In [2]:

```
!pip install backtesting
import yfinance as yf

import pandas as pd
from backtesting import Backtest, Strategy
from backtesting.lib import crossover
```

```
Requirement already satisfied: backtesting in /usr/local/lib/python3.10/dist-packages (0.
3.3)
Requirement already satisfied: numpy>=1.17.0 in /usr/local/lib/python3.10/dist-packages (
from backtesting) (1.25.2)
Requirement already satisfied: pandas!=0.25.0,>=0.25.0 in /usr/local/lib/python3.10/dist-
packages (from backtesting) (1.5.3)
Requirement already satisfied: bokeh>=1.4.0 in /usr/local/lib/python3.10/dist-packages (f
rom backtesting) (3.3.4)
Requirement already satisfied: Jinja2>=2.9 in /usr/local/lib/python3.10/dist-packages (fr
om bokeh>=1.4.0->backtesting) (3.1.3)
Requirement already satisfied: contourpy>=1 in /usr/local/lib/python3.10/dist-packages (f
rom bokeh>=1.4.0->backtesting) (1.2.0)
Requirement already satisfied: packaging>=16.8 in /usr/local/lib/python3.10/dist-packages
(from bokeh>=1.4.0->backtesting) (23.2)
Requirement already satisfied: pillow>=7.1.0 in /usr/local/lib/python3.10/dist-packages (
from bokeh>=1.4.0->backtesting) (9.4.0)
Requirement already satisfied: PyYAML>=3.10 in /usr/local/lib/python3.10/dist-packages (f
rom bokeh>=1.4.0->backtesting) (6.0.1)
Requirement already satisfied: tornado>=5.1 in /usr/local/lib/python3.10/dist-packages (f
rom bokeh>=1.4.0->backtesting) (6.3.3)
Requirement already satisfied: xyzservices>=2021.09.1 in /usr/local/lib/python3.10/dist-p
ackages (from bokeh>=1.4.0->backtesting) (2023.10.1)
Requirement already satisfied: python-dateutil>=2.8.1 in /usr/local/lib/python3.10/dist-p
ackages (from pandas!=0.25.0,>=0.25.0->backtesting) (2.8.2)
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.10/dist-packages (f
rom pandas!=0.25.0,>=0.25.0->backtesting) (2023.4)
Requirement already satisfied: MarkupSafe>=2.0 in /usr/local/lib/python3.10/dist-packages
(from Jinja2>=2.9->bokeh>=1.4.0->backtesting) (2.1.5)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.10/dist-packages (from
python-dateutil>=2.8.1->pandas!=0.25.0,>=0.25.0->backtesting) (1.16.0)
```

```
/usr/local/lib/python3.10/dist-packages/backtesting/_plotting.py:50: UserWarning: Jupyter
Notebook detected. Setting Bokeh output to notebook. This may not work in Jupyter clients
without JavaScript support (e.g. PyCharm, Spyder IDE). Reset with `backtesting.set_bokeh_
output(notebook=False)`.
  warnings.warn('Jupyter Notebook detected. '
```

In [3]:

```
"""
import re

def convert_treasury_price(price_str):
    # Check for non-applicable values and return None or a default value
    if price_str in ["", "NA", None]:
        return None

    # Split the price string into whole and fractional parts
    parts = price_str.split('-')

    # If there is no hyphen, the price is already a whole number
    if len(parts) == 1:
        return float(parts[0])

    whole_number = int(parts[0])  # The whole number part of the price
    fraction_part = parts[1]  # The fractional part of the price

    # Check for a plus sign indicating an additional 1/64
    plus_sign = '+' in fraction_part
    fraction_part = fraction_part.replace('+', '')
```

```
    # Convert the fraction into decimal
    numerator = int(fraction_part) if fraction_part.isdigit() else 0
    denominator = 32  # Fractions are usually out of 32 for Treasury prices
    fraction_decimal = numerator / denominator

    # Add additional 1/64 if there was a plus sign
    if plus_sign:
        fraction_decimal += 1/64

    # Combine the whole number and fractional decimal
    decimal_price = whole_number + fraction_decimal

    return decimal_price

import pandas as pd
data = pd.read_csv('/content/Final Version of Combined Data -FRE 7841.csv')

treasury_columns = [
    'WN1 (UST 30y)',
    'TY1 (10y US Treasury)',
    'FV1 (5y US Treasury)',
    'TU1 (2y US Treasury)'
]

# Convert the treasury prices to float for each specified column
for column in treasury_columns:
    data[column] = data[column].apply(convert_treasury_price)

# Display the dataframe to confirm the conversion
data[treasury_columns]
data.to_csv('/content/Final Version of Combined Data -FRE 7841.csv')
"""
```

Out[3]:

```
'\nimport re\n\ndef convert_treasury_price(price_str):\n    # Check for non-applicable va
lues and return None or a default value\n    if price_str in ["", "NA", None]:\n        r
eturn None\n\n    # Split the price string into whole and fractional parts\n    parts = p
rice_str.split(\'-\')\n\n    # If there is no hyphen, the price is already a whole number
\n    if len(parts) == 1:\n        return float(parts[0])\n\n    whole_number = int(parts
[0])  # The whole number part of the price\n    fraction_part = parts[1]  # The fractiona
l part of the price\n\n    # Check for a plus sign indicating an additional 1/64\n    plu
s_sign = \'+\' in fraction_part\n    fraction_part = fraction_part.replace(\'+\', \'\')\n
\n    # Convert the fraction into decimal\n    numerator = int(fraction_part) if fraction
_part.isdigit() else 0\n    denominator = 32  # Fractions are usually out of 32 for Treas
ury prices\n    fraction_decimal = numerator / denominator\n\n    # Add additional 1/64 i
f there was a plus sign\n    if plus_sign:\n        fraction_decimal += 1/64\n\n    # Com
bine the whole number and fractional decimal\n    decimal_price = whole_number + fraction
_decimal\n\n    return decimal_price\n\nimport pandas as pd\ndata = pd.read_csv(\'/conten
t/Final Version of Combined Data -FRE 7841.csv\')\n\ntreasury_columns = [\n    \'WN1 (UST
30y)\',\n    \'TY1 (10y US Treasury)\',\n    \'FV1 (5y US Treasury)\',\n    \'TU1 (2y US
Treasury)\'\n]\n\n# Convert the treasury prices to float for each specified column\nfor c
olumn in treasury_columns:\n    data[column] = data[column].apply(convert_treasury_price)
\n\n# Display the dataframe to confirm the conversion\ndata[treasury_columns]\ndata.to_cs
v(\'/content/Final Version of Combined Data -FRE 7841.csv\')\n'
```

