به نام خدا

درس الگوریتم‌های معاملاتی

پروژه پایان‌ترم

اعضای گروه:

نوید ابراهیمی (۹۹۵۲۱۰۰۱)

مهدی قضاوی (۹۹۵۲۲۰۱۴)

بهمن ۱۴۰۲

Fetch Crypto Data

```
[2]  # Step 1: Get Historical Data for Cryptocurrencies
     # Choose four cryptocurrencies of your choice
     tickers = ['BTC-USD', 'ETH-USD', 'XRP-USD', 'LTC-USD']
```

```
[3]  # Define the time frame
     start_date = '2022-11-01'
     end_date = '2023-11-01'
```

```
[4]  # Download historical data
     data = yf.download(tickers, start=start_date, end=end_date)['Adj Close']
     data, type(data)
```

بخش اول پروژه) در ابتدا close داده‌های ۴ crypto را در بازه زمانی یکساله دریافت می‌کنیم.

Define Initial Cap and Random Weights

```
# Step 2: Define Initial Capital and Coefficients
initial_capital = 1000
coefficients = np.array([0.4, 0.18, 0.27, 0.15])  # Sum of coefficients must be equal to 1
```

```
[57] assert round(sum(coefficients)) == 1, "The sum of coefficients must be equal to one."
     assert all(value > 0 for value in coefficients), "All values are greater than 0."
```

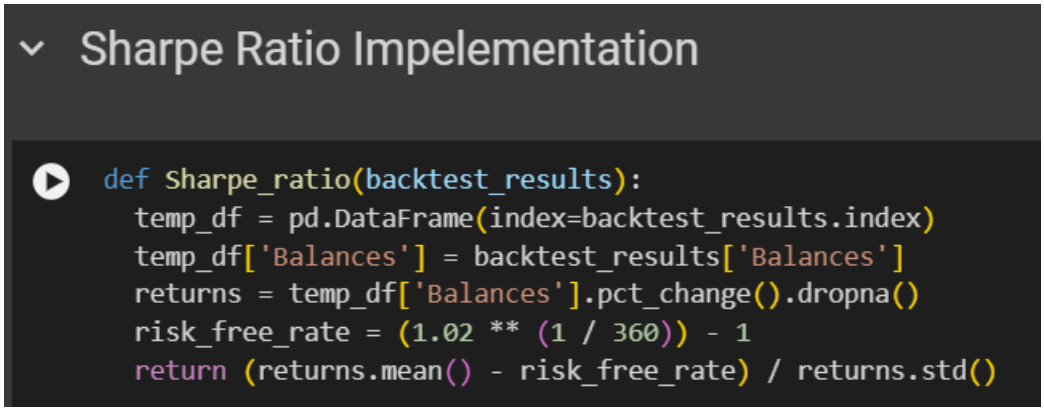
در ادامه مقدار capital را طبق خواست سوال ۱۰۰۰ و به هر crypto یک ضریب می‌دهیم. با این شرایط که جمع همه آنها برابر ۱ و هریک از آنها مقدار بزرگ‌تر از صفر داشته باشند. این مورد را با assert نیز چک می‌کنیم.

The Buy and Hold Strategy Implementation

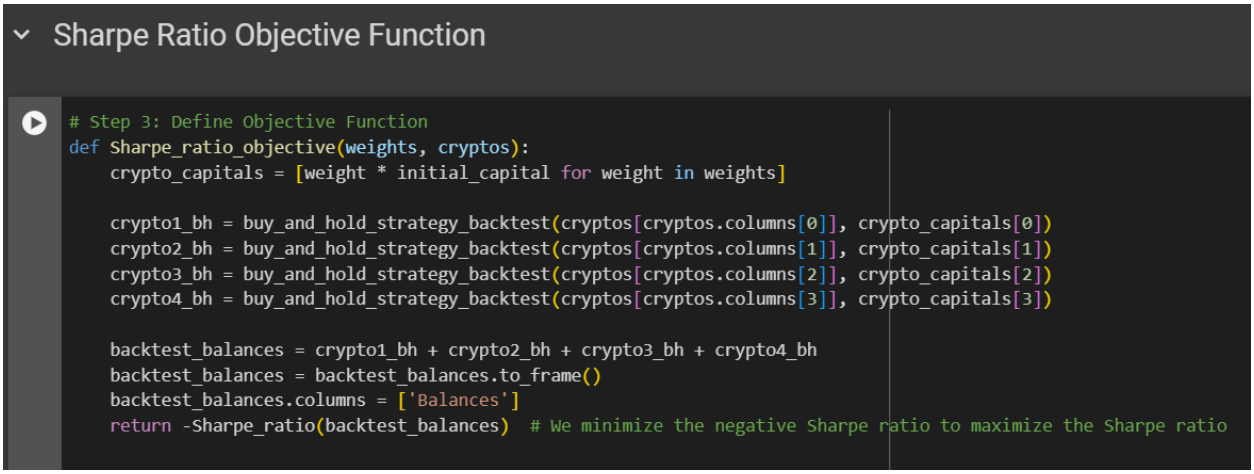
```
[58] def buy_and_hold_strategy_backtest(asset_returns, initial_cap=1000):
         account_balance = initial_cap
         cumulative_return = (1 + asset_returns).cumprod()
         account_balance *= cumulative_return

         return account_balance
```

