=10);
```

The Algo

```
seed(1)
count = 0  # Initialize a counter for buying opportunities
date = []  # Initialize empty lists to store dates for buying opportunities
end_date = dt.datetime(2023, 4, 30)

# Loop 20 times to simulate different end dates
for _ in range(300):
  # Generate the next day after April 1, 2023
  end_date = end_date + timedelta(1)

  for ticker in selected:
      # Download historical data for the ticker from 30 days before end_date to end_date
      data = yf.download(ticker, end_date - timedelta(30), end_date).reset_index()

      # Calculate the 21-day and 9-day moving averages
      ma_21 = data['Close'].rolling(window=21).mean().iloc[-1]  # 21-day MA
      ma_9 = data['Close'].rolling(window=9).mean().iloc[-1]   # 9-day MA
```

```
        # Get the closing price of the ticker on the last day of the data
        price = data['Close'].iloc[-1]

        # Calculate percentage differences and gaps relative to price
        diff_9 = ((ma_9 - ma_21) / price) * 100
        gap_9 = ((price - ma_9) / price) * 100 if price > ma_9 else ((ma_9 - price) / price) * 100

        # Check conditions for potential buying opportunities based on moving averages and gaps
        if ma_9 <= ma_21 and diff_9 < 0.08 and gap_9 < 0.08:
            # Print ticker symbol and buy indication
            print(f"{ticker}: Buy ")
            count += 1  # Increment count of buying opportunities
            date.append(end_date)  # Store end_date in date1 list
            break  # Exit inner loop
```

Visualizing Trades

```
temp = []     # List to store profits
trades = []   # List to store trade results (1 for profit, 0 for loss)

# Iterate through each ticker in the 'choice' list
for ticker in selected:
    # Iterate through each date in the 'date' list
    for x in range(len(date)):
        start_date = date[x]                    # Get start date from 'date'
        end_date = start_date + timedelta(30)           # Calculate end date as 30 days after start_date

        # Download historical data for the ticker from start_date to end_date
        data = yf.download(ticker, start_date, end_date)
        data = data.reset_index()                   # Reset index of the downloaded data

        buy_price = data['Close'].iloc[0]             # Get buy price (close price on start_date)
        sell_price = data['Close'].iloc[-1]           # Get sell price (close price on end_date)

        profit = ((sell_price - buy_price) / buy_price) * 100 # Calculate profit percentage

        temp.append(profit)                     # Append profit to temp1 list

        # Determine if the trade resulted in a profit (1) or loss (0)
        if buy_price < sell_price:
            trades.append(1)  # Profit
        else:
            trades.append(0)  # Loss
```

```
# Calculate trade result counts
counter = Counter(trades)
categories = list(counter.keys())  # 0 for losses, 1 for wins
counts = list(counter.values())

# Create a figure with two subplots
```

```
fig, axs = plt.subplots(1, 2, figsize=(12, 4))

# Subplot 1: Bar chart of trade result counts
axs[0].bar(categories, counts, color=['skyblue', 'lightgreen'])
axs[0].set_title('Counts of Wins and Losses')
axs[0].set_ylabel('No. of Trades')
axs[0].set_xticks([0, 1])
axs[0].set_xticklabels(['Loss', 'Wins'])

# Subplot 2: Histogram of trade returns
axs[1].hist(temp, bins=7, color='lightcoral')
axs[1].set_title('Returns of Trades')
axs[1].set_xlabel('Returns')

# Adjust layout and display the plot
plt.tight_layout()
plt.show()

# Calculate and print the mean return
mean_return = st.mean(temp)  # Assuming st is a statistics library or similar
print(f"Mean return: {mean_return}")
counts
```

Statistical Fitting

```
from scipy import stats
from tabulate import tabulate

returns = temp

# Define a list of distributions to check
distributions = [stats.norm, stats.t, stats.laplace, stats.expon, stats.lognorm, stats.beta, stats.gamma, stats.weibull_min]

# List to store the results
results = []

# Fit each distribution and calculate the goodness-of-fit metrics
for distribution in distributions:
    # Fit the distribution to the data
    params = np.round(distribution.fit(returns),4)

    # Get the fitted PDF
    pdf_fitted = np.round(distribution.pdf(np.sort(returns), *params[:-2], loc=params[-2], scale=params[-1]),4)

    # Kolmogorov-Smirnov test
    D, p_value = np.round(stats.kstest(returns, distribution.cdf, args=params),4)

    # Akaike Information Criterion
```

```
    log_likelihood = np.sum(np.log(distribution.pdf(returns, *params[:-2], loc=params[-2], scale=params[-1])))
    AIC = np.round(2 * len(params) - 2 * log_likelihood,2)

    results.append({
        'Distribution': distribution.name,
        'KS D statistic': D,
        'p-value': p_value,
        'AIC': AIC,
        'Parameters': params
    })

# Create a DataFrame from the results
results_df = pd.DataFrame(results)

# Print the results as a table
print(tabulate(results_df, headers='keys', tablefmt='pretty', showindex=False))

# Determine the best-fitting distribution based on AIC
best_fit = results_df.loc[results_df['AIC'].idxmin()]['Distribution']
print(f'\nBest fitting distribution: {best_fit}')

# Plot the data and the best-fitting distribution
best_fit_params = results_df.loc[results_df['Distribution'] == best_fit, 'Parameters'].values[0]
best_distribution = getattr(stats, best_fit)
pdf_fitted = best_distribution.pdf(np.sort(returns), *best_fit_params[:-2], loc=best_fit_params[-2], scale=best_fit_params[-1])

plt.figure(figsize=(8,4))
plt.hist(returns, bins=7, density=True, alpha=0.6, color='g')
plt.plot(np.sort(returns), pdf_fitted, 'r-', label=f'{best_fit} PDF')
plt.legend()
plt.xlabel('Returns')
plt.ylabel('Density')
plt.title('Fitted Distributions to Algorithm Returns')
plt.show()
```

Simulating Trading Account

```
from scipy.stats import weibull_min

# Set initial account balance and Weibull distribution parameters
initial_balance = 100
shape, loc, scale = 1.8918, -6.3144, 9.4235
win_rate = 0.5714  # 57.14% win rate

# Define the number of trading days (3 years of trading days)
num_years = 1
num_days = num_years * 252  # Approx. 252 trading days in a year

# Number of simulations to run
```

```
num_simulations = 20

# Initialize a list to store all simulated account balances
all_simulations = []

# Simulate account balance over time for each run
for _ in range(num_simulations):
  # Generate daily returns from the Weibull distribution with specified win rate
  scale_adjusted = scale / win_rate
  daily_returns = weibull_min.rvs(shape, loc, scale_adjusted, size=num_days) / 100
  negative_returns_indices = np.random.choice(num_days, size=int(num_days * (1 - win_rate)), replace=False)
  daily_returns[negative_returns_indices] *= -1

  # Initialize account balance array
  account_balance = np.zeros(num_days)
  account_balance[0] = initial_balance

  # Simulate account balance over time
  for i in range(1, num_days):
    account_balance[i] = account_balance[i-1] * (1 + daily_returns[i])

  # Store the account balance simulation in the list
  all_simulations.append(account_balance)

# Convert the list of simulations into a DataFrame
dates = pd.date_range(start='2024-06-27', periods=num_days, freq='B')
simulation_df = pd.DataFrame(all_simulations, index=[f'Simulation {i+1}' for i in range(num_simulations)]).transpose()
simulation_df['Date'] = dates

# Calculate mean of all simulations
mean_balance = simulation_df.iloc[:, :num_simulations].mean(axis=1)

# Plot all simulations and the mean on the same line graph
plt.figure(figsize=(16, 5))
for i in range(num_simulations):
  plt.plot(simulation_df['Date'], simulation_df[f'Simulation {i+1}'], label=f'Simulation {i+1}')

plt.plot(simulation_df['Date'], mean_balance, color='black', linestyle='--', linewidth=2, label='Mean of Simulations')

plt.xlabel('Date')
plt.ylabel('Account Balance ($)')
plt.title('Simulated Trading Account Balance Over One year')
plt.legend(loc='center left', bbox_to_anchor=(1, 0.5))  # Place legend on the left side
plt.grid(True)
plt.show()
```