1. Extract data of each asset type

In [4]:

```
!pip install pandas openpyxl

import pandas as pd
import numpy as np

excel_path = '/content/financial_data_completed.xlsx'
xls = pd.ExcelFile(excel_path)
ticker_list = xls.sheet_names  # Assuming you're interested in all sheet names
```

```
# Define your date range
start_date = '2015-01-02'
end_date = '2023-12-31'

# Initialize an empty list to store processed DataFrames
processed_data = []

# Iterate over each sheet name, process, and store the result
for ticker in ticker_list:
    df = pd.read_excel(xls, sheet_name=ticker, parse_dates=['Date'])
    df.set_index('Date', inplace=True)

    # Filter the DataFrame to the specified date range
    df = df.loc[start_date:end_date]

    # Create a new index that includes all business days in the range, for backfilling
    new_index = pd.date_range(start_date, end_date, freq='B')
    df = df.reindex(new_index)

    # Backfill missing data
    df.fillna(method='ffill', inplace=True)

    # Store the processed DataFrame
    processed_data.append(df)

# At this point, `processed_data` contains all your processed DataFrames
data = processed_data
```

```
Requirement already satisfied: pandas in /usr/local/lib/python3.10/dist-packages (1.5.3)
Requirement already satisfied: openpyxl in /usr/local/lib/python3.10/dist-packages (3.1.2
)
Requirement already satisfied: python-dateutil>=2.8.1 in /usr/local/lib/python3.10/dist-p
ackages (from pandas) (2.8.2)
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.10/dist-packages (f
rom pandas) (2023.4)
Requirement already satisfied: numpy>=1.21.0 in /usr/local/lib/python3.10/dist-packages (
from pandas) (1.25.2)
Requirement already satisfied: et-xmlfile in /usr/local/lib/python3.10/dist-packages (fro
m openpyxl) (1.1.0)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.10/dist-packages (from
python-dateutil>=2.8.1->pandas) (1.16.0)
```

In [5]:

```
#intersection = set(date[0]).intersection(*date[1:])

#intersection = list(intersection).sorted() # This will display the common elements in al
l sub-lists
```

In [6]:

```
import pandas as pd

def get_data(data):
    train_ls = []
    test_ls = []

    # Check if each item in data is a DataFrame and proceed if true
    for df in data:  # Directly iterate over DataFrames in 'data'
        if isinstance(df, pd.DataFrame):
            # Assuming the 'Date' is the index after previous processing, no need to rese
t it
            df.index = pd.to_datetime(df.index)  # Ensure index is datetime

            # Initialize containers for each asset's train and test DataFrames
            d_train = []
            d_test = []

            # Assuming 'df' is structured with datetime index and has been filtered to th
e date range
            for j in range(0, len(df)-252, 252):  # Define rolling windows
```

```
                # Ensure we have enough data for both training and testing in each windo
w
                if len(df)-252-j >= 252:
                    # Slice DataFrames for training and testing
                    train = df.iloc[j:j+252]
                    test = df.iloc[j+252:j+504]

                    d_train.append(train)
                    d_test.append(test)

            # Append training and testing sets for the current asset to the lists
            train_ls.append(d_train)
            test_ls.append(d_test)
        else:
            print("Error: One or more items in 'data' are not pandas DataFrame objects."
)

    return train_ls, test_ls


train, test = get_data(data)
```

In [7]:

```
test_eq=test[0:23]
test_fi=test[23:26]
test_fx=test[26:39]
test_cmdt=test[39:47]

train_eq=train[0:23]
train_fi=train[23:26]
train_fx=train[26:39]
train_cmdt=train[39:47]
test=[test_eq,test_fi,test_fx,test_cmdt]
train=[train_eq, train_fi,train_fx, train_cmdt]
```

In [8]:

```
def switch_dimensions(data):
    # Initialize a new list to store the reorganized data
    new_data = []

    # Loop through each sublist i in the original list
    for i in range(len(data)):
        # Initialize a temporary list to store the switched dimensions for the current i
        temp_i = []

        # Determine the new dimensions based on the original dimensions j and k
        # It's assumed all sublists at level j have the same length, and similarly for le
vel k
        len_j = len(data[i][0])  # Length of the first j dimension (assuming uniform len
gth across all j)
        len_k = len(data[i])     # Length of the k dimension

        # Loop through each new j index (originally k) to reorganize the elements
        for j in range(len_j):
            temp_j = []
            # Loop through each new k index (originally j)
            for k in range(len_k):
                # Append the element from the original data, switching j and k positions
                temp_j.append(data[i][k][j])
            # Append the reorganized sublist for the current i to temp_i
            temp_i.append(temp_j)

        # Append the reorganized data for the current i to the new_data list
        new_data.append(temp_i)

    return new_data

# Assuming 'test' is your list with the structure test[i][j][k]
test = switch_dimensions(test)
```

```
train = switch_dimensions(train)


# 'new_test' now has dimensions j and k switched for every i
```

In [9]:

```
test_data = []
for i in range(len(test)):
  ls1 = []
  for j in range(len(test[0])):
    ls2 = []
    for z in range(3):
      for y in range(len(test[i][0])):
        ls2.append(test[i][j][y])
    ls1.append(ls2)
  test_data.append(ls1)
```

2. Build Strategies of SMA, RSI, MACD

SMA

In [10]:

```
import pandas as pd
def SMA(values, n):
    """
    Return simple moving average of `values`, at
    each step taking into account `n` previous values.
    """
    return pd.Series(values).rolling(n).mean()

class SmaCross(Strategy):
  def init(self,n1=10,n2=20):
      # Precompute the two moving averages
      self.sma1 = self.I(SMA, self.data.Close, self.n1)
      self.sma2 = self.I(SMA, self.data.Close, self.n2)

  def next(self):
      # If sma1 crosses above sma2, close any existing and short trades, and buy the as
set
      if crossover(self.sma1, self.sma2):
          self.position.close()
          self.buy()

      # Else, if sma1 crosses below sma2, close any existing and long trades, and sell
the asset
      elif crossover(self.sma2, self.sma1):
          self.position.close()
          self.sell()

class CustomSmaCross(SmaCross):
    n1 = 30
    n2 = 40
```

