<u>FE5214 Assignment 2</u>

Gaurav Agrawal - A0294856J

Part A

In this part of the assignment, you will construct an alpha factor based on the prior 5-day returns and evaluate the effectiveness of the factor.

1. Consider the US market and the years 2005 to 2024. The universe used will be based on the universe from the start of the year.

2. To construct the factor,

a) calculate the volatility using the prior 21 days of daily returns (use the log return, the return is set to 0 if there is an "NA" in the adjusted prices). If obtained is less than 0.005, set it to 0.005

b) calculate the prior 5-day return; you can use the log return

c) normalize the variable by dividing the volatility obtained in step a),

d) subtract out the industry component, over all stocks in the industry that the stock i belongs to.

<u>Answer:</u>

```
df_grand.head()
```

Date	Ticker	Price	Daily_simple_return	Daily_log_return	Volatility	5_day_return	Normalized_5_day_return	Industry_code	Industry_avg_5_day_return
2005-01-03	0111145D	26.790000	-0.017241	-0.017392	0.012329	-0.014453	-1.172219	nan	-0.040952
2005-01-04	0111145D	26.570000	-0.008212	-0.008246	0.012238	-0.032219	-2.632739	nan	-1.773800
2005-01-05	0111145D	26.400000	-0.006398	-0.006419	0.011001	-0.029485	-2.680136	nan	-2.584019
2005-01-06	0111145D	26.600000	0.007576	0.007547	0.010035	-0.028171	-2.807378	nan	-2.514027
2005-01-07	0111145D	26.390000	-0.007895	-0.007926	0.009858	-0.032435	-3.290385	nan	-2.633882

Date	alpha_factor	Industry_avg_daily_return	Industry_removed_daily_return	Rank	N	Ranked_factor	Industry_avg_ranked_factor	alpha_factor_rank
2005-01-03	-1.131266	-0.012683	-0.004709	388.000000	500	-0.551102	0.011568	-0.562670
2005-01-04	-0.858939	-0.016698	0.008452	341.000000	500	-0.362725	0.064336	-0.427061
2005-01-05	-0.096117	-0.009282	0.002863	229.000000	500	0.086172	0.019584	0.066588
2005-01-06	-0.293351	0.003407	0.004141	275.000000	500	-0.098196	0.005432	-0.103628
2005-01-07	-0.656503	-0.001810	-0.006116	301.000000	500	-0.202405	-0.000622	-0.201783

Steps:

1. **Load and Preprocess Data**

   - Read the universe dataset (univ_h.csv), which defines the stock universe per year, and set the index as the year.
   - Read the tickers dataset (tickers.csv) containing ticker symbols and their corresponding GICS industry codes.
   - Read the adjusted price dataset (adjusted.csv), convert dates to a standard format, and set them as the index.

2. **Iterate Over Years (2005–2024)**

   - For each year in the range: Select stocks that were part of universe in that year.
   - Extract last 25 trading days of the previous year and all data for the current year.
   - Convert the dataset into long format with columns: Ticker, Date, and Price.

3. **Compute Financial Metrics**

   - Calculate Daily Simple Returns and Daily Log Returns for each stock.
   - Compute 21-day rolling volatility, ensuring a minimum threshold of 0.005
   - Compute 5-day cumulative log returns and normalize them using volatility.
   - Merge with the tickers dataset to extract the industry code.

4. **Industry-Level Aggregation**

   - Compute the industry average 5-day return for each date and industry.
   - Used the GICS code first 6 digits for industry mapping. **Classified all the tickers that does not have GICS code as 'nan' group and considered its average.**
   - Define the alpha factor as the deviation of a stock's normalized return from its industry average.
   - Calculate industry average daily return and subtract it from individual stock returns to remove industry effects.

5. **Ranking-Based Factor Computation**

   - Rank stocks daily based on their normalized 5-day return (highest return ranked first).
   - Compute the rank-based factor using ranking formula v ← (N+1-2k)/(N-1)
   - Compute industry average ranked factor and define the ranked alpha factor as the difference between a stock's rank and the industry's average rank.

6. **Final Data Compilation**

   - Filter the processed data to include only the current year's results.
   - Append yearly results into a consolidated dataset (df_grand) containing all computed factors

Note: Detailed Python Code for above steps can be found in Appendix at end of this report

3. Do a cross-sectional regression

$$R_i(t+1) = \beta(t)v_i(t) + \epsilon_i(t), \qquad i = 1, \ldots, N$$

on everyday t (except for the last day of the available data) and get a time series of $\beta(t)$ and a time series of $R^2(t)$. Here the industry return is removed from $r_i(t+1)$: $R_i(t+1) = r_i(t+1) - r_I(t+1)$

Answer:

df_beta_r2

Date	Beta	R^2
2005-01-03	-0.000937	0.007912
2005-01-04	-0.000515	0.002367
2005-01-05	-0.000620	0.005109
2005-01-06	0.000143	0.000528
2005-01-07	-0.000175	0.001344
...	...	...
2024-12-23	-0.000377	0.009806
2024-12-24	0.000099	0.000434
2024-12-26	-0.000202	0.002632
2024-12-27	0.000612	0.008389
2024-12-30	-0.000028	0.000018

Steps:

- Shift the Industry-removed daily return by one day forward to align returns at time (t+1) with factor values at time (t). Drop the last day's data.
- For each trading day, extract stock-level alpha factors (v_i_t) and future industry-removed returns (R_i_t+1).
- Compute Beta (β_t) using the cross-sectional regression formula:
  - $\beta(t) = \frac{\sum_i R_i(t+1)v_i(t)}{\sum_i v_i(t)v_i(t)}$. Ensured to handle cases where the denominator is zero.
- Compute R^2 using: $R^2(t) = 1 - \frac{\sum_i \epsilon_i^2(t)}{\sum_i R_i^2(t+1)}$.
- Store computed values of β_t and R^2_t for each date.
- Create separate DataFrames for Beta and R^2, then merge them into a final dataset (df_beta_r2).

Note: Detailed Python Code for above steps can be found in Appendix at end of this report

4. For the years 2005 to 2024, calculate and list the average of $\beta(t)$ for each year and the corresponding t-stat, $\sqrt{T}$ ×(the average of $\beta(t)$)/(the standard deviation of $\beta(t)$), where T is the number of β values obtained for the year. Comment on your results.

Answer:

```
print(df_beta_stats)
```

```
    Year  Mean_Beta  Std_Beta  Count_Beta    t_stat
0   2005  -0.000071  0.000677         252 -1.668749
1   2006  -0.000103  0.000667         251 -2.450820
2   2007  -0.000076  0.000997         251 -1.215268
3   2008  -0.000422  0.004450         253 -1.510077
4   2009  -0.000124  0.002077         252 -0.944924
5   2010  -0.000053  0.000767         252 -1.092108
6   2011  -0.000187  0.001319         252 -2.253198
7   2012   0.000045  0.000662         250  1.067904
8   2013  -0.000057  0.000556         252 -1.633802
9   2014  -0.000026  0.000634         252 -0.649245
10  2015  -0.000019  0.000874         252 -0.344989
11  2016  -0.000070  0.001058         252 -1.048558
12  2017  -0.000041  0.000599         251 -1.080492
13  2018   0.000023  0.000932         251  0.394479
14  2019   0.000097  0.001161         252  1.327495
15  2020  -0.000369  0.003775         253 -1.555849
16  2021   0.000014  0.001478         252  0.149436
17  2022  -0.000019  0.001787         251 -0.167628
18  2023  -0.000179  0.001727         250 -1.640380
19  2024  -0.000102  0.000878         251 -1.848868
```

Comments:

The table shows yearly summary statistics for Beta estimates from 2005 to 2024. Here's a brief commentary on the results:

1. **Mean_Beta**: The average Beta across each year is consistently close to zero, indicating that on average, the observed beta values have a minimal net directional effect.