این تابع سناریویی را مدل‌سازی می‌کند که در آن مقدار مشخصی از سرمایه را در یک دارایی سرمایه‌گذاری می‌کنیم، آن دارایی را برای کل دوره زمانی نگه می‌داریم، و در پایان ارزش سرمایه‌گذاری خود را محاسبه می‌کنیم.

تابع buy_and_hold_strategy_backtest ، استراتژی خرید و نگهداری را پیاده‌سازی می‌کند. این تابع با استفاده از بازده دارایی‌ها، موجودی حساب را محاسبه می‌کند. این تابع ابتدا موجودی حساب را با مقدار سرمایه اولیه مشخص شده در ورودی تابع مقداردهی می‌کند. سپس با استفاده از بازده دارایی‌ها، بازده کلی(cumulative return) را محاسبه می‌کند. در نهایت، موجودی حساب را با ضرب موجودی حساب در بازده کلی به‌روزرسانی می‌کند.



در اینجا نیز تابع Sharpe ratio را طبق فرمول $\frac{R-r_f}{std}$ پیاده‌سازی کردیم.



در این بخش، یک *Objective Function* برای پیاده‌سازی *Buy&Hold* و محاسبه *Sharpe ratio* پیاده کرده‌ایم. دلیل این امر تابع *minimize* است زیرا این تابع حتما یک *object* از *Sharpe* بگیرد. در داخل این تابع، در ابتدا برروی هر رمزارز، باتوجه به سرمایه تعیین شده توسط آرایه وزن‌ها، استراتژی *B&H* را برروی آن رمزارز پیاده کرده و سپس سرمایه نهایی که حاصل جمع خروجی هر استراتژی برروی هر رمزارز است را حساب

می‌کنیم. سپس *Sharpe Ratio* را روی این *Balance*ها محاسبه می‌کنیم. حال هدف این‌است که متریک *Sharpe Ratio* را بیشینه کنیم، به‌همین منظور، از تابع *minimize* استفاده می‌کنیم، برای این‌کار، در این تابع *Objective*، منفی مقدار *Sharpe Ratio* را خروجی می‌دهیم تا با *minimize* کردن این مقدار، مقدار خود متریک *Sharpe Ratio* بیشینه شود(درواقع قرینه آن را مینیمم می‌کنیم تا خود آن بیشینه شود!).

```
[63] init_sr = -1 * Sharpe_ratio_objective(coefficients, data)
     init_sr

     0.04502073952504293
```

خروجی *Sharpe ratio* برروی *B&H* با وزن‌های ابتدایی.

```
Find Optimum Weights for Maximizing the Sharpe Ratio

[121] # Step 5: Define Constraints
      n = len(coefficients)
      constraints = [{'type': 'eq', 'fun': lambda x: np.sum(x) - 1}]
      for i in range(n):
          constraints.append({'type': 'ineq', 'fun': lambda x,i=i: x[i] - 1e-10})

[122] bounds = [(0, None)] * n

[123] # Step 6: Optimize Portfolio for Sharpe Ratio
      result = minimize(Sharpe_ratio_objective, coefficients, args=(data,), method='SLSQP', constraints=constraints, bounds=bounds)
      optimal_weights_sharpe = result.x

[124] np.set_printoptions(precision=15)
      optimal_weights_sharpe

      array([9.999999996999998e-01, 9.999964875259670e-11,
             1.000003370041358e-10, 1.000002580742176e-10])

[125] assert round(sum(optimal_weights_sharpe)) == 1, "The sum of coefficients must be equal to one."
      assert all(value > 0 for value in optimal_weights_sharpe), "All values are greater than 0."

Calculate Sharpe Ratio for Buy and Hold with optimum weights

[126] optimum_sr = -1 * Sharpe_ratio_objective(optimal_weights_sharpe, data)
      optimum_sr

      0.07061678668857137
```