MACD

In [11]:

```
import pandas as pd
def crossunder(series1, series2):
    """
    Returns True if series1 crosses under series2, False otherwise.
    """
    return crossover(series2, series1)
# Assuming crossover and crossunder functions are defined as before
```

```
def MACD(values, fast_period, slow_period, signal_period):
    """
    Calculate the Moving Average Convergence Divergence (MACD) indicator, its signal line
, and histogram.
    """
    exp1 = pd.Series(values).ewm(span=fast_period, adjust=False).mean()
    exp2 = pd.Series(values).ewm(span=slow_period, adjust=False).mean()
    macd_line = exp1 - exp2
    signal_line = macd_line.ewm(span=signal_period, adjust=False).mean()
    histogram = macd_line - signal_line
    return macd_line, signal_line, histogram

class MACDStrategy(Strategy):
    def __init__(self, *args, **kwargs):
        self.fast_period = kwargs.pop('fast_period', 12)
        self.slow_period = kwargs.pop('slow_period', 26)
        self.signal_period = kwargs.pop('signal_period', 9)
        super().__init__(*args, **kwargs)

    def init(self):
        # Calculate MACD indicator using parameters
        self.macd, self.signal, _ = self.I(MACD, self.data.Close, self.fast_period, self
.slow_period, self.signal_period)

    def next(self):
        if not self.position:
            # If MACD line crosses above Signal line, buy
            if crossover(self.macd, self.signal):
                self.buy()
        else:
            # If MACD line crosses below Signal line, sell
            if crossunder(self.macd, self.signal):
                self.sell()


class CustomMACD(MACDStrategy):
  fast_period=5
  slow_period=20
  signal_period=2
```

RSI

In [12]:

```
def RSI(values, n):
    """
    Return Relative Strength Index (RSI) of `values`, at
    each step taking into account `n` previous values.
    """
    deltas = pd.Series(values).diff(1)
    gain = (deltas.where(deltas > 0, 0)).rolling(n).mean()
    loss = (-deltas.where(deltas < 0, 0)).rolling(n).mean()
    rs = gain / loss
    rsi = 100 - (100 / (1 + rs))
    return rsi

class BasicRsiStrategy(Strategy):
    n1 = 30
    n2 = 80

    def __init__(self, *args, **kwargs):
        # Call the parent class's __init__ with all arguments
        super().__init__(*args, **kwargs)

    def init(self):
        # Framework-specific initialization code here
        self.RSI = self.I(RSI, self.data.Close, 14)

    def next(self):
```

```
        if len(self.RSI) < 2:  # Ensure there are at least two RSI values to compare
            return

        today = self.RSI[-1]
        yesterday = self.RSI[-2]

        if yesterday > self.n1 and today < self.n1 and not self.position.is_long:
            self.buy()
        elif yesterday < self.n2 and today > self.n2 and self.position.size > 0:
            self.position.close()

class CustomRsiStrategy(BasicRsiStrategy):
  n1 = 25
  n2 = 70
```