2. **Std_Beta**: The standard deviation of Beta varies over the years, showing periods of higher variability (e.g., 2008, 2011, 2020) possibly due to market turbulence like the financial crisis (2008) or COVID-19 pandemic (2020).

3. **Count_Beta**: The number of observations per year is consistently around 251–253, matching typical trading days in a year.

4. **t_stat**: The t-statistic tests whether the mean beta is significantly different from zero. Most years have t-statistics between -2 and +2, suggesting that the mean beta is not significantly different from zero at the 5% significance level. However, years like 2011 show more extreme t-statistics, indicating potential significance.

Steps:

- Extract the year from each date to facilitate yearly aggregation.
- Compute the mean (Mean_Beta) and standard deviation (Std_Beta) of Beta for each year.
- Count the number of observations (Count_Beta), which should be approximately 250 trading days per year.
- Calculate the t-statistic to assess the statistical significance of Beta: $\sqrt{T}$ ×(the average of $\beta(t)$)/(the standard deviation of $\beta(t)$)
- This helps determine whether Beta is significantly different from zero over time.
- Store and display the results in a structured yearly dataset (df_beta_stats).

Note: Detailed Python Code for above steps can be found in Appendix at end of this report

5. Repeat the calculation (Steps 3-4) using a ranked variable, by adding a sub-step in the construction of the factor:
rank the normalized variable after step 2c) from the largest (rank 1) to the smallest (rank N) and redefine the factor in terms of the rank, k, v ← (N+1-2k)/(N-1),

Answer:

df_beta_r2_ranked

Date	Beta_Ranked	R^2_Ranked
2005-01-03	-0.002531	0.009236
2005-01-04	-0.001855	0.004733
2005-01-05	-0.001968	0.005826
2005-01-06	0.000166	0.000063
2005-01-07	-0.000379	0.000462
...	...	...
2024-12-23	-0.001572	0.017163
2024-12-24	0.000405	0.000960
2024-12-26	-0.000586	0.003162
2024-12-27	0.001381	0.010246
2024-12-30	0.000207	0.000268

```
print(df_beta_stats_ranked)
```

	Year	Mean_Beta	Std_Beta	Count_Beta	t_stat
0	2005	-0.000257	0.002014	252	-2.027004
1	2006	-0.000402	0.002035	251	-3.131211
2	2007	-0.000224	0.003288	251	-1.078360
3	2008	-0.000970	0.010362	253	-1.489546
4	2009	-0.000079	0.005424	252	-0.232261
5	2010	-0.000159	0.002055	252	-1.226659
6	2011	-0.000339	0.002971	252	-1.812117
7	2012	0.000062	0.002059	250	0.476460
8	2013	-0.000209	0.001673	252	-1.981417
9	2014	-0.000070	0.002050	252	-0.545735
10	2015	-0.000100	0.002474	252	-0.643860
11	2016	-0.000205	0.003286	252	-0.988287
12	2017	-0.000226	0.002046	251	-1.751343
13	2018	0.000081	0.002707	251	0.476457
14	2019	0.000190	0.003204	252	0.941279
15	2020	-0.000836	0.010800	253	-1.230735
16	2021	-0.000034	0.004815	252	-0.112421
17	2022	-0.000142	0.005495	251	-0.410177
18	2023	-0.000598	0.005178	250	-1.826291
19	2024	-0.000326	0.003035	251	-1.702588

Comments:

The table shows yearly summary statistics for Beta for ranked_alpha_factor estimates from 2005 to 2024. Here's a brief commentary on the results:

1. **Mean_Beta**: The average Beta remains close to zero across years, with some fluctuations.
2. **Std_Beta**: The standard deviation varies significantly across years. 2008 (0.010362) and 2020 (0.018020) exhibit the highest standard deviations, reflecting extreme market volatility during the global financial crisis and the COVID-19 pandemic.
3. **Count_Beta**: The number of observations per year remains consistent, with values around 250–253, matching the expected number of trading days in a year.
4. **t_stat**: The t-statistic suggests whether the mean beta is significantly different from zero. The most extreme values occur in 2006 (-3.13) and 2005 (-2.02), indicating statistical significance at 5% levels.

Steps:

- For each trading day, extract ranked alpha factor (v_i_t_ranked) and future industry-removed returns (R_i_t+1).
- Compute Beta (β_t_ranked) and R^2 Value as done in Q3.
- Store computed values of β_t_ranked and R^2_t_ranked for each date.
- Create separate DataFrames for Beta_Ranked and R^2_Ranked, then merge them into a final dataset (df_beta_r2_ranked).
- Extract the year to facilitate yearly aggregation.
- Compute the mean (Mean_Beta) and standard deviation (Std_Beta) of Beta_Ranked for each year.
- Count the number of observations (Count_Beta), which should be approximately 250 trading days per year.
- Calculate the t-statistic to assess the statistical significance of Beta_Ranked as done in Q4.
- This helps determine whether Beta_Ranked is significantly different from zero over time.
- Store and display the results in a structured yearly dataset (df_beta_stats_ranked).

Note: Detailed Python Code for above steps can be found in Appendix at end of this report

Part B

From the years 2006 to 2024, use the previous year's average beta $\overline{\beta_v}$, calculated in Part A (for example, for the year 2007, use the average β_v obtained for the year 2006, and evaluate the expected returns for all trading days of the year 2007, $R_{Ei}(t, t+1) = \overline{\beta_v} v_i(t)$

Construct and evaluate the portfolio as follows,

1. On each day t, rank the stocks according to the expected returns, and long (with equal weights) the top 20% of the stocks with the largest values of $R_{Ei}(t, t+1)$ and short the bottom 20% of the stocks with the smallest values (most negative values) of $R_{Ei}(t, t+1)$
2. Get the portfolio return at each time step t. The return is on the long market value of the portfolio, so it is the sum of the returns on individual positions divided by the number of long positions in the portfolio. Note that when calculating the portfolio return, the full return $r_i(t+1) \equiv r_i(t, t+1)$ without subtracting the market return is used.

Answer:

daily_portfolio_df

Date	Portfolio_Return
2006-01-03	0.005560
2006-01-04	0.001497
2006-01-05	0.003302
2006-01-06	-0.001455
2006-01-09	0.001030
...	...
2024-12-23	0.001869
2024-12-24	-0.000501
2024-12-26	0.000987
2024-12-27	-0.001329
2024-12-30	0.000000

4780 rows × 1 columns

Steps:

1. **Initialize Storage Structures**:
   - portfolio_results and portfolio_results_cost: These lists store annual return, annualized volatility, and Sharpe ratio for each year, with and without trading costs.
   - daily_portfolio_df and daily_portfolio_df_cost: These DataFrames store daily portfolio returns for each day, with and without trading costs.
2. **Iterate Over Each Year (2006–2024)**:

- Extract the previous year's mean beta (β_v): For each year, the average beta of the previous year is retrieved from the df_beta_stats DataFrame.
- Filter the data for the current year: The data for each year is filtered from df_grand to only include entries from the current year.
- Compute expected return: The expected return for each stock is calculated using the formula: $R_{Ei}(t, t+1) = \overline{\beta_v} v_i(t)$