در ادامه، بهترین وزن‌ها را با توجه به *Sharpe Ratio* و تابع هدفی که تعریف کردیم و به‌کمک متد *minimize* از کتابخانه *scipy*، محاسبه کنیم. در ابتدا یک سری *constraints* و *bounds* تعریف می‌کنیم.

بخش محدودیت‌ها برای یک مسئله بهینه‌سازی با استفاده از تابع *scipy.optimize.minimize* تعیین شده‌اند، به ویژه برای روش *Sequential Least Squares Quadratic Programming (SLSQP)*.

متغیر *n* تعداد وزن‌ها است. خط زیر لیست *constraints* را با یک محدودیت تساوی مقداردهی اولیه می‌کند. این محدودیت تضمین می‌کند که مجموع تمام وزن‌ها (*x*) برابر با ۱ باشد، که نشان دهنده این است که وزن‌ها باید یک تخصیص پرتفوی معتبر را نمایش دهند که تمام سرمایه در آن سرمایه‌گذاری شده است .

```
constraints = [{'type': 'eq', 'fun': lambda x: np.sum(x) - 1}]
```

حلقه زیر، محدودیت‌های نامساوی را برای هر وزن اضافه می‌کند. برای هر وزن، یک محدودیت اضافه می‌کند که وزن مربوطه *(x[i])* باید بزرگتر یا مساوی یک مقدار بسیار کوچک مثل *(1e-10)* باشد. این یک عمل برای این‌است تا اطمینان حاصل شود که هر وزن بزرگتر یا مساوی صفر باشد.

هدف از این محدودیت‌ها جلوگیری از اختصاص وزن منفی به هر دارایی است و اطمینان حاصل می‌کند که وزن‌ها همواره مثبت یا صفر باقی می‌مانند.

```
for i in range(n):
    constraints.append({'type': 'ineq', 'fun': lambda x,i=i: x[i] - 1e-10})
```

در ادامه تابع *minimize* را اجرا کرده و وزن‌های بهینه را دریافت می‌کنیم. سپس با *assert* چک می‌کنیم این هم تمام وزن‌ها بزرگ‌تر از صفر و هم مجموعشان یک شود.

در استفاده از این متد، از الگوریتم *SLSQP* برای کمینه کردن یک تابع اسکالر یک یا چند متغیره با *Constraint*های مشخص‌شده استفاده کرده‌ایم. محدودیت‌های تعریف شده، تضمین می‌کنند که هر وزن بزرگتر از صفر است و مجموع آن‌ها برابر ۱ است.

نحوه توزیع وزن‌های بهینه پیدا شده (وزن‌های نزدیک به یک برای رمزارز اول و وزن‌های نزدیک به صفر برای سه ارز دیگر) نشان‌دهنده این است که الگوریتم بهینه‌سازی بهینه‌ترین حالت برای *Objective Function* را وقتی پیداکرده که تقریباً تمام سرمایه به ارز دیجیتال اول تخصیص یابد.

دلیل این نتیجه می‌تواند به خصوصیات مسئله بهینه‌سازی و محدودیت‌هایی که تعریف کرده‌ایم، باز گردد. اینجا ما بهینه‌سازی را برای بیشینه‌سازی *Sharpe Ratio* تنظیم کرده‌ایم و با توجه به این که محدودیت‌ها تضمین می‌کنند که وزن‌ها بزرگتر از صفر باشند و مجموع آنها برابر با یک باشد، بهینه‌ساز به‌این نتیجه رسیده که بیشترین سرمایه را به رمزارز اول تخصیص دهد.