2.1 Grid Search optimal parameter

In [20]:

```
param_grid_sma = {'n1': range(3, 23, 10), 'n2': range(10, 90, 40)}
param_grid_rsi = {'n1': range(10, 40, 15), 'n2': range(60, 90, 15)}
param_grid_macd = {
    'fast_period': range(5, 20, 5),   # Example range, adjust as necessary
    'slow_period': range(20, 50, 15), # Example range, adjust as necessary
    'signal_period': range(10, 30, 10) # Example range, adjust as necessary
}



n_One_sma=[]
n_Two_sma=[]
res_sma = []
p_win_sma_ls=[]
bp_sma_ls = []

n_One_rsi=[]
n_Two_rsi=[]
res_rsi = []
p_win_rsi_ls=[]
bp_rsi_ls=[]


res_macd = []
p_win_macd_ls=[]
bp_macd_ls=[]
fast = []
slow = []
signal = []


for i in range(len(train)):
  for j in range(len(train[0])):
    n1_sma = []
    n2_sma = []
    res_Array_sma = []
    p_win_sma = []
    bp_sma = []

    fast_ls = []
    slow_ls = []
    signal_ls = []
    res_Array_macd = []
    p_win_macd = []
    bp_macd = []

    n1_rsi = []
    n2_rsi = []
    res_Array_rsi = []
    p_win_rsi = []
    bp_rsi = []
```

```
    for t in range(len(train[i][0])):
      t = 2
      # SMA
      print([i,j,t])
      bt = Backtest(train[i][j][t], CustomSmaCross, cash= 10000, commission= 0.002)
      res = bt.optimize(**param_grid_sma)
      bt.plot()
      n1_sma.append(res['_strategy'].n1)
      n2_sma.append(res['_strategy'].n2)
      res_Array_sma.append(res)
      if 'Win Rate [%]' in res:
        p_win_sma.append(res['Win Rate [%]'] / 100)
      else:
        p_win_sma.append(0)   # Or you can choose a default value

        # Similarly, check for other keys you are accessing
      if 'Avg. Trade [%]' in res:
        bp_sma.append(1 + res["Avg. Trade [%]"] / 100)
      else:
        bp_sma.append(-1)   # Or you can choose a default value

      # MACD
      bt = Backtest(train[i][j][t], CustomMACD, cash= 10000, commission= 0.002)
      res = bt.optimize(**param_grid_macd)
      bt.plot()
      fast_ls.append(res['_strategy'].fast_period)
      slow_ls.append(res['_strategy'].slow_period)
      signal_ls.append(res['_strategy'].signal_period)
      res_Array_macd.append(res)
      if 'Win Rate [%]' in res:
        p_win_macd.append(res['Win Rate [%]'] / 100)
      else:
        p_win_macd.append(0)   # Or you can choose a default value

        # Similarly, check for other keys you are accessing
      if 'Avg. Trade [%]' in res:
        bp_macd.append(1 + res["Avg. Trade [%]"] / 100)
      else:
        bp_macd.append(-1)   # Or you can choose a default value


      # RSI
      bt = Backtest(train[i][j][t], CustomRsiStrategy, cash= 10000, commission= 0.002)
      res = bt.optimize(**param_grid_rsi)
      bt.plot()
      n1_rsi.append(res['_strategy'].n1)
      n2_rsi.append(res['_strategy'].n2)
      res_Array_rsi.append(res)
      if 'Win Rate [%]' in res:
        p_win_rsi.append(res['Win Rate [%]'] / 100)
      else:
        p_win_rsi.append(0)   # Or you can choose a default value

        # Similarly, check for other keys you are accessing
      if 'Avg. Trade [%]' in res:
        bp_rsi.append(1 + res["Avg. Trade [%]"] / 100)
      else:
        bp_rsi.append(-1)   # Or you can choose a default value

    n_One_sma.append(n1_sma)
    n_Two_sma.append(n2_sma)
    res_sma.append(res_Array_sma)
    p_win_sma_ls.append(p_win_sma)
    bp_sma_ls.append(bp_sma)

    n_One_rsi.append(n1_rsi)
    n_Two_rsi.append(n2_rsi)
    res_rsi.append(res_Array_rsi)
    p_win_rsi_ls.append(p_win_rsi)
    bp_rsi_ls.append(bp_rsi)
```

```
    res_macd.append(res_Array_macd)
    p_win_macd_ls.append(p_win_macd)
    bp_macd_ls.append(bp_macd)
    fast.append(fast_ls)
    slow.append(slow_ls)
    signal.append(signal_ls)
```

```
[0, 0, 2]
```

```
BokehDeprecationWarning: Passing lists of formats for DatetimeTickFormatter scales was de
precated in Bokeh 3.0. Configure a single string format for each scale
/usr/local/lib/python3.10/dist-packages/backtesting/_plotting.py:250: UserWarning: Dateti
meFormatter scales now only accept a single format. Using the first provided: '%d %b'
  formatter=DatetimeTickFormatter(days=['%d %b', '%a %d'],
BokehDeprecationWarning: Passing lists of formats for DatetimeTickFormatter scales was de
precated in Bokeh 3.0. Configure a single string format for each scale
/usr/local/lib/python3.10/dist-packages/backtesting/_plotting.py:250: UserWarning: Dateti
meFormatter scales now only accept a single format. Using the first provided: '%m/%Y'
  formatter=DatetimeTickFormatter(days=['%d %b', '%a %d'],
/usr/local/lib/python3.10/dist-packages/backtesting/_plotting.py:659: UserWarning: found
multiple competing values for 'toolbar.active_drag' property; using the latest value
  fig = gridplot(
/usr/local/lib/python3.10/dist-packages/backtesting/_plotting.py:659: UserWarning: found
multiple competing values for 'toolbar.active_scroll' property; using the latest value
  fig = gridplot(
```

```
BokehDeprecationWarning: Passing lists of formats for DatetimeTickFormatter scales was de
precated in Bokeh 3.0. Configure a single string format for each scale
/usr/local/lib/python3.10/dist-packages/backtesting/_plotting.py:250: UserWarning: Dateti
meFormatter scales now only accept a single format. Using the first provided: '%d %b'
  formatter=DatetimeTickFormatter(days=['%d %b', '%a %d'],
BokehDeprecationWarning: Passing lists of formats for DatetimeTickFormatter scales was de
precated in Bokeh 3.0. Configure a single string format for each scale
/usr/local/lib/python3.10/dist-packages/backtesting/_plotting.py:250: UserWarning: Dateti
meFormatter scales now only accept a single format. Using the first provided: '%m/%Y'
  formatter=DatetimeTickFormatter(days=['%d %b', '%a %d'],
/usr/local/lib/python3.10/dist-packages/backtesting/_plotting.py:659: UserWarning: found
multiple competing values for 'toolbar.active_drag' property; using the latest value
  fig = gridplot(
/usr/local/lib/python3.10/dist-packages/backtesting/_plotting.py:659: UserWarning: found
multiple competing values for 'toolbar.active_scroll' property; using the latest value
  fig = gridplot(
```

```
BokehDeprecationWarning: Passing lists of formats for DatetimeTickFormatter scales was de
precated in Bokeh 3.0. Configure a single string format for each scale
/usr/local/lib/python3.10/dist-packages/backtesting/_plotting.py:250: UserWarning: Dateti
meFormatter scales now only accept a single format. Using the first provided: '%d %b'
  formatter=DatetimeTickFormatter(days=['%d %b', '%a %d'],
BokehDeprecationWarning: Passing lists of formats for DatetimeTickFormatter scales was de
precated in Bokeh 3.0. Configure a single string format for each scale
/usr/local/lib/python3.10/dist-packages/backtesting/_plotting.py:250: UserWarning: Dateti
meFormatter scales now only accept a single format. Using the first provided: '%m/%Y'
  formatter=DatetimeTickFormatter(days=['%d %b', '%a %d'],
/usr/local/lib/python3.10/dist-packages/backtesting/_plotting.py:659: UserWarning: found
multiple competing values for 'toolbar.active_drag' property; using the latest value
  fig = gridplot(
/usr/local/lib/python3.10/dist-packages/backtesting/_plotting.py:659: UserWarning: found
multiple competing values for 'toolbar.active_scroll' property; using the latest value
  fig = gridplot(
```

In [14]:

```
ls = [n_One_sma, n_Two_sma, res_sma, p_win_sma_ls, bp_sma_ls, n_One_rsi, n_Two_rsi, res_
rsi, p_win_rsi_ls, bp_rsi_ls, res_macd, p_win_macd_ls, bp_macd_ls, fast, slow, signal]
```

```
!pip install dill
import dill as pickle
with open("result_train.pkl",'wb') as f:
  pickle.dump(ls, f)
"""
!pip install dill
import dill as pickle
file_path = '/content/result_train.pkl'
with open(file_path, 'rb') as file:
    ls = pickle.load(file)
(n_One_sma, n_Two_sma, res_sma, p_win_sma_ls, bp_sma_ls,
n_One_rsi, n_Two_rsi, res_rsi, p_win_rsi_ls, bp_rsi_ls,
res_macd, p_win_macd_ls, bp_macd_ls, fast, slow, signal) = ls
"""
```

```
Requirement already satisfied: dill in /usr/local/lib/python3.10/dist-packages (0.3.8)
```

Out[14]:

```
"\n!pip install dill\nimport dill as pickle\nfile_path = '/content/result_train.pkl'\nwit
h open(file_path, 'rb') as file:\n    ls = pickle.load(file)\n(n_One_sma, n_Two_sma, res_
sma, p_win_sma_ls, bp_sma_ls,\nn_One_rsi, n_Two_rsi, res_rsi, p_win_rsi_ls, bp_rsi_ls,\nr
es_macd, p_win_macd_ls, bp_macd_ls, fast, slow, signal) = ls\n"
```

2.2 Calculate Kelly Criterion

In [15]:

```
def Calc_KC(bp, p_win, c):
  # input: result of bp, p_win
  KC = []
  for i in range(len(bp)):
    kc = (bp[i]*p_win[i]-(1-p_win[i])-c/10000)/bp[i]
    KC.append(kc)
  return KC

bp_all = [bp_sma_ls, bp_macd_ls, bp_rsi_ls]
p_win_all = [p_win_sma_ls, p_win_macd_ls, p_win_rsi_ls]
KC_eqt = []
KC_fi = []
KC_fx = []
KC_comd = []
for w in range(len(bp_all[0])):
  kc = []
  for s in range(3):
    kc_value = Calc_KC(bp_all[s][w], p_win_all[s][w], 100)
    kc.extend(kc_value)
  # Set negative elements to zero
  kc = [x if x > 0 else 0 for x in kc]
  #print(KC)
  # Sum of positive elements
  positive_sum = sum(kc)
  #print(KC)
  # Normalize positive elements to sum up to 1
  kc = [x / positive_sum for x in kc]
  print(len(kc))
  print(w)
  if w+1 <= 8:
    KC_eqt.append(kc)
  elif w+1 <= 16:
    KC_fi.append(kc)
  elif w+1 <= 24:
    KC_fx.append(kc)
  elif w+1 <= 32:
    KC_comd.append(kc)
```

```
3
0
```

3. Testing period

3. Testing period

In [17]:

```
res_sma_tls = []
res_rsi_tls = []
res_macd_tls = []

for i in range(len(test)):
  for j in range(len(test[0])):
    res_test_sma = []
    res_test_macd = []
    res_test_rsi = []
    if (i==0) and (j==0):
      w = 0
    else:
      w = w+1
    for t in range(len(test[i][0])):
      # SMA
      print(w)
      print(t)
      print('/n')
      class CustomStrategy(SmaCross):
        n1 = n_One_sma[w][t]
        n2 = n_Two_sma[w][t]

      bt = Backtest(test[i][j][t], CustomStrategy, cash=10000, commission=.000)
      stats = bt.run()
      res_test_sma.append(stats)

      # MACD
      class CustomMACD(MACDStrategy):
        fast_period=fast[w][t]
        slow_period=slow[w][t]
        signal_period=signal[w][t]

      bt = Backtest(test[i][j][t], CustomMACD, cash=10000, commission=.000)
      stats = bt.run()
      res_test_macd.append(stats)


      # RSI
      class CustomRsiStrategy(BasicRsiStrategy):
        n1 = n_One_rsi[w][t]
        n2 = n_Two_rsi[w][t]
      bt = Backtest(test[i][j][t], CustomRsiStrategy, cash=10000, commission=.000)
      stats = bt.run()
      res_test_rsi.append(stats)

    res_sma_tls.append(res_test_sma)
    res_rsi_tls.append(res_test_rsi)
    res_macd_tls.append(res_test_macd)
```

```
0
0
/n
0
1
/n

---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-17-d22190604390> in <cell line: 5>()
     17       print(t)
     18       print('/n')
---> 19       class CustomStrategy(SmaCross):
     20         n1 = n_One_sma[w][t]
     21         n2 = n_Two_sma[w][t]

<ipython-input-17-d22190604390> in CustomStrategy()
     18       print('/n')
     19       class CustomStrategy(SmaCross):
```

```
---> 20         n1 = n_One_sma[w][t]
     21         n2 = n_Two_sma[w][t]
     22

IndexError: list index out of range
```

In [ ]:

```
res_tls = [res_sma_tls, res_macd_tls, res_rsi_tls]
res_ls = [] #4*9*69*252
for i in range(len(test)):
  temp = []
  for j in range(len(test[0])):
    res_p = []
    for s in range(3):
      for t in range(len(test[i][0])):
        res_p.append(res_tls[s][j][t])
    temp.append(res_p)
  res_ls.append(temp)
```

Normalize position

In [ ]:

```
import pandas as pd

def normalize_position_sizes(data, price_data):
    # Convert EntryTime and ExitTime to datetime objects
    data['EntryTime'] = pd.to_datetime(data['EntryTime'])
    data['ExitTime'] = pd.to_datetime(data['ExitTime'])

    # Get the date range from price_data
    date_list = price_data.index

    # Initialize a list to store position sizes
    position_size_list = []

    # Iterate over each date
    for date in date_list:
        # Filter trades that are active on this date
        active_trades = data[(data['EntryTime'] <= date) & (data['ExitTime'] >= date)]

        # Calculate the total size of active trades on this date
        total_size = active_trades['Size'].sum()

        # Append the total size to the position size list
        position_size_list.append(total_size)

    # Normalize position sizes
    return position_size_list
```

Calculate DailyPNL

In [ ]:

```
import pandas as pd

def calculate_daily_pnl(trades_data, daily_close_prices):
    # Convert trades data and daily prices into DataFrames
    trades_df = pd.DataFrame(trades_data)
    close_prices_df = pd.DataFrame(daily_close_prices, columns=['Close'])
    close_prices_df.index = pd.to_datetime(close_prices_df.index)

    # Ensure trades' dates are in datetime format
    trades_df['EntryTime'] = pd.to_datetime(trades_df['EntryTime'])
    trades_df['ExitTime'] = pd.to_datetime(trades_df['ExitTime'])
```

```
    # Initialize a DataFrame for daily P&L
    daily_pl_df = pd.DataFrame(index=close_prices_df.index, columns=['DailyPnL'])
    daily_pl_df['DailyPnL'] = 0.0

    for index, trade in trades_df.iterrows():
        # Find dates within the trade period
        trade_period_dates = daily_pl_df.index[(daily_pl_df.index >= trade['EntryTime'])
& (daily_pl_df.index <= trade['ExitTime'])]

        for trade_date in trade_period_dates:
            if trade_date in close_prices_df.index:
                closing_price_on_date = close_prices_df.loc[trade_date, 'Close']

                # Find yesterday's date
                prev_date = trade_date - pd.Timedelta(days=1)

                # Check if previous date exists in the data
                if prev_date in close_prices_df.index:
                    prev_closing_price = close_prices_df.loc[prev_date, 'Close']

                    if prev_closing_price != 0:  # Avoid division by zero
                        # Calculate PNL as a percentage of yesterday's price, adjusted f
or the trade direction
                        pnl_percent = ((closing_price_on_date - prev_closing_price) / pr
ev_closing_price) * 100

                        # Adjust PNL based on the direction (sign) of the trade size
                        pnl_adjusted_for_direction = pnl_percent * (trade['Size'] / abs(
trade['Size']))

                        # Update daily P&L
                        daily_pl_df.at[trade_date, 'DailyPnL'] += pnl_adjusted_for_direc
tion

    return daily_pl_df
```

In [ ]:

```
def CalTransactCost(tradesData,CostInBP,priceData,defaultCapital,KC_):
  Cost=[]
  for i in range(len(tradesData)):
    temp=[]
    for j in range(len(tradesData[0])):
      temp.append((tradesData[i][j]*priceData[i]['Close'][j]*CostInBP*100*KC_[i][j])/def
aultCapital)
    Cost.append(temp)
  return Cost

#CostMatrix=CalTransactCost(PosDif,0.02,test_data_final,10000,transposed_rescaled_result)

import pandas as pd

tc = {
    "Ticker": [
        "SPX", "S&P GSCI Excess Return Rate", "Bloomberg commodity excess return",
        "DBC US Equity", "LBUSTRUU (Benchmark for fixed income)", "MSFT", "AAPL",
        "NVDA", "AMZN", "GOOG", "META", "LLY", "TSLA", "AVGO", "V", "TSM", "NVO", "JPM",
        "UNH", "WMT", "MA", "XOM", "JNJ", "PG", "ASML", "HD", "MRK", "COST",
        "WN1 (UST 30y)", "UB1 (EUR 30y)", "TY1 (10y US Treasury)", "FV1 (5y US Treasury)"
,
        "TU1 (2y US Treasury)", "EURUSD", "GBPUSD", "AUDUSD", "NZDUSD", "USDCHF",
        "USDMYR", "EURCZK", "USDHKD", "USDTWD", "USDRUB", "USDJPY", "USDNOK", "USDMXN",
        "XB1 Comdty", "GC1 Comdty", "C 1 Comdty", "W 1 Comdty", "CO1 Comdty",
        "KC1 Comdty", "SB1 Comdty", "NG1 Comdty"
    ],
    "TC (bps)": [
        1, 4, 4, 12, 10, 2, 2, 5, 3, 3, 3, 6, 3, 6, 5, 6, 6, 6,
        6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 3, 3, 1, 2, 2, 1, 2, 2, 3, 1,
        7, 4, 3, 7, 10, 1, 4, 3, 4, 3, 5, 5, 4, 8, 8, 5
    ],
```

```
    "TC (decimal)": [
        0.0001, 0.0004, 0.0004, 0.0012, 0.001, 0.0002, 0.0002, 0.0005, 0.0003, 0.0003,
        0.0003, 0.0006, 0.0003, 0.0006, 0.0005, 0.0006, 0.0006, 0.0006, 0.0006, 0.0006,
        0.0006, 0.0006, 0.0006, 0.0006, 0.0006, 0.0006, 0.0006, 0.0006, 0.0003, 0.0003,
        0.0001, 0.0002, 0.0002, 0.0001, 0.0002, 0.0002, 0.0003, 0.0001, 0.0007, 0.0004,
        0.0003, 0.0007, 0.001, 0.0001, 0.0004, 0.0003, 0.0004, 0.0003, 0.0005, 0.0005,
        0.0004, 0.0008, 0.0008, 0.0005
    ]
}
tc_df = pd.DataFrame(tc)
# List of values to check for in the 'Ticker' column for rows to drop
values_to_drop = ["UB1 (EUR 30y)", "TU1 (2y US Treasury)"]

# Drop rows where 'Ticker' column contains any of the values in the list `values_to_drop`
tc_df_drop = tc_df[~tc_df['Ticker'].isin(values_to_drop)].reset_index()
tc_df_drop = tc_df_drop.drop(columns=['index'])
#tc_bp = tc_df_drop['TC (decimal)']
tc_df_drop
tc_bp = tc_df_drop['TC (decimal)']
```

In [ ]:

```
KC_ls = [KC_eqt, KC_fi, KC_fx, KC_comd] #4*9*69
final_pnl = []
import numpy as np
for i in range(len(test_data)):
  pnl = []
  for j in range(len(test_data[0])):
    if (i==0) and (j==0):
      tc_idx = 0
    else:
      tc_idx = tc_idx+1
    pos_Sizes_Test = []
    daily_pnl_df_res = []
    dailyPNL = []
    for t in range(len(test_data[i][0])):
      pos_Sizes_Test.append(normalize_position_sizes(pd.DataFrame(res_ls[i][j][t]["_trad
es"]), test_data[i][j][t]))
      dpnl = calculate_daily_pnl(res_ls[i][j][t]["_trades"], test_data[i][j][t]["Close"]
)
      daily_pnl_df_res.append(dpnl)
      dailyPNL.append(dpnl['DailyPnL'])
    transpose_res = [list(row) for row in zip(*pos_Sizes_Test)]
    print(i)
    print(j)
    print(len(transpose_res[i]))
    print(len(KC_ls[i][j]))
    print('/n')
    result_trans_res = [np.multiply(KC_ls[i][j], pos) for pos in transpose_res]
    transposed_dailyPNL = [list(sublist) for sublist in zip(*dailyPNL)]
    converted_pos_Sizes_Test = [[1 if item != 0 else 0 for item in sublist] for sublist
in pos_Sizes_Test]
    transposed_pos_Sizes_Test = list(map(list, zip(*converted_pos_Sizes_Test)))
    result = []
    for row in transposed_pos_Sizes_Test:
        multiplied_row = [item * KC_ls[i][j][k] for k, item in enumerate(row)]
        result.append(multiplied_row)
    rescaled_result = [[item / sum(sublist) if sum(sublist) != 0 else 0 for item in subl
ist] for sublist in result]
    dot_product_results = [sum(a * b for a, b in zip(sublist_rescaled, sublist_transpose
d))
                            for sublist_rescaled, sublist_transposed in zip(rescaled_result,
transposed_dailyPNL)]

    PosDif = []
    for m in range(len(pos_Sizes_Test)):
        k = []
        for n in range(len(pos_Sizes_Test[0])):
            if n == 0:
                k.append(0)
            else:
```

```
            k.append(abs(pos_Sizes_Test[m][n] - pos_Sizes_Test[m][n - 1]))
        PosDif.append(k)
    # PosDif is a 2-D list of shape 69 * 232, PosDif[i][j] is the difference in position
size of asset i between day j and day j-1
    sumPosDiff = [sum(sublist) for sublist in zip(*PosDif)]
    # sumPosDiff is a 1-D list of shape 232, sumPosDiff[i] is the sum of the differences
in position size of all assets between day i and day i-1
    transposed_rescaled_result = [list(sublist) for sublist in zip(*rescaled_result)]
    print(len(transposed_rescaled_result))
    CostMatrix = CalTransactCost(PosDif, tc_bp[tc_idx], test_data[i][j], 10000, transpos
ed_rescaled_result)
    # CostMatrix is a 2-D list of shape 69 * 232, CostMatrix[i][j] is the transaction cos
t of asset i on day j

    dailyTransactionCost = [sum(sublist) for sublist in zip(*CostMatrix)]
    # dailyTransactionCost is a 1-D list of shape 232, dailyTransactionCost[i] is the sum
of transaction costs of all assets

    returnsWithTransactionCosts = []
    for h in range(len(dailyTransactionCost)):
        returnsWithTransactionCosts.append(dot_product_results[h] - dailyTransactionCost
[h])
    pnl.append(returnsWithTransactionCosts)
  final_pnl.append(pnl)
```

In [ ]:

```
!pip install dill
import dill as pickle
with open("final_pnl.pkl",'wb') as f:
  pickle.dump(final_pnl, f)
```

In [ ]:

```
!pip install xlsxwriter
excel_path = "/content/final_excel.xlsx"
writer = pd.ExcelWriter(excel_path, engine='xlsxwriter')

for i, data in enumerate(final_pnl):
    df = pd.DataFrame(data)
    df.to_excel(writer, sheet_name=f'Sheet{i+1}', index=False)

writer.save()
```

In [ ]:

```

```

Editing from Here

In [ ]:

```
import numpy as np
weights = np.array([0.4, 0.2, 0.2, 0.2])

# Assuming the PnL arrays and weights are defined as before
asset1_pnl_array = np.array(final_pnl[0])
asset2_pnl_array = np.array(final_pnl[1])
asset3_pnl_array = np.array(final_pnl[2])
asset4_pnl_array = np.array(final_pnl[3])
# Compute the weighted average of the PnL across assets for each period
weighted_avg_pnl = (asset1_pnl_array * weights[0] +
                    asset2_pnl_array * weights[1] +
                    asset3_pnl_array * weights[2] +
                    asset4_pnl_array * weights[3])

# Convert the result to a list if needed
weighted_avg_pnl_list = weighted_avg_pnl.tolist()

risk_free_rate = 0.025
```

```
period_sharpe_ratios = []
period_sortino_ratios = []

for i in range(len(weighted_avg_pnl_list)):
    returns = np.array(weighted_avg_pnl_list[i])
    mean_return = np.mean(returns)
    std_dev = np.std(returns)
    downside_returns = returns[returns < risk_free_rate]
    downside_deviation = np.std(downside_returns) if len(downside_returns) > 0 else 0

    # Sharpe Ratio calculation
    sharpe_ratio = (mean_return - risk_free_rate) / std_dev * np.sqrt(252)
    period_sharpe_ratios.append(sharpe_ratio)

    # Sortino Ratio calculation
    sortino_ratio = (mean_return - risk_free_rate) / downside_deviation * np.sqrt(252) i
f downside_deviation > 0 else 0
    period_sortino_ratios.append(sortino_ratio)

# If you wish to see the calculated ratios
print("Sharpe Ratios:", period_sharpe_ratios)
print("Sortino Ratios:", period_sortino_ratios)
```

In [ ]:

```
max(weighted_avg_pnl[1])
```

In [ ]:

```
final_pnl_by_asset=[]
for i in range(len(final_pnl)):
  temp=[]
  for j in range(len(final_pnl[i])):
    for k in range(len(final_pnl[i][j])):
      temp.append(final_pnl[i][j][k])
  final_pnl_by_asset.append(temp)
```

In [ ]:

```
initial_capital=100
```

In [ ]:

```
import matplotlib.pyplot as plt

capital_evolution = [initial_capital]
for percentage_return in final_pnl_by_asset[0]:
    capital_evolution.append(capital_evolution[-1] * (1 + percentage_return / 100))

# Plotting
plt.figure(figsize=(15, 3))
plt.plot(capital_evolution, linestyle='-', color='b')
plt.title('Capital Evolution using Returns from Equities')
plt.xlabel('Time Period (Days)')
plt.ylabel('Capital')
plt.grid(True)
plt.show()
```

In [ ]:

```
capital_evolution = [initial_capital]
for percentage_return in final_pnl_by_asset[1]:
    capital_evolution.append(capital_evolution[-1] * (1 + percentage_return / 100))

# Plotting
plt.figure(figsize=(15, 3))
plt.plot(capital_evolution, linestyle='-', color='red')
plt.title('Capital Evolution using Returns from Fixed Income Investments')
plt.xlabel('Time Period (Days)')
plt.ylabel('Capital')
plt.grid(True)
```

```
plt.show()
```

In [ ]:

```
capital_evolution = [initial_capital]
for percentage_return in final_pnl_by_asset[2]:
    capital_evolution.append(capital_evolution[-1] * (1 + percentage_return / 100))

# Plotting
plt.figure(figsize=(15, 3))
plt.plot(capital_evolution, linestyle='-', color='green')
plt.title('Capital Evolution using Returns from Foreign Exchange Investments')
plt.xlabel('Time Period (Days)')
plt.ylabel('Capital')
plt.grid(True)
plt.show()
```

In [ ]:

```
capital_evolution = [initial_capital]
for percentage_return in final_pnl_by_asset[3]:
    capital_evolution.append(capital_evolution[-1] * (1 + percentage_return / 100))

# Plotting
plt.figure(figsize=(15, 3))
plt.plot(capital_evolution, linestyle='-', color='orange')
plt.title('Capital Evolution using Returns from Commodities Investments')
plt.xlabel('Time Period (Days)')
plt.ylabel('Capital')
plt.grid(True)
plt.show()
```

In [ ]:

```
weighted_avg_pnl_final=[]
for i in range(len(final_pnl_by_asset[0])):
  weighted_avg_pnl_final.append(final_pnl_by_asset[0][i]*weights[0]+final_pnl_by_asset[1
][i]*weights[1]+final_pnl_by_asset[2][i]*weights[2]+final_pnl_by_asset[3][i]*weights[3])
```

In [ ]:

```
capital_evolution = [initial_capital]
for percentage_return in weighted_avg_pnl_final:
    capital_evolution.append(capital_evolution[-1] * (1 + percentage_return / 100))

# Plotting
plt.figure(figsize=(15, 6))
plt.plot(capital_evolution, linestyle='-', color='green')
plt.title('Capital Evolution of our Portfolio')
plt.xlabel('Time Period (Days)')
plt.ylabel('Capital')
plt.grid(True)
plt.show()
```

Complete Period Metrics

In [ ]:

```
trading_days = 252
risk_free_rate = 0.025
for i in range(len(final_pnl_by_asset)):
    pnl_array = np.array(final_pnl_by_asset[i])
    geometric_mean_daily = np.prod(1 + pnl_array/100)**(1/len(pnl_array)) - 1
    average_annual_return = (1 + geometric_mean_daily)**trading_days - 1
    annualized_volatility = np.std(pnl_array, ddof=1) * np.sqrt(trading_days)
    annualized_sharpe_ratio = (average_annual_return - risk_free_rate) / annualized_vola
tility
    print("Asset", i+1)
    print("average_annual_return", average_annual_return*100)
    print("annualized_volatility", annualized_volatility)
```

```
    print("annualized_sharpe_ratio", annualized_sharpe_ratio*100)
```

In [ ]:

```
trading_days = 252

# Calculate geometric mean of daily returns
geometric_mean_daily = np.prod(1 + final_pnl_by_asset)**(1/len(final_pnl_by_asset)) - 1

# Convert the average daily return to average annual return using geometric mean
average_annual_return = (1 + geometric_mean_daily)**trading_days - 1

# Calculate annualized volatility
annualized_volatility = np.std(final_pnl_by_asset, ddof=1) * np.sqrt(trading_days)

# Assuming risk-free rate is 0 for simplicity; replace with actual risk-free rate if avai
lable
risk_free_rate = 0

# Calculate annualized Sharpe ratio
annualized_sharpe_ratio = (average_annual_return - risk_free_rate) / annualized_volatilit
y

average_annual_return, annualized_volatility, annualized_sharpe_ratio
```

First Half metrics

In [ ]:

```
final_pnl_by_asset_fh=[]
for i in range(len(final_pnl)):
  temp=[]
  for j in range(len(final_pnl[i])//2):
    for k in range(len(final_pnl[i][j])):
      temp.append(final_pnl[i][j][k])
  final_pnl_by_asset_fh.append(temp)
```

In [ ]:

```
trading_days = 252
risk_free_rate = 0.025
for i in range(len(final_pnl_by_asset_fh)):
    pnl_array = np.array(final_pnl_by_asset_fh[i])
    geometric_mean_daily = np.prod(1 + pnl_array/100)**(1/len(pnl_array)) - 1
    average_annual_return = (1 + geometric_mean_daily)**trading_days - 1
    annualized_volatility = np.std(pnl_array, ddof=1) * np.sqrt(trading_days)
    annualized_sharpe_ratio = (average_annual_return - risk_free_rate) / annualized_vola
tility
    print("Asset", i+1)
    print("average_annual_return", average_annual_return*100)
    print("annualized_volatility", annualized_volatility)
    print("annualized_sharpe_ratio", annualized_sharpe_ratio*100)
```

Second Half Metrics

In [ ]:

```
final_pnl_by_asset_sh=[]
for i in range(len(final_pnl)):
  temp=[]
  for j in range(len(final_pnl[i])//2,len(final_pnl[i]),1):
    for k in range(len(final_pnl[i][j])):
      temp.append(final_pnl[i][j][k])
  final_pnl_by_asset_sh.append(temp)
```

In [ ]:

```
trading_days = 252
risk_free_rate = 0.025
```

```
for i in range(len(final_pnl_by_asset_sh)):
    pnl_array = np.array(final_pnl_by_asset_sh[i])
    geometric_mean_daily = np.prod(1 + pnl_array/100)**(1/len(pnl_array)) - 1
    average_annual_return = (1 + geometric_mean_daily)**trading_days - 1
    annualized_volatility = np.std(pnl_array, ddof=1) * np.sqrt(trading_days)
    annualized_sharpe_ratio = (average_annual_return - risk_free_rate) / annualized_vola
tility
    print("Asset", i+1)
    print("average_annual_return", average_annual_return*100)
    print("annualized_volatility", annualized_volatility)
    print("annualized_sharpe_ratio", annualized_sharpe_ratio*100)
```

Portfolio Metrics

In [ ]:

```
pnl_array = np.array(weighted_avg_pnl_final)
geometric_mean_daily = np.prod(1 + pnl_array/100)**(1/len(pnl_array)) - 1
average_annual_return = (1 + geometric_mean_daily)**trading_days - 1
annualized_volatility = np.std(pnl_array, ddof=1) * np.sqrt(trading_days)
annualized_sharpe_ratio = (average_annual_return - risk_free_rate) / annualized_volatilit
y
print("Complete Portfolio - Total Testing")
print("average_annual_return", average_annual_return*100)
print("annualized_volatility", annualized_volatility)
print("annualized_sharpe_ratio", annualized_sharpe_ratio*100)
```

In [ ]:

```
pnl_array = np.array(weighted_avg_pnl_final)
midpoint = len(pnl_array) // 2
pnl_array_fh=pnl_array[:midpoint]
geometric_mean_daily = np.prod(1 + pnl_array_fh/100)**(1/len(pnl_array_fh)) - 1
average_annual_return = (1 + geometric_mean_daily)**trading_days - 1
annualized_volatility = np.std(pnl_array_fh, ddof=1) * np.sqrt(trading_days)
annualized_sharpe_ratio = (average_annual_return - risk_free_rate) / annualized_volatilit
y
print("Complete Portfolio - First Half")
print("average_annual_return", average_annual_return*100)
print("annualized_volatility", annualized_volatility)
print("annualized_sharpe_ratio", annualized_sharpe_ratio*100)
```

In [ ]:

```
pnl_array = np.array(weighted_avg_pnl_final)
midpoint = len(pnl_array) // 2
pnl_array_sh=pnl_array[midpoint:]
geometric_mean_daily = np.prod(1 + pnl_array_sh/100)**(1/len(pnl_array_sh)) - 1
average_annual_return = (1 + geometric_mean_daily)**trading_days - 1
annualized_volatility = np.std(pnl_array_sh, ddof=1) * np.sqrt(trading_days)
annualized_sharpe_ratio = (average_annual_return - risk_free_rate) / annualized_volatilit
y
print("Complete Portfolio - Second Half")
print("average_annual_return", average_annual_return*100)
print("annualized_volatility", annualized_volatility)
print("annualized_sharpe_ratio", annualized_sharpe_ratio*100)
```