3. **Compute Daily Returns for Each Stock**:
   - Shift the daily log return by one day: The daily returns are shifted by one day to align with the expected returns of the next day.
   - Apply trading costs: A trading cost of 5 basis points per trade is considered for each position change. This cost is factored into the portfolio's return later in the process.
4. **Construct the Long-Short Portfolio for Each Day**:
   - Determine N (total stocks), N_L (top 20%), and N_S (bottom 20%): For each day, the total number of stocks (N) is calculated, and the top 20% (N_L) are assigned to the long positions, while the bottom 20% (N_S) are assigned to short positions.
   - Rank stocks based on expected return: Stocks are ranked by their expected return, and the top and bottom 20% are selected for long and short positions, respectively.
   - Create Signals: Signals are created based on the ranked stocks:
     - Long positions (Signal = 1): Top 20% of stocks based on expected returns.
     - Short positions (Signal = -1): Bottom 20% of stocks based on expected returns.
     - Neutral or No Position (Signal = 0): Stocks outside of the top or bottom 20%.
5. **Compute Daily Portfolio Return**:
   - Long Return: The return for the long positions is calculated by summing the next-day log returns of the stocks selected for long positions.
   - Short Return: The return for the short positions is calculated similarly but using the stocks selected for short positions.
   - Portfolio Return: The portfolio return is calculated as the difference between long
6. **Compute Annual Metrics:**
   - Annual Return: The annual return is the sum of the daily portfolio returns over the year.
   - Annualized Volatility: The standard deviation of daily returns is annualized by multiplying by 252\sqrt{252}252 (the typical number of trading days in a year).
7. **Store and Output Results**:
   - Convert results to DataFrame: The results for each year (annual return, annualized volatility, Sharpe ratio) are stored in two DataFrames: df_portfolio_results (without costs) and df_portfolio_results_cost (with costs).
   - Store daily returns for further analysis: The daily returns (with and without costs) are stored in daily_portfolio_df and daily_portfolio_df_cost for potential future analysis or plotting.

Note: Detailed Python Code for above steps can be found in Appendix at end of this report

3. For each year calculate the annual return (assuming the cost of trading is 0, and for simplicity, simply add up all daily portfolio returns to get the annual return) and the annualized return volatility of the portfolio. List your results in a table. Which are the best and the worst years for the strategy?

Answer:

```
df_portfolio_results  #annual results
```

	Year	Annual Return	Annualized Volatility	Sharpe Ratio
0	2006	0.130048	0.046117	2.818964
1	2007	0.079170	0.073012	1.083855
2	2008	0.331588	0.223993	1.480306
3	2009	0.045828	0.118626	0.386399
4	2010	0.039365	0.046998	0.837095
5	2011	0.158924	0.064150	2.478354
6	2012	-0.006986	0.045407	-0.153858
7	2013	-0.037771	0.037580	-1.005095
8	2014	0.019737	0.042442	0.464636
9	2015	0.032675	0.053299	0.613040
10	2016	0.049079	0.067459	0.727761
11	2017	0.068524	0.041835	1.637845
12	2018	-0.048233	0.051646	-0.933963
13	2019	0.069221	0.068652	1.007947
14	2020	-0.307040	0.225987	-1.358551
15	2021	0.020481	0.097534	0.209965
16	2022	-0.078204	0.103273	-0.756897
17	2023	0.109809	0.100619	1.091346
18	2024	0.126679	0.065991	1.919739

This is table of **annual portfolio returns**, annualized volatility, and Sharpe ratios from 2006 to 2024. It provides insights into the strategy's performance over time:

- The strategy exhibits significant fluctuations in annual returns, with extreme values such as +33.16% (2008) and -30.7% (2020). This suggests that the strategy is highly sensitive to market conditions.
- 2008 and 2020 show the highest annualized volatility (~22%), indicating market turmoil (Global Financial Crisis and COVID-19 pandemic).
- Several years (e.g., 2013, 2018, 2020, 2022) had negative returns, highlighting potential drawdowns.

```
Best and Worst years based on annual returns

Best Year for the Strategy: 2008  Annual Return: 0.3315884513673465
Annualized Volatility: 0.22399301741619793    Sharpe Ratio: 1.4803055249310313

Worst Year for the Strategy: 2020   Annual Return: -0.30703969786625424
Annualized Volatility: 0.2259870448848793    Sharpe Ratio: -1.3585506167766566
```

Best and Worst Years Based on Annual Returns

- **2008 was the best year**, likely benefiting a short strategy during the Global Financial Crisis (GFC). However, liquidity issues could have made it difficult to realize these gains.
- **2020 was the worst year**, reflecting extreme market volatility due to COVID-19 disruptions.

```
Best and Worst years based on sharpe ratio

Best Year for the Strategy: 2006  Annual Return: 0.13004756670931872
Annualized Volatility: 0.04611707624065673    Sharpe Ratio: 2.8189641910042003

Worst Year for the Strategy: 2020   Annual Return: -0.30703969786625424
Annualized Volatility: 0.2259870448848793    Sharpe Ratio: -1.3585506167766566
```

Best and Worst Years Based on Sharpe Ratio

- **2006 had the best Sharpe ratio**, indicating strong risk-adjusted returns with relatively low volatility.
- **2020 had the worst Sharpe ratio**, reinforcing the negative impact of high volatility on returns

Steps:

- print(daily_portfolio_results) will print the annual portfolio returns for each date in the dataset stored as explained in Q2.
- df_portfolio_results.loc[df_portfolio_results["Annual Return"].idxmax()] gets the row with the maximum annual return to identify the best year.
- df_portfolio_results.loc[df_portfolio_results["Annual Return"].idxmin()] gets the row with the minimum annual return to identify the worst year.
- Outputs the year, annual return, and annualized volatility for both the best and worst years.
- Similarly best results based on sharpe ratio are also generated

Note: Detailed Python Code for above steps can be found in Appendix at end of this report

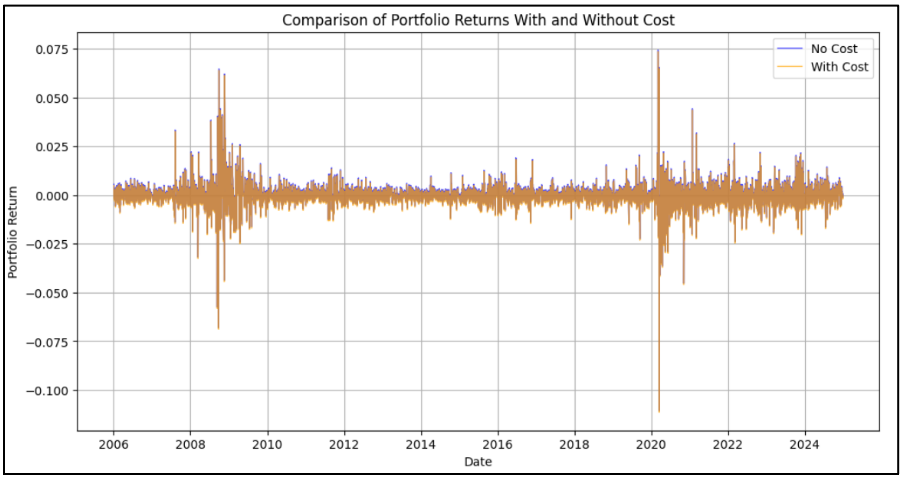
4. Assume that the percentage trading cost is 5 bps. Calculate the portfolio returns, taking into account the cost. Compare the results to the case when the costs are not taken into account. For simplicity, we assume the LMV (the long market value) of the portfolio is kept the same and we ignore the cost of maintaining the constant LMV.

Answer:

```
print(merged_df)
```

```
Comparison of Portfolio returns with and without cost
            Portfolio_Return_no_cost  Portfolio_Return_with_cost
Date
2006-01-03                  0.005560                    0.004895
2006-01-04                  0.001497                    0.000785
2006-01-05                  0.003302                    0.002587
2006-01-06                 -0.001455                   -0.002160
2006-01-09                  0.001030                    0.000325
...                              ...                         ...
2024-12-23                  0.001869                    0.001176
2024-12-24                 -0.000501                   -0.001196
2024-12-26                  0.000987                    0.000317
2024-12-27                 -0.001329                   -0.001999
2024-12-30                  0.000000                   -0.000748

[4780 rows x 2 columns]
```



The table and plot compares portfolio returns before and after trading costs. Trading costs is calculated based on the position changed from day to day basis.