دلایل این نتیجه ممکن است عبارت باشد از:

- **پویایی بازار(*Market Dynamics*)**: داده‌های عملکرد تاریخی برای رمزارز اول، ممکن است نشان دهنده بازدهی با ریسک بهتر(*Higher risk-adjusted return*) نسبت به سه ارز دیگر باشد. بنابراین، بهینه‌ساز برای بیشینه‌سازی نسبت*Sharpe* ، بیشترین سرمایه را به این ارز اختصاص داده است.
- **همبستگی و تنوع(*Correlation and Diversification*)**: اگر رمزارز اول همبستگی کم یا منفی با سه ارز دیگر داشته باشد، بهینه‌ساز ممکن است به تمرکز سرمایه بر روی این ارز برای *Risk Diversification* بهتر رسیده باشد.
- **داده‌ها و عملکرد تاریخی:** داده‌های تاریخی رمزارز اول ممکن‌است به‌گونه‌ای بوده باشند که *Performance* بهتری نسبت به سایر رمزارزها داشته و درنتیجه، بهینه‌ساز بیشترین سرمایه را به آن تخصیص داده است.
- **غیرخطی بودن تابع هدف:** نسبت *Sharpe* و بیشینه‌سازی آن گاهی اوقات بهینه‌سازی‌های غیرخطی را نتیجه می‌دهد. احتمالاً *Solution Space* نزدیک محل بهینه برای رمزارز اول نسبت به سایر ارزها شیب شدیدتری دارد که باعث می‌شود تغییرات کوچک در وزن آن ارز اثرات بسیار زیادی داشته باشد.

برای درک بهتر دلایل پشت این وزن‌ها، ممکن است بخواهیم *Performance* تاریخی هر ارز را بررسی کرده، نسبت *Sharpe* هر یک را محاسبه و همچنین همبستگی بین آنها را بررسی کنیم. همچنین، باید درنظر داشت که محدودیت‌های بهینه‌سازی را تنظیم یا روش‌های بهینه‌سازی مختلف را برای بررسی تخصیص‌های پرتفوی مختلف استفاده کنیم.

در ادامه همانطور که مشاهده می‌شود با وزن‌های بهتر، مقدار *Sharpe ratio* افزایش می‌یابد و این یعنی این مقادیر به درستی انتخاب و بهینه شده‌اند.

Sortino Ratio Implementation

```python
def downside_deviation(backtest_results):
    temp_df = pd.DataFrame(index=backtest_results.index)
    temp_df['Balances'] = backtest_results['Balances']
    returns = temp_df['Balances'].pct_change().dropna()
    downside_returns = returns[returns < 0]
    return downside_returns.std()

def Sortino_ratio(backtest_results):
    temp_df = pd.DataFrame(index=backtest_results.index)
    temp_df['Balances'] = backtest_results['Balances']
    returns = temp_df['Balances'].pct_change().dropna()
    risk_free_rate = (1.02 ** (1 / 360)) - 1
    return (returns.mean() - risk_free_rate) / downside_deviation(backtest_results)
```

در اینجا *Sortino ratio* را پیاده کرده‌ایم. به این صورت که مانند *Sharpe ratio* است با این تفاوت که صرفا *Return*هایی که *std* منفی دارند درنظر گرفته شده‌اند.

Sortino Ratio Objective Function

```python
def Sortino_ratio_objective(weights, cryptos):
    crypto_capitals = [weight * initial_capital for weight in weights]

    crypto1_bh = buy_and_hold_strategy_backtest(cryptos[cryptos.columns[0]], crypto_capitals[0])
    crypto2_bh = buy_and_hold_strategy_backtest(cryptos[cryptos.columns[1]], crypto_capitals[1])
    crypto3_bh = buy_and_hold_strategy_backtest(cryptos[cryptos.columns[2]], crypto_capitals[2])
    crypto4_bh = buy_and_hold_strategy_backtest(cryptos[cryptos.columns[3]], crypto_capitals[3])

    backtest_balances = crypto1_bh + crypto2_bh + crypto3_bh + crypto4_bh
    backtest_balances = backtest_balances.to_frame()
    backtest_balances.columns = ['Balances']
    return -Sortino_ratio(backtest_balances)  # We minimize the negative Sortino ratio to maximize the Sortino ratio
```

Sortino Ratio for Buy and Hold with initial weights

```python
[129] init_sortino = -1 * Sortino_ratio_objective(coefficients, data)
      init_sortino
```

```
0.06155679890556221
```

در اینجا نیز به مانند *Sharpe ratio* داده‌ها را ترکیب و *Sortino ratio* را روی آن محاسبه کردیم. به همان دلیل *Sharpe ratio*، تابع *Sortino ratio* نیز مقدار منفی برمیگرداند.