- The table shows that transaction costs reduce the portfolio returns. In each instance, the "Portfolio_Return_with_cost" is slightly lower than the "Portfolio_Return_no_cost."
- Over time, these small differences can accumulate, significantly impacting long-term performance.
- The graph highlights that the general pattern of returns and volatility remains the same for both cases, but the "With Cost" returns tend to have slightly lower values.
- Large spikes in return deviations, particularly around the 2008 financial crisis and the 2020 COVID-19 crash, show more pronounced negative values when costs are factored in. This suggests that costs exacerbate drawdowns during high-volatility periods.

```
df_portfolio_results_cost  # annual returns with cost
```

	Year	Annual Return	Annualized Volatility	Sharpe Ratio
0	2006	-0.044449	0.046133	-0.963489
1	2007	-0.096506	0.073045	-1.321191
2	2008	0.155178	0.224000	0.692757
3	2009	-0.130372	0.118602	-1.099240
4	2010	-0.137135	0.047026	-2.916168
5	2011	-0.017617	0.064125	-0.274735
6	2012	-0.181403	0.045407	-3.995019
7	2013	-0.213847	0.037579	-5.690583
8	2014	-0.156005	0.042478	-3.672588
9	2015	-0.143752	0.053300	-2.697051
10	2016	-0.126910	0.067438	-1.881888
11	2017	-0.106146	0.041838	-2.537063
12	2018	-0.222570	0.051644	-4.309706
13	2019	-0.105531	0.068675	-1.536673
14	2020	-0.483124	0.226005	-2.137666
15	2021	-0.155175	0.097546	-1.590786
16	2022	-0.252754	0.103321	-2.446289
17	2023	-0.065089	0.100618	-0.646898
18	2024	-0.048229	0.065988	-0.730875

The table shows **annual portfolio return with cost.** It has experienced negative annual returns in the majority of years, with only a few exceptions (e.g., 2008). Without costs, some of these years might have shown positive performance as shown in earlier table of results without cost. This could be due to frequent trading and high transaction costs.

```
Best and Worst years based on annual returns taking cost into account

Best Year for the Strategy: 2008  Annual Return: 0.1551776449157336
Annualized Volatility: 0.22400000931077757    Sharpe Ratio: 0.6927573145786801

Worst Year for the Strategy: 2020   Annual Return: -0.4831238818001116
Annualized Volatility: 0.2260053428077248    Sharpe Ratio: -2.1376657551460263
```

Best and Worst Years Based on Annual Returns taking cost into account

- **2008 was the best year**, similar results as portfolio return without cost. Likely benefiting a short strategy during the Global Financial Crisis (GFC). However, liquidity issues could have made it difficult to realize these gains.
- **2020 was the worst year**, reflecting extreme market volatility due to COVID-19 disruptions.

```
Best and Worst years based on sharpe ratio taking cost into account

Best Year for the Strategy: 2008  Annual Return: 0.1551776449157336
Annualized Volatility: 0.22400000931077757    Sharpe Ratio: 0.6927573145786801

Worst Year for the Strategy: 2013   Annual Return: -0.21384729452605994
Annualized Volatility: 0.03757915366110958    Sharpe Ratio: -5.690583041186718
```

Best and Worst Years Based on Sharpe Ratio taking cost into account

Best Year: 2008, with the highest Sharpe Ratio of 0.69, confirming strong risk-adjusted performance.

Worst Year: 2013, Sharpe Ratio of -5.69, While 2020 had a worse return, 2013 had the worst risk-adjusted performance, indicating very poor returns relative to risk.

Steps: To Apply Trading Costs:

- Shift the data to determine position changes: The stock signals (long or short) are shifted by one day, and position changes are calculated by comparing the current day's signals with the previous day's signals.
- Determine active stocks: Active stocks are those for which there is a non-zero signal (either long or short).
- Calculate Trading Penalty: The trading cost penalty is based on the number of position changes across the active stocks. Specifically:
    - Position Changes: The number of changes in stock signals (e.g., switching from long to short or vice versa).
    - Trading Cost Penalty: The penalty is calculated as the product of the trading cost (5 bps) and the ratio of position changes to active stocks, which accounts for the number of trades executed each day, **assuming equally weighted portfolio**.
- Adjust Portfolio Return with Costs: The portfolio return is adjusted by subtracting the trading cost penalty from the original portfolio return to account for the cost of executing trades.
- Rest results are generated in same way as for portfolio return without cost earlier

Note: Detailed Python Code for above steps can be found in Appendix at end of this report

Appendix

In [1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import statsmodels.api as sm
import warnings
warnings.filterwarnings("ignore")

pd.set_option('display.max_rows', 20) # Show all rows
pd.set_option('display.max_columns', 20) # Show all columns
pd.options.display.float_format = '{:,.6f}'.format
```

In [2]:
```
# Read the required CSV files and perform necessary preprocessing
df_universe = pd.read_csv('../Data/data_us/univ_h.csv')
df_universe.set_index('year', inplace=True)

columns = ['Ticker', 'GICS_Code']
df_tickers = pd.read_csv('../Data/data_us/tickers.csv', header=None, names = columns, dtype=str)
df_tickers.set_index('Ticker', inplace=True)

df_adjusted = pd.read_csv('../Data/data_us/adjusted.csv')
df_adjusted['Date'] = pd.to_datetime(df_adjusted['Date'], format = '%Y%m%d')
df_adjusted.set_index('Date', inplace=True)
df_adjusted.columns = df_adjusted.columns.str.strip()
```

In [3]:
```
df_universe.head()
```

Out[3]:

year	0111145D	0202445Q	0203524D	0226226D	0544749D	0574018D	0772031D	0848680D	0867887D	0910150D	...
2004	1	1	1	1	1	0	1	1	0	1	...
2005	1	1	1	1	1	1	1	1	0	1	...
2006	1	1	1	1	1	1	1	1	0	1	...
2007	1	1	1	1	0	1	1	1	0	1	...
2008	1	1	1	1	0	0	1	1	0	0	...

5 rows × 947 columns

In [4]:
```
df_tickers.tail()
```

Out[4]:

Ticker	GICS_Code
YUM	25301040
ZBH	35101010
ZBRA	45203010
ZION	40101015
ZTS	35202010

In [5]:
```
df_adjusted.head()
```

Out[5]:

Date	0111145D	0202445Q	0203524D	0226226D	0544749D	0574018D	0772031D	0848680D	0867887D	0910150D	...
2004-01-02	23.600000	42.990000	17.634400	46.155500	21.816700	23.197600	3.332900	14.117200	NaN	16.711400	...
2004-01-05	23.720000	43.060000	18.282400	45.355300	22.232600	23.214300	3.578300	14.696400	NaN	16.868400	...
2004-01-06	23.760000	42.500000	18.601900	46.553400	22.754800	23.197600	3.562500	15.284200	NaN	16.888000	...
2004-01-07	23.520000	43.390000	18.323500	46.332300	22.722400	22.997400	3.515000	15.085400	NaN	16.583800	...
2004-01-08	23.360000	43.590000	19.167800	43.936100	22.680800	22.980700	3.578300	15.301500	NaN	16.799700	...

5 rows × 947 columns

Part - A

Question 1, 2

In [6]:
```
all_years_data = []

# Loop from 2005 to 2024
for year in range(2005, 2025):
    # Select tickers for the current year
    selected_tickers = df_universe.loc[year][df_universe.loc[year] == 1].index.tolist()

    # Extract data for the previous year (e.g., 2004 for 2005) and the current year (e.g., 2005)
    start_year = year - 1
    end_year = year
    df1 = df_adjusted.loc[f"{start_year}-01-01":f"{start_year}-12-31", selected_tickers].tail(25)
    df2 = df_adjusted.loc[f"{end_year}-01-01":f"{end_year}-12-31", selected_tickers]
    df_filtered = pd.concat([df1, df2])

    # Convert to long format
    df_long = df_filtered.melt(ignore_index=False, var_name="Ticker", value_name="Price")

    # Sort values by Ticker and Date
    df_long = df_long.sort_values(by=["Ticker", "Date"])

    # Calculate Simple Return
    df_long["Daily_simple_return"] = df_long.groupby("Ticker",group_keys=False)["Price"].apply(lambda x: x.pct_ch

    # Calculate Log Return
    df_long["Daily_log_return"] = df_long.groupby("Ticker",group_keys=False)["Price"].apply(lambda x: np.log(x /
    df_long.fillna(0, inplace = True)

    # Calculate Volatility (21-day rolling window)
    df_long["Volatility"] = df_long.groupby("Ticker",group_keys=False)["Daily_log_return"].apply(lambda x: x.roll
    df_long["Volatility"] = df_long["Volatility"].apply(lambda x: max(x, 0.005))

    # Calculate 5-day return (rolling sum)
    df_long["5_day_return"] = df_long.groupby("Ticker",group_keys=False)["Daily_log_return"].apply(lambda x: x.ro
    df_long["5_day_return"] = df_long["5_day_return"].fillna(0)

    #Calculate Normalized 5-day return by dividing it with volatility
    df_long["Normalized_5_day_return"] = df_long["5_day_return"] / df_long["Volatility"]

    # Merge with df_tickers to get GICS_Code and extract Industry Code
    df_long = df_long.merge(df_tickers[['GICS_Code']], left_on="Ticker", right_index=True, how="left")
    df_long['Industry_code'] = df_long['GICS_Code'].astype(str).str[:6]
    df_long = df_long.drop(columns=['GICS_Code'])

    # Calculate Industry average 5-day return (for each date and industry code)
    df_long['Industry_avg_5_day_return'] = df_long.groupby(["Industry_code", df_long.index.date])["Normalized_5_d

    # Calculate alpha factor
```

```python
    df_long["alpha_factor"] = df_long["Normalized_5_day_return"] - df_long["Industry_avg_5_day_return"]

    # Calculate Industry average daily return (for each date and industry code)
    df_long['Industry_avg_daily_return'] = df_long.groupby(["Industry_code", df_long.index.date])["Daily_log_retu

    # Remove Industry average daily return from daily log return
    df_long["Industry_removed_daily_return"] = df_long["Daily_log_return"] - df_long["Industry_avg_daily_return"]

    # Rank Normalized 5-day return (largest = rank 1, smallest = rank N)
    df_long["Rank"] = df_long.groupby(df_long.index)["Normalized_5_day_return"].rank(ascending=False, method="fir

    # Calculate N (number of stocks on each date)
    df_long["N"] = df_long.groupby(df_long.index)["Normalized_5_day_return"].transform("count")

    # Apply ranking formula for ranked factor
    df_long["Ranked_factor"] = (df_long["N"] + 1 - 2 * df_long["Rank"]) / (df_long["N"] - 1)

    # Calculate Industry average ranked facto (for each date and industry code)
    df_long['Industry_avg_ranked_factor'] = df_long.groupby(["Industry_code", df_long.index.date])["Ranked_factor

    # Calculate alpha factor ranked
    df_long["alpha_factor_rank"] = df_long["Ranked_factor"] - df_long["Industry_avg_ranked_factor"]

    # Filter data for current year onwards
    df_long = df_long.loc[df_long.index >= f'{end_year}-01-01']

    # Append the results of the current year to the list
    all_years_data.append(df_long)

# Concatenate all yearly datasets into one grand dataset
df_grand = pd.concat(all_years_data)
```

In [7]:
```python
df_grand.head()
```

Out[7]:

	Ticker	Price	Daily_simple_return	Daily_log_return	Volatility	5_day_return	Normalized_5_day_return	Industr
Date								
2005-01-03	0111145D	26.790000	-0.017241	-0.017392	0.012329	-0.014453	-1.172219	
2005-01-04	0111145D	26.570000	-0.008212	-0.008246	0.012238	-0.032219	-2.632739	
2005-01-05	0111145D	26.400000	-0.006398	-0.006419	0.011001	-0.029485	-2.680136	
2005-01-06	0111145D	26.600000	0.007576	0.007547	0.010035	-0.028171	-2.807378	
2005-01-07	0111145D	26.390000	-0.007895	-0.007926	0.009858	-0.032435	-3.290385	

In [8]:
```python
df_grand.shape
```

Out[8]: (2527068, 17)

Question 3

In [9]:
```python
# Shift Industry_removed_daily_return up by 1 day to align with time t
df_grand["Industry_removed_daily_return_shifted"] = df_grand.groupby("Ticker")["Industry_removed_daily_return"].s

# Drop last day's data since we can't compute R(t+1)
df_grand = df_grand.dropna(subset=["Industry_removed_daily_return_shifted"])

# Initialize lists to store results
beta_series = []
r_squared_series = []

# Group by date and perform cross-sectional regression
for date, group in df_grand.groupby(df_grand.index):
    v_i_t = group["alpha_factor"]
```

```python
    R_i_t1 = group["Industry_removed_daily_return_shifted"]

    # Compute beta(t)
    numerator = np.sum(R_i_t1 * v_i_t)
    denominator = np.sum(v_i_t * v_i_t)
    beta_t = numerator / denominator if denominator != 0 else np.nan  # Avoid division by zero

    # Compute residuals
    epsilon_t = R_i_t1 - beta_t * v_i_t
    sum_epsilon_sq = np.sum(epsilon_t ** 2)
    sum_R_sq = np.sum(R_i_t1 ** 2)

    # Compute R^2(t)
    r_squared_t = 1 - (sum_epsilon_sq / sum_R_sq) if sum_R_sq != 0 else np.nan

    # Store results
    beta_series.append((date, beta_t))
    r_squared_series.append((date, r_squared_t))

# Create DataFrame for beta and R² separately
df_beta_r2 = pd.DataFrame(beta_series, columns=["Date", "Beta"]).set_index("Date")
df_r2 = pd.DataFrame(r_squared_series, columns=["Date", "R^2"]).set_index("Date")

# Merge the two DataFrames into one beta dataset
df_beta_r2 = df_beta_r2.merge(df_r2, left_index=True, right_index=True)
```

In [10]:
```python
df_beta_r2
```

Out[10]:

Date	Beta	R^2
2005-01-03	-0.000937	0.007912
2005-01-04	-0.000515	0.002367
2005-01-05	-0.000620	0.005109
2005-01-06	0.000143	0.000528
2005-01-07	-0.000175	0.001344
...	...	...
2024-12-23	-0.000377	0.009806
2024-12-24	0.000099	0.000434
2024-12-26	-0.000202	0.002632
2024-12-27	0.000612	0.008389
2024-12-30	-0.000028	0.000018

5032 rows × 2 columns

Question 4

In [11]:
```python
# Ensure 'Date' column is datetime type if not already
df_beta_r2["Year"] = df_beta_r2.index.year  # Extract year from DateTime index

# Aggregate at the yearly level
df_beta_stats = df_beta_r2.groupby("Year")["Beta"].agg(
    Mean_Beta="mean",
    Std_Beta="std",
    Count_Beta="count"  # Should be ~250 per year
).reset_index()