Find Optimum Weights for Maximizing the Sortino Ratio

```
[230] # Step 5: Define Constraints
      n = len(coefficients)
      constraints = [{'type': 'eq', 'fun': lambda x: np.sum(x) - 1}]
      for i in range(n):
          constraints.append({'type': 'ineq', 'fun': lambda x,i=i: x[i] - 1e-10})
```

```
[231] result = minimize(Sortino_ratio_objective, coefficients, args=(data,), method='SLSQP', constraints=constraints, bounds=bounds)
      optimal_weights_sortino = result.x
      optimal_weights_sortino
```

```
array([9.999999996999994e-01, 1.000002199103012e-10,
       1.000007264495562e-10, 9.999971684049314e-11])
```

```
[232] assert round(sum(optimal_weights_sortino)) == 1, "The sum of coefficients must be equal to one."
      assert all(value > 0 for value in optimal_weights_sortino), "All values are greater than 0."
```

Calculate Sortino Ratio for Buy and Hold with optimum weights

```
[233] optimum_sortino = -1 * Sortino_ratio_objective(optimal_weights_sortino, data)
      optimum_sortino
```

```
0.10083459939218704
```

این بخش نیز به مانند *Sharpe ratio* پیاده‌سازی شده‌است. همانطور که مشخص است با وزن‌ها بهینه مقدار *Sortino ratio* افزایش پیدا کرده است.

Net Profit Implementation

```
[234] def compute_net_profit(initial_investment, final_portfolio_value):
          net_profit = final_portfolio_value - initial_investment
          return net_profit
```

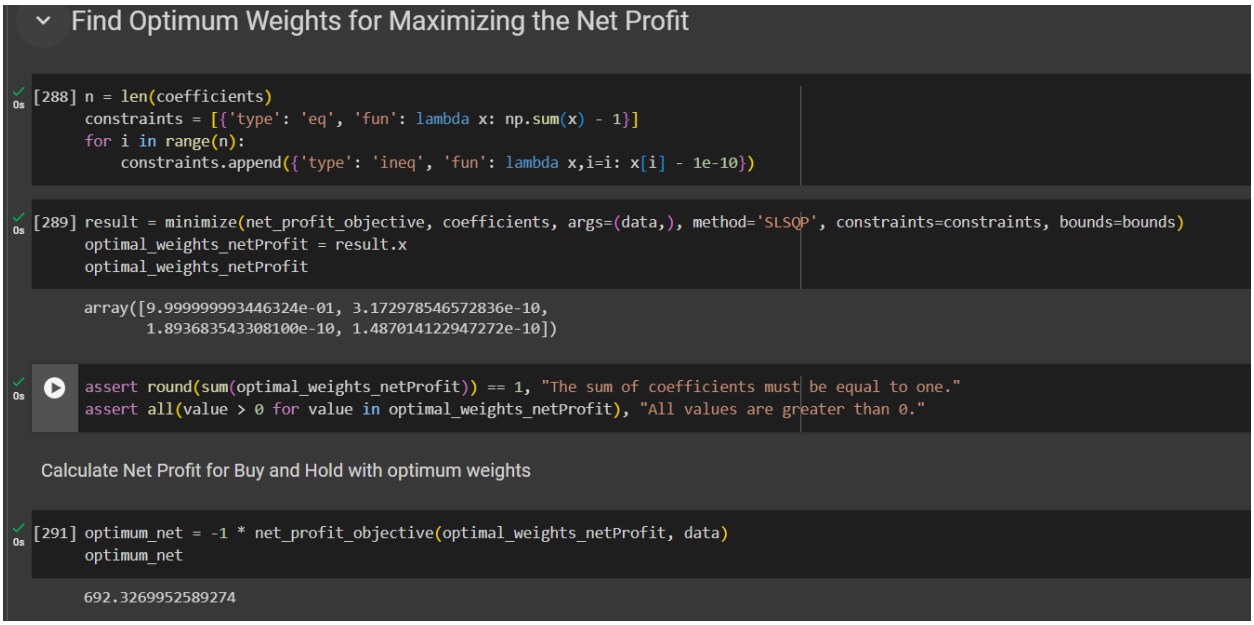
```
[235] def net_profit_objective(weights, cryptos):
          crypto_capitals = [weight * initial_capital for weight in weights]

          crypto1_bh = buy_and_hold_strategy_backtest(cryptos[cryptos.columns[0]], crypto_capitals[0])
          crypto2_bh = buy_and_hold_strategy_backtest(cryptos[cryptos.columns[1]], crypto_capitals[1])
          crypto3_bh = buy_and_hold_strategy_backtest(cryptos[cryptos.columns[2]], crypto_capitals[2])
          crypto4_bh = buy_and_hold_strategy_backtest(cryptos[cryptos.columns[3]], crypto_capitals[3])

          backtest_balances = crypto1_bh + crypto2_bh + crypto3_bh + crypto4_bh
          final_portfolio_value = crypto1_bh[-1] + crypto2_bh[-1] + crypto3_bh[-1] + crypto4_bh[-1]

          backtest_balances = backtest_balances.to_frame()
          backtest_balances.columns = ['Balances']
          return -compute_net_profit(initial_capital, final_portfolio_value)
```

Net Profit for Buy and Hold with initial weights

```
[236] init_net = -1 * net_profit_objective(coefficients, data)
      init_net
```

```
415.422261975481
```

در اینجا نیز پیاده‌سازی دقیقا به مانند ***Sharpe ratio*** است با این تفاوت که که فرمول ***Net profit*** را در آن اثر دادیم که آخرین مقدار را از اولین مقدار کم می‌کنیم که حاصل ***Net profit*** است.



Find Optimum Weights for Maximizing the Net Profit

```
[288] n = len(coefficients)
      constraints = [{'type': 'eq', 'fun': lambda x: np.sum(x) - 1}]
      for i in range(n):
          constraints.append({'type': 'ineq', 'fun': lambda x,i=i: x[i] - 1e-10})
```

```
[289] result = minimize(net_profit_objective, coefficients, args=(data,), method='SLSQP', constraints=constraints, bounds=bounds)
      optimal_weights_netProfit = result.x
      optimal_weights_netProfit
```

```
array([9.999999993446324e-01, 3.172978546572836e-10,
       1.893683543308100e-10, 1.487014122947272e-10])
```

```
assert round(sum(optimal_weights_netProfit)) == 1, "The sum of coefficients must be equal to one."
assert all(value > 0 for value in optimal_weights_netProfit), "All values are greater than 0."
```

Calculate Net Profit for Buy and Hold with optimum weights

```
[291] optimum_net = -1 * net_profit_objective(optimal_weights_netProfit, data)
      optimum_net
```

```
692.3269952589274
```

در اینجا نیز دقیقا مثل قبل پیاده‌سازی کردیم. همانطور که مشخص است مقدار ***Net Profit*** نسبت به قبل افزایش پیدا کرده است.

بخش دوم پروژه)

Sharpe Ratio Calculation on Buy and Hold with data from 2023-11-02 till 2023-12-02

Calculate Sharpe Ratio for Buy and Hold with initial Weights

```
[ ] init_sr = -1 * Sharpe_ratio_objective(coefficients, test_data)
    init_sr
```

```
0.1285905855559773
```

Calculate Sharpe Ratio for Buy and Hold with optimum Weights from 2022-11-01 till 2023-11-01

```
[ ] optimum_sr = -1 * Sharpe_ratio_objective(optimal_weights_sharpe, test_data)
    optimum_sr
```

```
0.17728326095656688
```

در ابتدا ***Sharpe ratio*** را با وزن‌های اولیه در داده‌ها زمانی یک ماهه اجرا می‌کنیم. سپس با استفاده از وزن‌های بهینه شده آن را محاسبه می‌کنیم.

همانطور که مشخص است با وزن‌های بهینه ***Sharpe ratio*** بالاتری میگیریم.

Sortino Ratio Calculation on Buy and Hold with data from 2023-11-02 till 2023-12-02

Calculate Sortino Ratio for Buy and Hold with initial Weights

```
[ ] init_sortino = -1 * Sortino_ratio_objective(coefficients, test_data)
    init_sortino
```

```
0.1781280722317655
```

Calculate Sortino Ratio for Buy and Hold with optimum weights from 2022-11-01 till 2023-11-01

```
[ ] optimum_sortino = -1 * Sortino_ratio_objective(optimal_weights_sortino, test_data)
    optimum_sortino
```

```
0.2517768331859311
```

Net Profit Calculation on Buy and Hold with data from 2023-11-02 till 2023-12-02

Calculate Net Profit for Buy and Hold with initial Weights

```
[ ] init_net = -1 * net_profit_objective(coefficients, test_data)
    init_net
```

```
81.55078648430435
```

Calculate Net Profit for Buy and Hold with optimum weights from 2022-11-01 till 2023-11-01

```
[ ] optimum_net = -1 * net_profit_objective(optimal_weights_netProfit, test_data)
    optimum_net
```

```
107.34678041153438
```

برای *Sortino ratio* و *net profit* نیز به همین صورت عمل کرده‌ایم.

حال برای هر سه معیار، داده‌ها را روی داده‌های یک ماهه بررسی می‌کنیم و وزن‌های بهینه این ۳ماه را به دست می‌آوریم. همانطور که مشخص است مقادیر این ۳ معیار بعد از بهینه‌سازی وزن‌ها مقدار بهتری پیدا کرده‌اند:

Find Optimum Weights for Maximizing the Sharpe Ratio on Test Data

```
[352] # Step 6: Optimize Portfolio for Sharpe Ratio
      result = minimize(Sharpe_ratio_objective, coefficients, args=(test_data,), method='SLSQP', constraints=constraints, bounds=bounds)
      optimal_weights_sharpe = result.x
```

```
[353] optimal_weights_sharpe
```

```
array([8.154934652358800e-01, 1.845065345641201e-01,
       1.000001203805417e-10, 1.000000097919201e-10])
```

```
[354] assert round(sum(optimal_weights_sharpe)) == 1, "The sum of coefficients must be equal to one."
      assert all(value > 0 for value in optimal_weights_sharpe), "All values are greater than 0."
```

Calculate Sharpe Ratio for Buy and Hold with optimum weights from 2023-11-02 till 2023-12-02

```
[355] optimum_sr = -1 * Sharpe_ratio_objective(optimal_weights_sharpe, test_data)
      optimum_sr
```

```
0.17989591604262797
```

Find Optimum Weights for Maximizing the Sortino Ratio on Test Data

```
[356] result = minimize(Sortino_ratio_objective, coefficients, args=(test_data,), method='SLSQP', constraints=constraints, bounds=bounds)
      optimal_weights_sortino = result.x
      optimal_weights_sortino
```

```
array([9.999892580658809e-11, 9.999999997000011e-01,
       9.999998745735539e-11, 1.000000724588057e-10])
```

```
[357] assert round(sum(optimal_weights_sortino)) == 1, "The sum of coefficients must be equal to one."
      assert all(value > 0 for value in optimal_weights_sortino), "All values are greater than 0."
```

Calculate Sortino Ratio for Buy and Hold with optimum weights from 2023-11-02 till 2023-12-02

```
[358] optimum_sortino = -1 * Sortino_ratio_objective(optimal_weights_sortino, test_data)
      optimum_sortino
```

```
0.31220518881769305
```

Find Optimum Weights for Maximizing the Net Profit on Test Data

```
[359] result = minimize(net_profit_objective, coefficients, args=(test_data,), method='SLSQP', constraints=constraints, bounds=bounds)
      optimal_weights_netProfit = result.x
      optimal_weights_netProfit
```

```
array([1.067818611311111e-10, 9.999999997171569e-01,
       9.011741353148750e-11, 8.594394640404346e-11])
```

```
      assert round(sum(optimal_weights_netProfit)) == 1, "The sum of coefficients must be equal to one."
      assert all(value > 0 for value in optimal_weights_netProfit), "All values are greater than 0."
```

Calculate Net Profit for Buy and Hold with optimum weights from 2023-11-02 till 2023-12-02

```
[361] optimum_net = -1 * net_profit_objective(optimal_weights_netProfit, test_data)
      optimum_net
```

```
159.12228356278865
```