# Compute t-statistic
df_beta_stats["t_stat"] = np.sqrt(df_beta_stats["Count_Beta"]) * (df_beta_stats["Mean_Beta"] / df_beta_stats["Std

# Display results
print(df_beta_stats)
```

```
    Year  Mean_Beta  Std_Beta  Count_Beta    t_stat
0   2005  -0.000071  0.000677         252 -1.668749
1   2006  -0.000103  0.000667         251 -2.450820
2   2007  -0.000076  0.000997         251 -1.215268
3   2008  -0.000422  0.004450         253 -1.510077
4   2009  -0.000124  0.002077         252 -0.944924
5   2010  -0.000053  0.000767         252 -1.092108
6   2011  -0.000187  0.001319         252 -2.253198
7   2012   0.000045  0.000662         250  1.067904
8   2013  -0.000057  0.000556         252 -1.633802
9   2014  -0.000026  0.000634         252 -0.649245
10  2015  -0.000019  0.000874         252 -0.344989
11  2016  -0.000070  0.001058         252 -1.048558
12  2017  -0.000041  0.000599         251 -1.080492
13  2018   0.000023  0.000932         251  0.394479
14  2019   0.000097  0.001161         252  1.327495
15  2020  -0.000369  0.003775         253 -1.555849
16  2021   0.000014  0.001478         252  0.149436
17  2022  -0.000019  0.001787         251 -0.167628
18  2023  -0.000179  0.001727         250 -1.640380
19  2024  -0.000102  0.000878         251 -1.848868
```

Question 5

In [12]:
```python
# Initialize lists to store results for ranked alpha factor
beta_series_ranked = []
r_squared_series_ranked = []

# Group by date and perform cross-sectional regression with ranked alpha factor
for date, group in df_grand.groupby(df_grand.index):
    v_i_t_ranked = group["alpha_factor_rank"]
    R_i_t1 = group["Industry_removed_daily_return_shifted"]

    # Compute beta(t) for ranked alpha factor
    numerator = np.sum(R_i_t1 * v_i_t_ranked)
    denominator = np.sum(v_i_t_ranked * v_i_t_ranked)
    beta_t_ranked = numerator / denominator if denominator != 0 else np.nan  # Avoid division by zero

    # Compute residuals
    epsilon_t_ranked = R_i_t1 - beta_t_ranked * v_i_t_ranked
    sum_epsilon_sq = np.sum(epsilon_t_ranked ** 2)
    sum_R_sq = np.sum(R_i_t1 ** 2)

    # Compute R^2(t) for ranked alpha factor
    r_squared_t_ranked = 1 - (sum_epsilon_sq / sum_R_sq) if sum_R_sq != 0 else np.nan

    # Store results
    beta_series_ranked.append((date, beta_t_ranked))
    r_squared_series_ranked.append((date, r_squared_t_ranked))

# Create DataFrame for beta and R² separately (using ranked alpha factor)
df_beta_r2_ranked = pd.DataFrame(beta_series_ranked, columns=["Date", "Beta_Ranked"]).set_index("Date")
df_r2_ranked = pd.DataFrame(r_squared_series_ranked, columns=["Date", "R^2_Ranked"]).set_index("Date")

# Merge the two DataFrames into one beta dataset
df_beta_r2_ranked = df_beta_r2_ranked.merge(df_r2_ranked, left_index=True, right_index=True)
```

In [13]:
```python
df_beta_r2_ranked
```

Out[13]:

Date	Beta_Ranked	R^2_Ranked
2005-01-03	-0.002531	0.009236
2005-01-04	-0.001855	0.004733
2005-01-05	-0.001968	0.005826
2005-01-06	0.000166	0.000063
2005-01-07	-0.000379	0.000462
...	...	...
2024-12-23	-0.001572	0.017163
2024-12-24	0.000405	0.000960
2024-12-26	-0.000586	0.003162
2024-12-27	0.001381	0.010246
2024-12-30	0.000207	0.000268

5032 rows × 2 columns

In [14]:

```
# Ensure 'Date' column is datetime type if not already
df_beta_r2_ranked["Year"] = df_beta_r2_ranked.index.year  # Extract year from DateTime index

# Aggregate at the yearly level
df_beta_stats_ranked = df_beta_r2_ranked.groupby("Year")["Beta_Ranked"].agg(
    Mean_Beta="mean",
    Std_Beta="std",
    Count_Beta="count"
).reset_index()

# Compute t-statistic
df_beta_stats_ranked["t_stat"] = np.sqrt(df_beta_stats_ranked["Count_Beta"]) * (df_beta_stats_ranked["Mean_Beta"]

# Display results
print(df_beta_stats_ranked)
```

```
    Year  Mean_Beta  Std_Beta  Count_Beta    t_stat
0   2005  -0.000257  0.002014         252 -2.027004
1   2006  -0.000402  0.002035         251 -3.131211
2   2007  -0.000224  0.003288         251 -1.078360
3   2008  -0.000970  0.010362         253 -1.489546
4   2009  -0.000079  0.005424         252 -0.232261
5   2010  -0.000159  0.002055         252 -1.226659
6   2011  -0.000339  0.002971         252 -1.812117
7   2012   0.000062  0.002059         250  0.476460
8   2013  -0.000209  0.001673         252 -1.981417
9   2014  -0.000070  0.002050         252 -0.545735
10  2015  -0.000100  0.002474         252 -0.643860
11  2016  -0.000205  0.003286         252 -0.988287
12  2017  -0.000226  0.002046         251 -1.751343
13  2018   0.000081  0.002707         251  0.476457
14  2019   0.000190  0.003204         252  0.941279
15  2020  -0.000836  0.010800         253 -1.230735
16  2021  -0.000034  0.004815         252 -0.112421
17  2022  -0.000142  0.005495         251 -0.410177
18  2023  -0.000598  0.005178         250 -1.826291
19  2024  -0.000326  0.003035         251 -1.702588
```

Part - B

Question 1

In [48]:

```
# Initialize an empty list to store the results for each year
portfolio_results = []
portfolio_results_cost = []
trading_cost_bps = 5
trading_cost = trading_cost_bps / 10000
```

```python
# Create an empty DataFrame to store daily portfolio returns
daily_portfolio_df = pd.DataFrame(columns=["Date", "Portfolio_Return"])
daily_portfolio_df_cost = pd.DataFrame(columns=["Date", "Portfolio_Return"])

# Loop through each year from 2006 to 2024
for year in range(2006, 2025):
    # Get the previous year's average beta (β_v) for the current year
    avg_beta_v = df_beta_stats.loc[df_beta_stats["Year"] == (year - 1), "Mean_Beta"].values[0]

    # Filter the df_grand_rank for the current year
    df_year = df_grand[df_grand.index.year == year].copy()

    # Calculate expected returns R_Ei(t,t+1) = β_v * v_i(t)
    df_year["Expected_Return"] = avg_beta_v * df_year["alpha_factor"]

    # Calculate returns for long and short positions (r_L and r_S)
    df_year = df_year.sort_values(by=["Ticker", "Date"])
    df_year["Daily_log_return_Shifted"] = df_year.groupby("Ticker")["Daily_log_return"].shift(-1)
    df_year["Daily_log_return_Shifted"].fillna(0, inplace=True)  # Assuming no return for the last day

    # Initialize signal DataFrame
    df_year["Signal"] = 0

    # Initialize a list to store daily portfolio returns
    daily_portfolio_returns = []
    daily_portfolio_returns_cost = []

    # Loop through each date and calculate the portfolio return for that day
    for date, df_date in df_year.groupby(df_year.index.date):
        N = len(df_date["Ticker"].unique())  # Number of unique stocks on this day
        if N == 0:
            continue

        N_l = int(0.2 * N)
        N_s = int(0.2 * N)

        df_date["Stock_Rank"] = df_date["Expected_Return"].rank(method="first", ascending=False)
        df_date.loc[df_date["Stock_Rank"] <= N_l, "Signal"] = 1  # Long
        df_date.loc[df_date["Stock_Rank"] > (N - N_s), "Signal"] = -1  # Short

        # Calculate portfolio return
        long_returns = df_date.loc[df_date["Signal"] == 1, "Daily_log_return_Shifted"].sum()
        short_returns = df_date.loc[df_date["Signal"] == -1, "Daily_log_return_Shifted"].sum()
        long_count = (df_date["Signal"] == 1).sum()

        portfolio_return = (long_returns - short_returns) / N_l if N_l > 0 else 0

        # Trading cost calculation
        df_date_shifted = df_date.shift(1).fillna(0)
        position_changes = (df_date_shifted["Signal"] != df_date["Signal"]).sum()
        active_stocks = (df_date["Signal"].abs() != 0).sum()
        trading_cost_penalty = trading_cost * position_changes / active_stocks if active_stocks > 0 else 0
        #trading_cost_penalty = trading_cost * position_changes
        portfolio_return_cost = portfolio_return - trading_cost_penalty

        # Append to daily portfolio returns list
        daily_portfolio_returns.append(portfolio_return)
        daily_portfolio_df = pd.concat([daily_portfolio_df, pd.DataFrame({"Date": [date], "Portfolio_Return": [po
        daily_portfolio_returns_cost.append(portfolio_return_cost)
        daily_portfolio_df_cost = pd.concat([daily_portfolio_df_cost, pd.DataFrame({"Date": [date], "Portfolio_Re

    # Compute annual return (sum of daily returns)
    annual_return = np.sum(daily_portfolio_returns)
    annual_return_cost  = np.sum(daily_portfolio_returns_cost)

    # Compute annualized return volatility (standard deviation of daily returns * sqrt(252))
    annual_volatility = np.std(daily_portfolio_returns) * np.sqrt(252)
    annual_volatility_cost  = np.std(daily_portfolio_returns_cost ) * np.sqrt(252)

    sharpe_ratio = annual_return / annual_volatility_cost              ##assuming risk free rate as 0
    sharpe_ratio_cost = annual_return_cost / annual_volatility_cost

    # Store the results for the year
    portfolio_results.append({"Year": year, "Annual Return": annual_return, "Annualized Volatility": annual_volat
```

```
    portfolio_results_cost.append({"Year": year, "Annual Return": annual_return_cost, "Annualized Volatility": an

# Set index for daily portfolio returns
daily_portfolio_df.set_index('Date', inplace=True)
daily_portfolio_df_cost.set_index('Date', inplace=True)

# Convert results to DataFrame
df_portfolio_results = pd.DataFrame(portfolio_results)
df_portfolio_results_cost = pd.DataFrame(portfolio_results_cost)
```

Question 2

In [49]:
```
daily_portfolio_df
```

Out[49]:

Date	Portfolio_Return
2006-01-03	0.005560
2006-01-04	0.001497
2006-01-05	0.003302
2006-01-06	-0.001455
2006-01-09	0.001030
...	...
2024-12-23	0.001869
2024-12-24	-0.000501
2024-12-26	0.000987
2024-12-27	-0.001329
2024-12-30	0.000000

4780 rows × 1 columns

Question 3

In [50]:
```
df_portfolio_results  #annual results
```

Out[50]:

	Year	Annual Return	Annualized Volatility	Sharpe Ratio
0	2006	0.130048	0.046117	2.818964
1	2007	0.079170	0.073012	1.083855
2	2008	0.331588	0.223993	1.480306
3	2009	0.045828	0.118626	0.386399
4	2010	0.039365	0.046998	0.837095
5	2011	0.158924	0.064150	2.478354
6	2012	-0.006986	0.045407	-0.153858
7	2013	-0.037771	0.037580	-1.005095
8	2014	0.019737	0.042442	0.464636
9	2015	0.032675	0.053299	0.613040
10	2016	0.049079	0.067459	0.727761
11	2017	0.068524	0.041835	1.637845
12	2018	-0.048233	0.051646	-0.933963
13	2019	0.069221	0.068652	1.007947
14	2020	-0.307040	0.225987	-1.358551
15	2021	0.020481	0.097534	0.209965
16	2022	-0.078204	0.103273	-0.756897
17	2023	0.109809	0.100619	1.091346
18	2024	0.126679	0.065991	1.919739

In [51]:

```
print('Best and Worst years based on annual returns')
best_year = df_portfolio_results.loc[df_portfolio_results["Annual Return"].idxmax()]
worst_year = df_portfolio_results.loc[df_portfolio_results["Annual Return"].idxmin()]

# Convert and print the year as an integer
print("\nBest Year for the Strategy:", int(best_year["Year"]), ' Annual Return:', best_year["Annual Return"],
      '  \nAnnualized Volatility:', best_year["Annualized Volatility"], '    Sharpe Ratio:', best_year["Sharpe Rat
print()
print("Worst Year for the Strategy:", int(worst_year["Year"]), '  Annual Return:', worst_year["Annual Return"],
      '  \nAnnualized Volatility:', worst_year["Annualized Volatility"], '    Sharpe Ratio:', worst_year["Sharpe R
```

```
Best and Worst years based on annual returns

Best Year for the Strategy: 2008  Annual Return: 0.3315884513673465
Annualized Volatility: 0.22399301741619793    Sharpe Ratio: 1.4803055249310313

Worst Year for the Strategy: 2020   Annual Return: -0.30703969786625424
Annualized Volatility: 0.2259870448848793    Sharpe Ratio: -1.3585506167766566
```

In [52]:

```
print('Best and Worst years based on sharpe ratio')
best_year = df_portfolio_results.loc[df_portfolio_results["Sharpe Ratio"].idxmax()]
worst_year = df_portfolio_results.loc[df_portfolio_results["Sharpe Ratio"].idxmin()]

# Convert and print the year as an integer
print("\nBest Year for the Strategy:", int(best_year["Year"]), ' Annual Return:', best_year["Annual Return"],
      '  \nAnnualized Volatility:', best_year["Annualized Volatility"], '    Sharpe Ratio:', best_year["Sharpe Rat

print("\nWorst Year for the Strategy:", int(worst_year["Year"]), '  Annual Return:', worst_year["Annual Return"],
      '  \nAnnualized Volatility:', worst_year["Annualized Volatility"], '    Sharpe Ratio:', worst_year["Sharpe R
```

```
Best and Worst years based on sharpe ratio

Best Year for the Strategy: 2006  Annual Return: 0.13004756670931872
Annualized Volatility: 0.04611707624065673    Sharpe Ratio: 2.8189641910042003

Worst Year for the Strategy: 2020   Annual Return: -0.30703969786625424
Annualized Volatility: 0.2259870448848793    Sharpe Ratio: -1.3585506167766566
```

Question 4

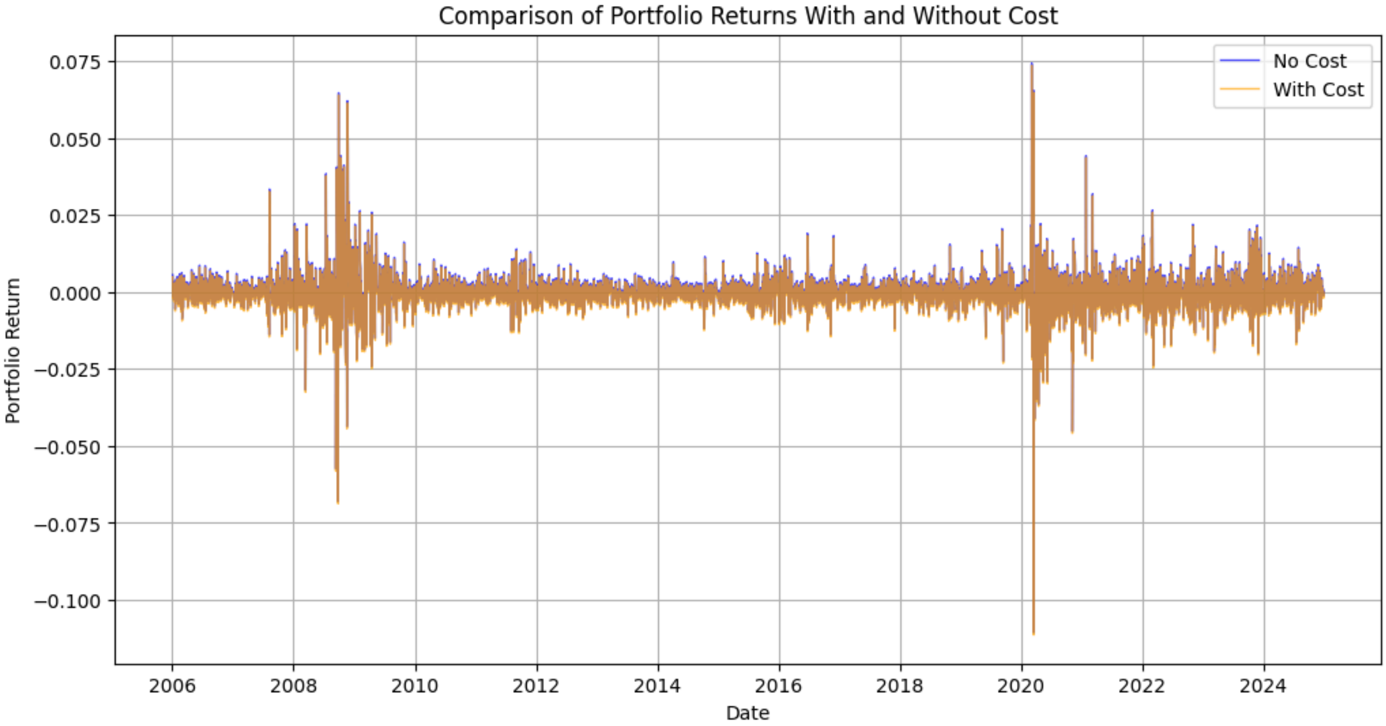
In [53]:
```
# Assuming you have loaded both DataFrames
merged_df = daily_portfolio_df.merge(daily_portfolio_df_cost, on="Date", suffixes=("_no_cost", "_with_cost"))
print('Comparison of Portfolio returns with and without cost')
# Display the merged DataFrame
print(merged_df)
```

```
Comparison of Portfolio returns with and without cost
            Portfolio_Return_no_cost  Portfolio_Return_with_cost
Date
2006-01-03                  0.005560                    0.004895
2006-01-04                  0.001497                    0.000785
2006-01-05                  0.003302                    0.002587
2006-01-06                 -0.001455                   -0.002160
2006-01-09                  0.001030                    0.000325
...                              ...                         ...
2024-12-23                  0.001869                    0.001176
2024-12-24                 -0.000501                   -0.001196
2024-12-26                  0.000987                    0.000317
2024-12-27                 -0.001329                   -0.001999
2024-12-30                  0.000000                   -0.000748

[4780 rows x 2 columns]
```

In [54]:
```
plt.figure(figsize=(12, 6))
plt.plot(merged_df.index, merged_df["Portfolio_Return_no_cost"], label="No Cost", linewidth=1, color="blue", alph
plt.plot(merged_df.index, merged_df["Portfolio_Return_with_cost"], label="With Cost", linewidth=1, color="orange"

plt.xlabel("Date")
plt.ylabel("Portfolio Return")
plt.title("Comparison of Portfolio Returns With and Without Cost")
plt.legend()
plt.grid(True)
plt.show()
```



In [55]:
```
df_portfolio_results_cost  # annual returns with cost
```

Out[55]:

	Year	Annual Return	Annualized Volatility	Sharpe Ratio
0	2006	-0.044449	0.046133	-0.963489
1	2007	-0.096506	0.073045	-1.321191
2	2008	0.155178	0.224000	0.692757
3	2009	-0.130372	0.118602	-1.099240
4	2010	-0.137135	0.047026	-2.916168
5	2011	-0.017617	0.064125	-0.274735
6	2012	-0.181403	0.045407	-3.995019
7	2013	-0.213847	0.037579	-5.690583
8	2014	-0.156005	0.042478	-3.672588
9	2015	-0.143752	0.053300	-2.697051
10	2016	-0.126910	0.067438	-1.881888
11	2017	-0.106146	0.041838	-2.537063
12	2018	-0.222570	0.051644	-4.309706
13	2019	-0.105531	0.068675	-1.536673
14	2020	-0.483124	0.226005	-2.137666
15	2021	-0.155175	0.097546	-1.590786
16	2022	-0.252754	0.103321	-2.446289
17	2023	-0.065089	0.100618	-0.646898
18	2024	-0.048229	0.065988	-0.730875

In [56]:

```
print('Best and Worst years based on annual returns taking cost into account')
best_year_cost = df_portfolio_results_cost.loc[df_portfolio_results_cost["Annual Return"].idxmax()]
worst_year_cost = df_portfolio_results_cost.loc[df_portfolio_results_cost["Annual Return"].idxmin()]

print("\nBest Year for the Strategy:", int(best_year_cost["Year"]), ' Annual Return:', best_year_cost["Annual Ret
      '  \nAnnualized Volatility:', best_year_cost["Annualized Volatility"], '   Sharpe Ratio:', best_year_cost["

print("\nWorst Year for the Strategy:", int(worst_year_cost["Year"]), '  Annual Return:', worst_year_cost["Annual
      '  \nAnnualized Volatility:', worst_year_cost["Annualized Volatility"], '   Sharpe Ratio:', worst_year_cost
```

```
Best and Worst years based on annual returns taking cost into account

Best Year for the Strategy: 2008  Annual Return: 0.1551776449157336
Annualized Volatility: 0.22400000931077757    Sharpe Ratio: 0.6927573145786801

Worst Year for the Strategy: 2020   Annual Return: -0.4831238818001116
Annualized Volatility: 0.2260053428077248    Sharpe Ratio: -2.1376657551460263
```

In [57]:

```
print('Best and Worst years based on sharpe ratio taking cost into account')
best_year_cost = df_portfolio_results_cost.loc[df_portfolio_results_cost["Sharpe Ratio"].idxmax()]
worst_year_cost = df_portfolio_results_cost.loc[df_portfolio_results_cost["Sharpe Ratio"].idxmin()]

print("\nBest Year for the Strategy:", int(best_year_cost["Year"]), ' Annual Return:', best_year_cost["Annual Ret
      '  \nAnnualized Volatility:', best_year_cost["Annualized Volatility"], '   Sharpe Ratio:', best_year_cost["

print("\nWorst Year for the Strategy:", int(worst_year_cost["Year"]), '  Annual Return:', worst_year_cost["Annual
      '  \nAnnualized Volatility:', worst_year_cost["Annualized Volatility"], '   Sharpe Ratio:', worst_year_cost
```

```
Best and Worst years based on sharpe ratio taking cost into account

Best Year for the Strategy: 2008  Annual Return: 0.1551776449157336
Annualized Volatility: 0.22400000931077757    Sharpe Ratio: 0.6927573145786801

Worst Year for the Strategy: 2013   Annual Return: -0.21384729452605994
Annualized Volatility: 0.03757915366110958    Sharpe Ratio: -5.690583041186718
```

In [ ]: