Optimization Project 2 - Integer Programming

Aman Sharma (as235548), Destin Blanchard (ddb3338), Luke Leon (ljl2556), Sarah Stephens (sgs2623)

Overview

Background

In equity management, investment strategies are generally split into two types: "active" and "passive." A common passive approach is using index funds, which aim to track the performance of a market index like the NASDAQ-100. However, buying every stock in an index can be expensive and complicated. Instead, by selecting a smaller group of *m* stocks rather than all *n* stocks in the index, investors can effectively capture the index's performance in a simpler, more cost-effective way, with less frequent rebalancing.

Objective

This project aims to build a cost-effective index fund that closely tracks the NASDAQ-100, using a smaller, optimized portfolio of *m* stocks rather than including every stock in the index. We'll use integer programming to select the best stocks based on their similarity (like return correlations) and then determine how much of each stock to hold using linear programming. As an alternate technique, we will also use Mixed Integer Programming (MIP) to determine the weights. Finally, we'll test our index fund's performance against the NASDAQ-100 with different values of m.

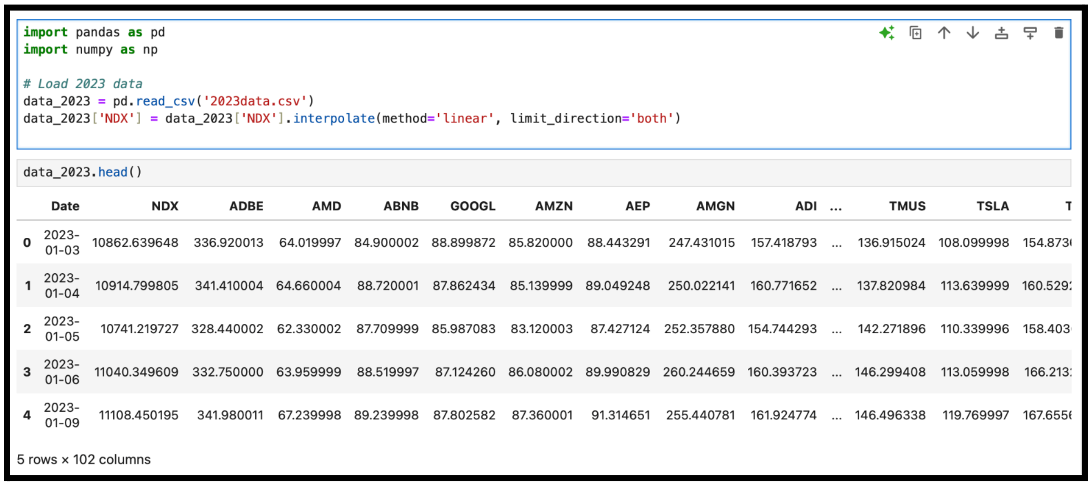
Data Preprocessing

Missing Value Imputation

Missing Values in Index

For the NDX column, we used linear interpolation to fill any missing values. For a single missing value in the time series, we impute it by averaging the NDX prices of the preceding and following trading days. If there are multiple consecutive missing values, we take the NDX prices from the days before and after the gap as references and fill in each missing value through linear interpolation within that range.We believe this imputation strategy is justified because we ultimately calculate returns using each NDX value. If these values were imputed with zero, then returns between the last known NDX price and the next available one would show a -100% change

{*(0 - last_available_price)/last_available_price*)}, which does not accurately reflect the index's actual performance. On the other hand, if they were left as N/A, then the returns would have been N/A.



```python
import pandas as pd
import numpy as np

# Load 2023 data
data_2023 = pd.read_csv('2023data.csv')
data_2023['NDX'] = data_2023['NDX'].interpolate(method='linear', limit_direction='both')
```

```python
data_2023.head()
```

	Date	NDX	ADBE	AMD	ABNB	GOOGL	AMZN	AEP	AMGN	ADI	...	TMUS	TSLA	T
0	2023-01-03	10862.639648	336.920013	64.019997	84.900002	88.899872	85.820000	88.443291	247.431015	157.418793	...	136.915024	108.099998	154.873
1	2023-01-04	10914.799805	341.410004	64.660004	88.720001	87.862434	85.139999	89.049248	250.022141	160.771652	...	137.820984	113.639999	160.529
2	2023-01-05	10741.219727	328.440002	62.330002	87.709999	85.987083	83.120003	87.427124	252.357880	154.744293	...	142.271896	110.339996	158.403
3	2023-01-06	11040.349609	332.750000	63.959999	88.519997	87.124260	86.080002	89.990829	260.244659	160.393723	...	146.299408	113.059998	166.213
4	2023-01-09	11108.450195	341.980011	67.239998	89.239998	87.802582	87.360001	91.314651	255.440781	161.924774	...	146.496338	119.769997	167.655

5 rows × 102 columns

Missing Values in Stock Prices:

The key question that we sought to answer when dealing with missing values for individual stocks was: *Why are there missing values?* As it turns out, it is perfectly normal for an index to include new stocks through the year, and drop others out. While building a fund that tracks an index, this is a situation that we will have to deal with pretty regularly.

Now, given that the fund cannot predetermine which stocks the index will include or drop in the future, we decided to not fill in missing stock price data. This is because doing so would not be realistic in the production environment, and if we impute those values with external data sources (such as Yahoo Finance) while training, then we may run into serious data leakage issues.

However, for the optimization algorithm to be able to do matrix operations without running into any challenges, we imputed the missing values in the Returns matrix (calculated using the daily stock prices data {1 - P_today / P_prev}) with zero. Note that this is practically the same as imputing the missing values in the original data with a constant value.

Various approaches to handling missing values yield different results; therefore, it is something to carefully consider. However, we are confident with our method—given our small dataset, our method balances the risk and benefit of heavily reducing our available data by deleting all rows with missing values. Additionally, our method provides sound, intuitive results at the end of our analysis.

Dropping first time period in Returns matrix:

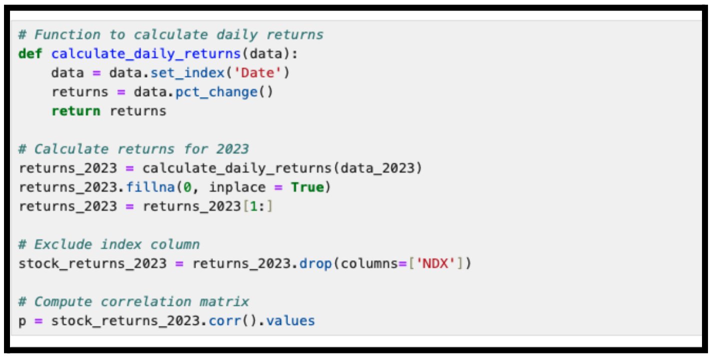
By design, when calculating returns from a time series of stock prices, the result yields one less value than the total number of time steps in the price data. We dropped the new null values corresponding to the first time step when generating the returns matrix.

Algorithm Overview

There are two methods to create a well balanced stock portfolio that tracks the NASDAQ-100. Below are step-by-step explanations of each.

<u>Method 1: Separation of Stock Selection & Weight Optimization</u>

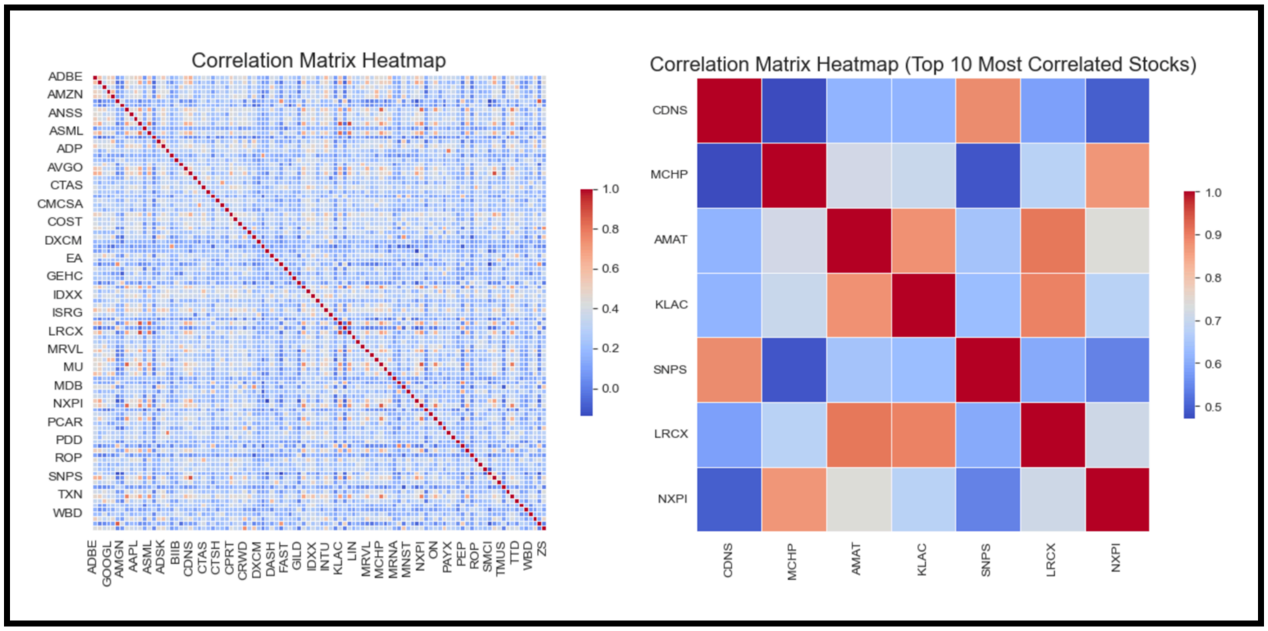
1. Define function to calculate the daily returns of the stocks and index



```
# Function to calculate daily returns
def calculate_daily_returns(data):
    data = data.set_index('Date')
    returns = data.pct_change()
    return returns

# Calculate returns for 2023
returns_2023 = calculate_daily_returns(data_2023)
returns_2023.fillna(0, inplace = True)
returns_2023 = returns_2023[1:]

# Exclude index column
stock_returns_2023 = returns_2023.drop(columns=['NDX'])

# Compute correlation matrix
p = stock_returns_2023.corr().values
```

2. Create correlation (similarity) matrix

   Our preferred measure of similarity is correlation between the returns, which is calculated as shown in the previous snippet. Below are two correlation heatmaps. The one on the left (though less informative) is the correlation of all stocks, and the one on the right is the correlation of top 10 stocks



3. Define function to select stocks

   Stock selection is a rather straightforward integer programming problem. The objective is to select those stocks that maximize the similarity of the fund to the index. It is mathematically shown below and the code implementation follows. The problem is designed such that the most similar *m* stocks (to the stocks in the index) will be chosen for the fund. The constraints ensure, 1) that exactly *m* stocks are selected, and 2) exactly one proxy is selected for each stock in the index.

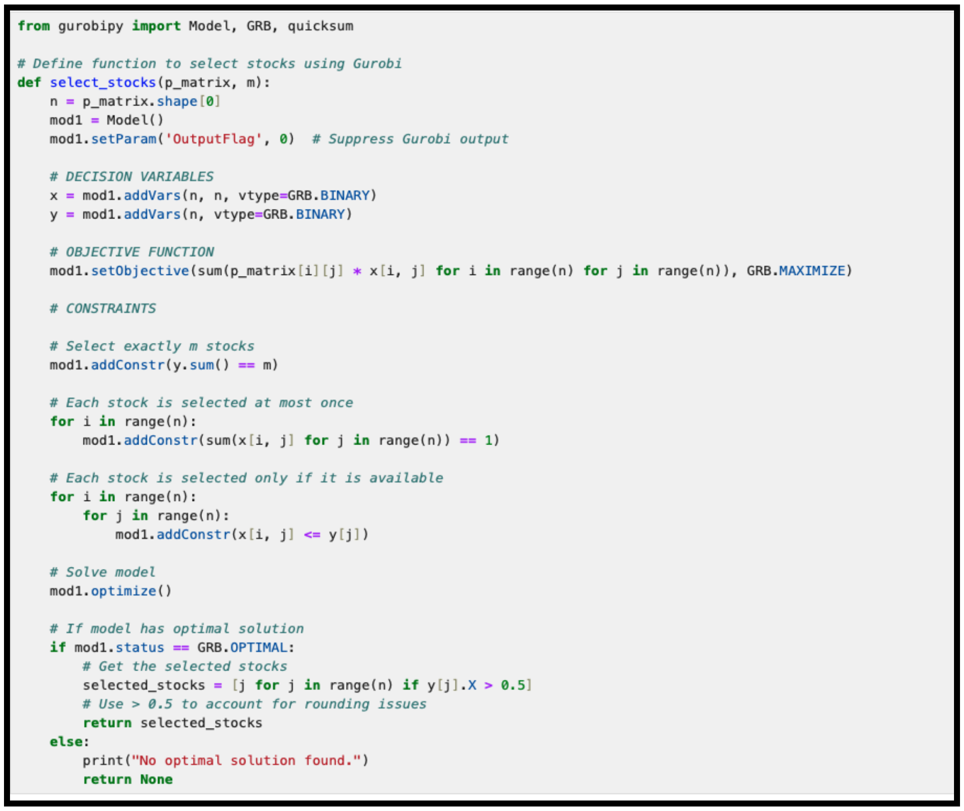
$$\max_{x,y} \sum_{i=1}^{n} \sum_{j=1}^{n} \rho_{ij} x_{ij}$$

$$s.t. \sum_{j=1}^{n} y_j = m.$$

$$\sum_{j=1}^{n} x_{ij} = 1 \quad for\ i = 1,2,\dots,n$$

$$x_{ij} \le y_j \qquad for\ i,j = 1,2,\dots,n$$

$$x_{ij}, y_j \in \{0,1\}$$



```python
from gurobipy import Model, GRB, quicksum

# Define function to select stocks using Gurobi
def select_stocks(p_matrix, m):
    n = p_matrix.shape[0]
    mod1 = Model()
    mod1.setParam('OutputFlag', 0)  # Suppress Gurobi output

    # DECISION VARIABLES
    x = mod1.addVars(n, n, vtype=GRB.BINARY)
    y = mod1.addVars(n, vtype=GRB.BINARY)

    # OBJECTIVE FUNCTION
    mod1.setObjective(sum(p_matrix[i][j] * x[i, j] for i in range(n) for j in range(n)), GRB.MAXIMIZE)

    # CONSTRAINTS

    # Select exactly m stocks
    mod1.addConstr(y.sum() == m)

    # Each stock is selected at most once
    for i in range(n):
        mod1.addConstr(sum(x[i, j] for j in range(n)) == 1)

    # Each stock is selected only if it is available
    for i in range(n):
        for j in range(n):
            mod1.addConstr(x[i, j] <= y[j])

    # Solve model
    mod1.optimize()

    # If model has optimal solution
    if mod1.status == GRB.OPTIMAL:
        # Get the selected stocks
        selected_stocks = [j for j in range(n) if y[j].X > 0.5]
        # Use > 0.5 to account for rounding issues
        return selected_stocks
    else:
        print("No optimal solution found.")
        return None
```

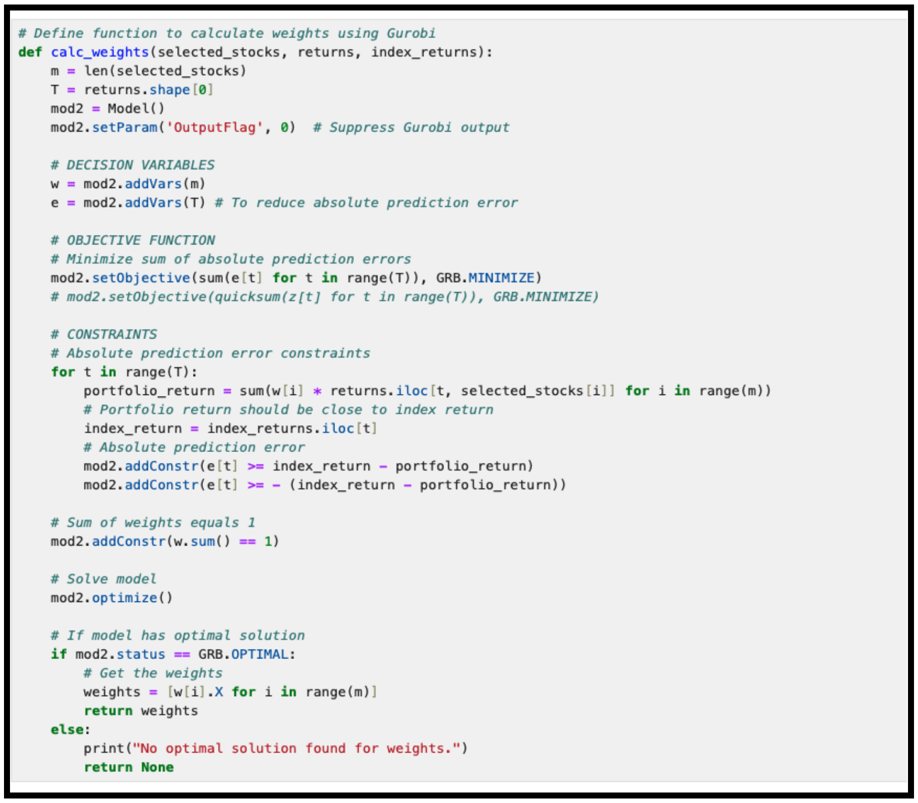
4. Define function to calculate the weights of the selected stocks

$$\min_{w} \sum_{t=1}^{T} \left| q_t - \sum_{i=1}^{m} w_i r_{it} \right|$$

$$s.t. \sum_{i=1}^{m} w_i = 1$$

$$w_i \geq 0.$$

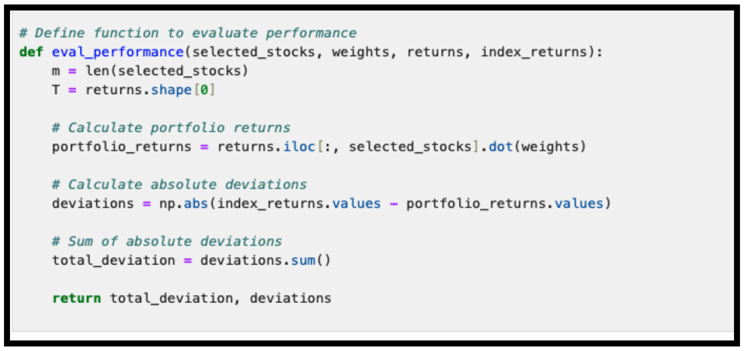
An absolute value objective function is achieved by introducing an additional decision variable and two constraints. As shown in the code below, our additional decision variable is *e*, and two corresponding constraints (apart from the weight constraint) have been introduced.



```python
# Define function to calculate weights using Gurobi
def calc_weights(selected_stocks, returns, index_returns):
    m = len(selected_stocks)
    T = returns.shape[0]
    mod2 = Model()
    mod2.setParam('OutputFlag', 0)  # Suppress Gurobi output

    # DECISION VARIABLES
    w = mod2.addVars(m)
    e = mod2.addVars(T) # To reduce absolute prediction error

    # OBJECTIVE FUNCTION
    # Minimize sum of absolute prediction errors
    mod2.setObjective(sum(e[t] for t in range(T)), GRB.MINIMIZE)
    # mod2.setObjective(quicksum(z[t] for t in range(T)), GRB.MINIMIZE)

    # CONSTRAINTS
    # Absolute prediction error constraints
    for t in range(T):
        portfolio_return = sum(w[i] * returns.iloc[t, selected_stocks[i]] for i in range(m))
        # Portfolio return should be close to index return
        index_return = index_returns.iloc[t]
        # Absolute prediction error
        mod2.addConstr(e[t] >= index_return - portfolio_return)
        mod2.addConstr(e[t] >= - (index_return - portfolio_return))

    # Sum of weights equals 1
    mod2.addConstr(w.sum() == 1)

    # Solve model
    mod2.optimize()

    # If model has optimal solution
    if mod2.status == GRB.OPTIMAL:
        # Get the weights
        weights = [w[i].X for i in range(m)]
        return weights
    else:
        print("No optimal solution found for weights.")
        return None
```

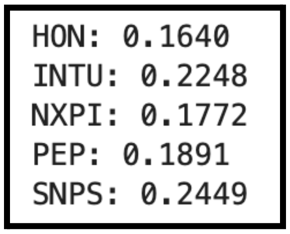
5. Define function to evaluate the performance of the portfolio

   Below is how we define the evaluate performance function. Note that while the function outputs total deviations, care has been taken to report mean deviations since the length of the train and test time series may be different.



```python
# Define function to evaluate performance
def eval_performance(selected_stocks, weights, returns, index_returns):
    m = len(selected_stocks)
    T = returns.shape[0]

    # Calculate portfolio returns
    portfolio_returns = returns.iloc[:, selected_stocks].dot(weights)

    # Calculate absolute deviations
    deviations = np.abs(index_returns.values - portfolio_returns.values)

    # Sum of absolute deviations
    total_deviation = deviations.sum()

    return total_deviation, deviations
```

6. Run optimization for when the number of stocks is 5 (*m* = 5)



```
HON: 0.1640
INTU: 0.2248
NXPI: 0.1772
PEP: 0.1891
SNPS: 0.2449
```

7. Run optimization for different values of *m*

   Figure A (see Appendix) visually shows how the index is tracked for arbitrary *m* = {2,50} and an arbitrary month {May 2023}. Observe that the tracking performance of the orange line (*m*=50) is significantly better than the blue line (*m*=2).

   Figure B (Appendix) compares the training and testing performance (test data is 2024 index performance). Figure B compares the total deviation or total absolute error on train and test data. In Figure C, we compare the mean absolute errors. Note that it is important to compare mean errors because the time periods of the train and test file are different. As expected, the mean training sample error nears zero once we include all index stocks (*m*=100). Another interesting thing to note is that for lower values of *m* (<60) insample and out of sample errors thread closely; however, as *m* increases, the gap increases significantly. To draw a parallel with machine learning, the models on left side of the graphs are high bias models (less complex - higher training and testing errors), and the ones on the right are high variance models (more complex, signified by higher training error than testing error)

<u>Method 2: Multiple Integer Programming (MIP)</u>

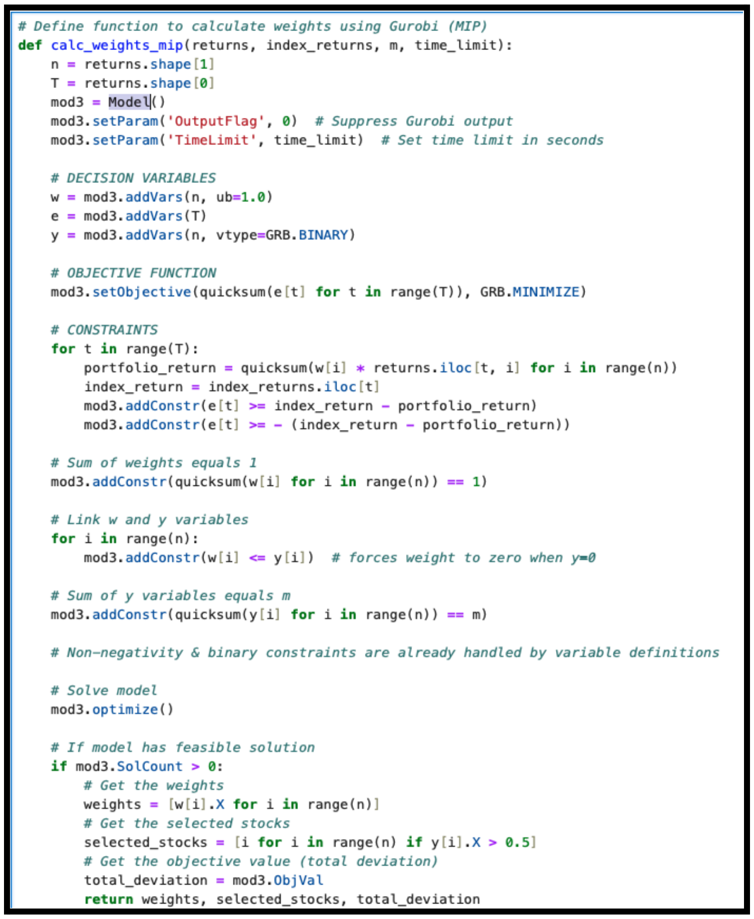
1. Define *one* function to optimize the selection of stocks and weights

   The following is a reformulation of the weight selection problem to be an MIP that constrains the number of non-zero weights to be an integer. We use the prior weight selection problem and replace *m* with *n* so to optimize over *all* weights:

$$\boxed{\min_{w} \sum_{t=1}^{T} \left| q_t - \sum_{i=1}^{n} w_i r_{it} \right|}$$

Below is the function we use to calculate the weights using MIP. A time limit is applied such that the best feasible solution found will be returned in the event the time limit is reached before the optimal solution is found.

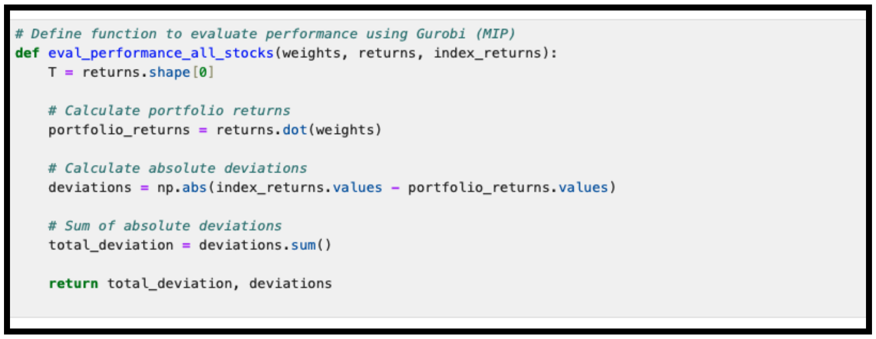
From an implementation perspective, the MIP problem formulation introduces a big "M" constraint on weights. The big "M" value that allows weights to grow sufficiently is M=1. That has been reflected in the condition, *w_i <= M * y_i*.



```python
# Define function to calculate weights using Gurobi (MIP)
def calc_weights_mip(returns, index_returns, m, time_limit):
    n = returns.shape[1]
    T = returns.shape[0]
    mod3 = Model()
    mod3.setParam('OutputFlag', 0)  # Suppress Gurobi output
    mod3.setParam('TimeLimit', time_limit)  # Set time limit in seconds

    # DECISION VARIABLES
    w = mod3.addVars(n, ub=1.0)
    e = mod3.addVars(T)
    y = mod3.addVars(n, vtype=GRB.BINARY)

    # OBJECTIVE FUNCTION
    mod3.setObjective(quicksum(e[t] for t in range(T)), GRB.MINIMIZE)

    # CONSTRAINTS
    for t in range(T):
        portfolio_return = quicksum(w[i] * returns.iloc[t, i] for i in range(n))
        index_return = index_returns.iloc[t]
        mod3.addConstr(e[t] >= index_return - portfolio_return)
        mod3.addConstr(e[t] >= - (index_return - portfolio_return))

    # Sum of weights equals 1
    mod3.addConstr(quicksum(w[i] for i in range(n)) == 1)

    # Link w and y variables
    for i in range(n):
        mod3.addConstr(w[i] <= y[i])  # forces weight to zero when y=0

    # Sum of y variables equals m
    mod3.addConstr(quicksum(y[i] for i in range(n)) == m)

    # Non-negativity & binary constraints are already handled by variable definitions

    # Solve model
    mod3.optimize()

    # If model has feasible solution
    if mod3.SolCount > 0:
        # Get the weights
        weights = [w[i].X for i in range(n)]
        # Get the selected stocks
        selected_stocks = [i for i in range(n) if y[i].X > 0.5]
        # Get the objective value (total deviation)
        total_deviation = mod3.ObjVal
        return weights, selected_stocks, total_deviation
```

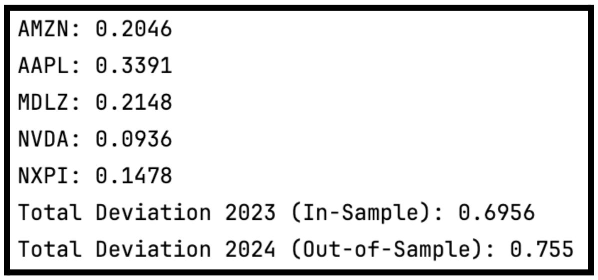
2. Define function to evaluate the performance of the portfolio

   Below is the evaluate performance function for MIP. It is very similar to the evaluate performance function from Method 1, again reporting mean deviations.



```python
# Define function to evaluate performance using Gurobi (MIP)
def eval_performance_all_stocks(weights, returns, index_returns):
    T = returns.shape[0]

    # Calculate portfolio returns
    portfolio_returns = returns.dot(weights)

    # Calculate absolute deviations
    deviations = np.abs(index_returns.values - portfolio_returns.values)

    # Sum of absolute deviations
    total_deviation = deviations.sum()

    return total_deviation, deviations
```

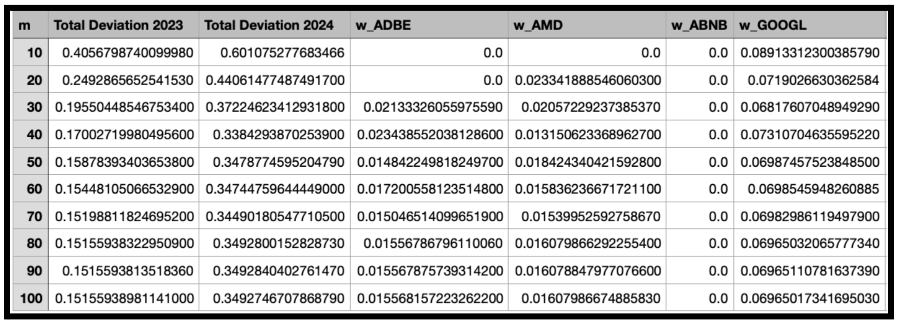
3. Run optimization for when the number of stocks is 5 (*m* = 5)



```
AMZN: 0.2046
AAPL: 0.3391
MDLZ: 0.2148
NVDA: 0.0936
NXPI: 0.1478
Total Deviation 2023 (In-Sample): 0.6956
Total Deviation 2024 (Out-of-Sample): 0.755
```

4. Run optimization for different values of *m* (applying time limits)

   Snippet of outputted CSV file:



m	Total Deviation 2023	Total Deviation 2024	w_ADBE	w_AMD	w_ABNB	w_GOOGL
10	0.4056798740099980	0.601075277683466	0.0	0.0	0.0	0.08913312300385790
20	0.2492865652541530	0.44061477487491700	0.0	0.023341888546060300	0.0	0.0719026630362584
30	0.19550448546753400	0.37224623412931800	0.02133326055975590	0.02057229237385370	0.0	0.06817607048949290
40	0.17002719980495600	0.3384293870253900	0.023438552038128600	0.013150623368962700	0.0	0.07310704635595220
50	0.15878393403653800	0.3478774595204790	0.014842249818249700	0.018424340421592800	0.0	0.06987457523848500
60	0.15448105066532900	0.34744759644449000	0.017200558123514800	0.015836236671721100	0.0	0.0698545948260885
70	0.15198811824695200	0.34490180547710500	0.015046514099651900	0.01539952592758670	0.0	0.06982986119497900
80	0.15155938322950900	0.3492800152828730	0.01556786796110060	0.016079866292255400	0.0	0.06965032065777340
90	0.1515593813518360	0.3492840402761470	0.015567875739314200	0.016078847977076600	0.0	0.06965110781637390
100	0.15155938981141000	0.3492746707868790	0.015568157223262200	0.01607986674885830	0.0	0.06965017341695030

Weights were calculated for the MIP problem, for *m* from 10 to 100, with increments of 10. Figure D and Figure E (see Appendix) reflect the total and mean absolute errors, respectively. Both graphs show a comparative view of 2023 (in sample errors) and 2024 (out of sample errors). A thing to note is at *m* = 40, the out of sample errors hits a minima. Beyond this point, the incremental value of including stocks in the funds diminishes.

Feasibility

<u>Strengths</u>

Method 1:

- *Computationally efficient*: Gurobipy can handle this method quickly because the problem is simplified into smaller steps that reduces the computation time. Thus, we can arrive at a solution much faster—a lower cost to our business.

- *Scalability*: Related to computational efficiency, separating the steps of stock selection and weight optimization allows the algorithm to effectively scale to larger numbers of stocks. This would allow us to more easily expand our portfolios.

- *Flexibility*: The segmentation of tasks provides us the freedom to customize parameters in one step without worrying about the other. For instance, if we need to change the objective function for selecting stock to address a different financial concern—such as volatility—we can do so without worrying about the impact on weight optimization.

- *Interpretability*: The separation of steps makes the algorithm easier to follow and understand, reducing confusion that may arise when combining multiple integer programming models.

Method 2:

- *All-Inclusive Optimization*: Because the selection of the stocks and weights are combined into a single step, tracking error is minimized. This performance is due to the model's ability to consider all combinations of stocks and directly optimize the model with the specific intention of minimizing tracking error.

- *Out-of-Sample Performance*: Given the model's increased flexibility in stock selection and improved optimization of weights, performance on new data is likely to be strong.

<u>Challenges</u>

Method 1:

- *Oversimplification*: By choosing stocks based on similarity prior to weight optimization, the selection process can become greedy. This method fails to take into account the full picture (all possible combinations of stocks and weights) before selecting the optimal stocks, which can lead to higher error rates.

    - There are other elements that contribute to a stock's performance aside from similarity of returns, which are not accounted for in either model. For example, stocks can represent a variety of industries, which each have their own characteristics and can be influenced differently by current events.

- *In-Sample Bias*: The model might not perform well on new data given that the stocks and weights are selected based on data from 2023. Given the highly variable nature of the stock market, overfitting should be monitored.

Method 2:

- *Computationally complex*: This method is very time consuming for Gurobipy to solve, taking 6-10 hours to find a solution. Thus, the MIP approach is not ideal for solving urgent problems, and its computational complexity may result in higher costs for our firm.

- *Difficult to scale*: As a result of being computationally intensive, the MIP method would be more difficult to solve as n, the total number of stocks, increases. An organization may also be involved in tracking multiple indices, in such a scenario, it may be too time consuming to run MIP for all indices one is interested in tracking.

- *Time Constraints*: The time constraints applied to the MIP method can make the model more efficient in finding a solution, but they can also lead to degraded performance.

Results

Findings

For both Method 1 and Method 2, we tracked the portfolio performance for different numbers of stocks as seen in Figure F (for out of sample) and Figure G (for in sample - see Appendix). Comparing the two graphs, the in and out of sample deviation for Method 2 is less than or equal to that of Method 1 for all values of *m*, indicating overall better performance. While both graphs show a general decrease in error as the number of stocks increases, this decrease is much more sharp for Method 2, and there is clear evidence of **diminishing returns** for adding more stocks to the portfolio **after 40 stocks**, whereas diminishing returns aren't reached for Method 1 until around 90 stocks. Furthermore, for Method 2 the out-of-sample deviation actually slightly increases after 40 stocks. When building the optimal portfolio, we want the best performance out-of-sample while trying to minimize the number of stocks to keep costs down; therefore, the optimal number of stocks to include in the fund is 40.

Future Considerations

To prioritize low tracking error for our clients by using Method 2, we should consider expanding our computational resources. Investing in additional computational power would allow us to scale the optimization of Method 2 more efficiently. Parallel computation can be leveraged to enhance speed.

To enhance Method 1's performance on future data, we could integrate other stock performance factors into the selection process beyond just similarity. Incorporating metrics such as risk or liquidity could provide more robust results.

Additionally, using an API like Yahoo Finance to supplement missing NDX values with actual historical data, rather than interpolating between values, would make our data set more accurate and reflective of real market conditions. This is difficult to automate as the desired data changes based on the specific index, time period, and stocks the client is considering.

Recommendations

The decision to choose between Method 1 and Method 2 ultimately relies on three key factors: computational resources, the importance of minimizing tracking error, and the number of stocks available for selection. While an efficient and interpretable model offers benefits in certain situations, the potential advantage of achieving the lowest possible tracking error might justify the additional computational costs in others. Therefore, we recommend carefully weighing the strengths and challenges of each method and applying them accordingly.

In case the priority for our team is to minimize the tracking error, the recommendation would be to use Method 2 with $m = 40$ (the point beyond which returns diminish). The portfolio construction is shown in Figure H (see Appendix).

Appendix

Figure A

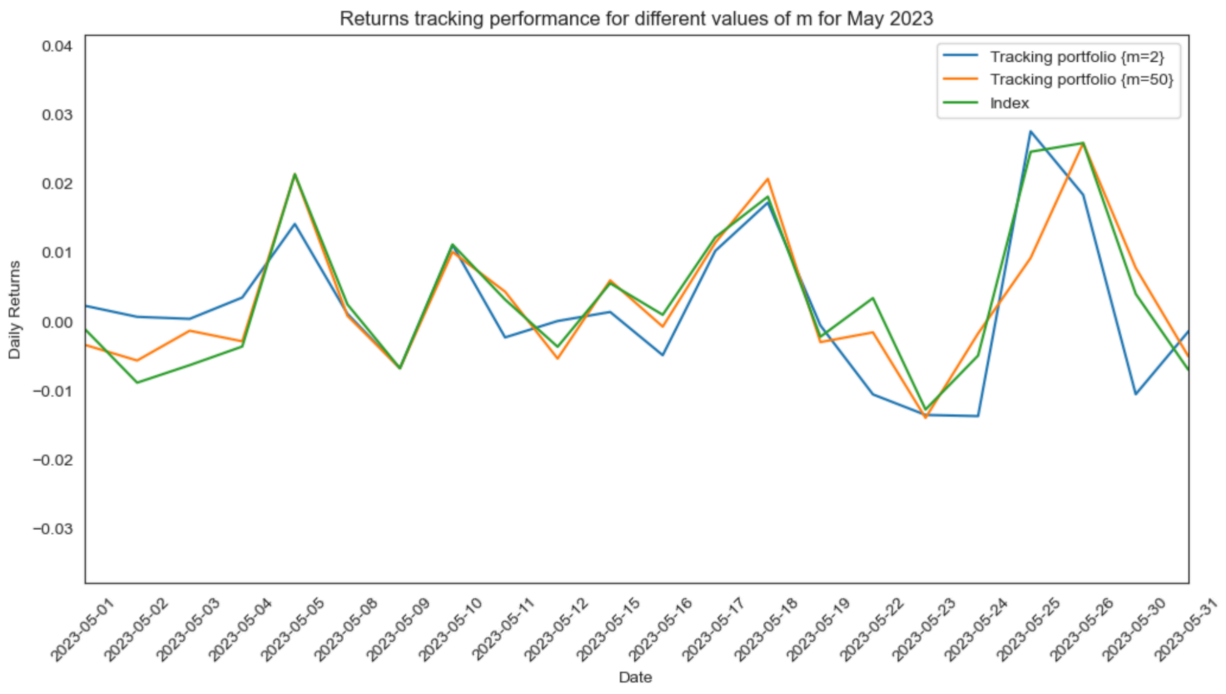


Figure B

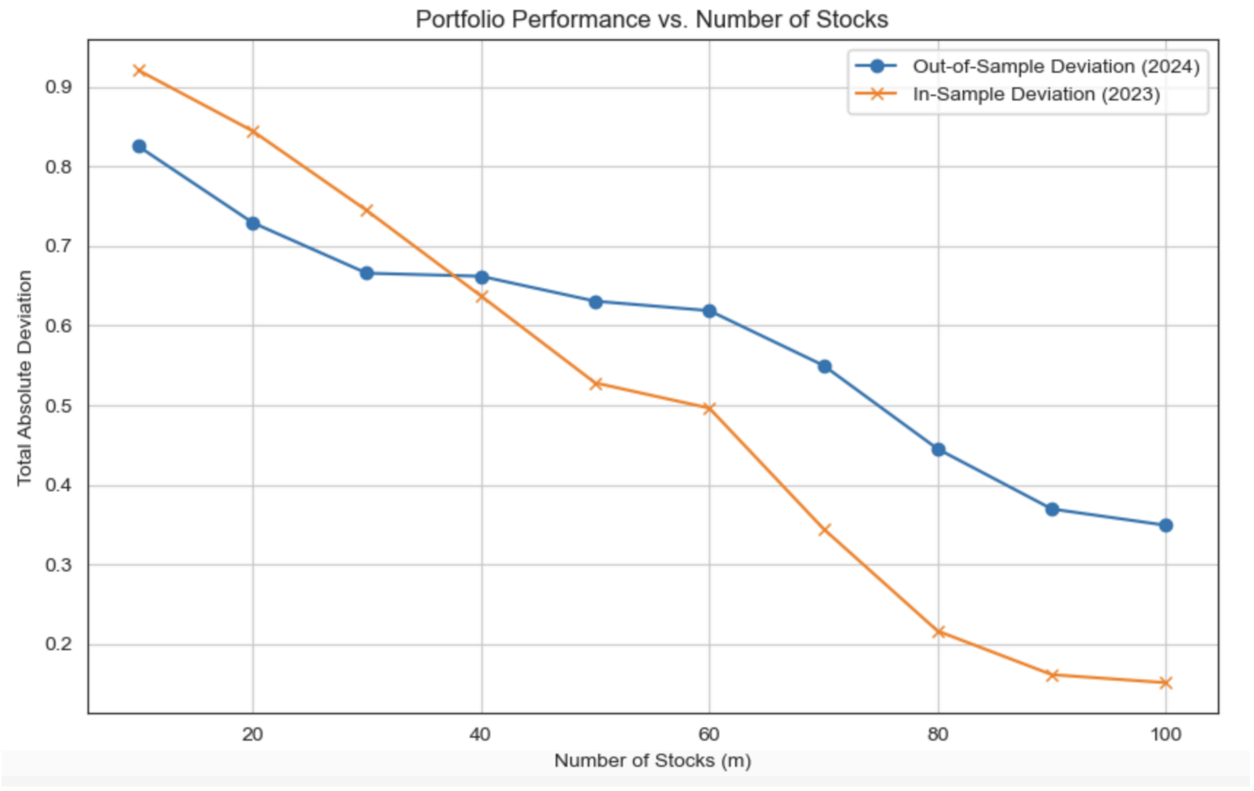


Figure C

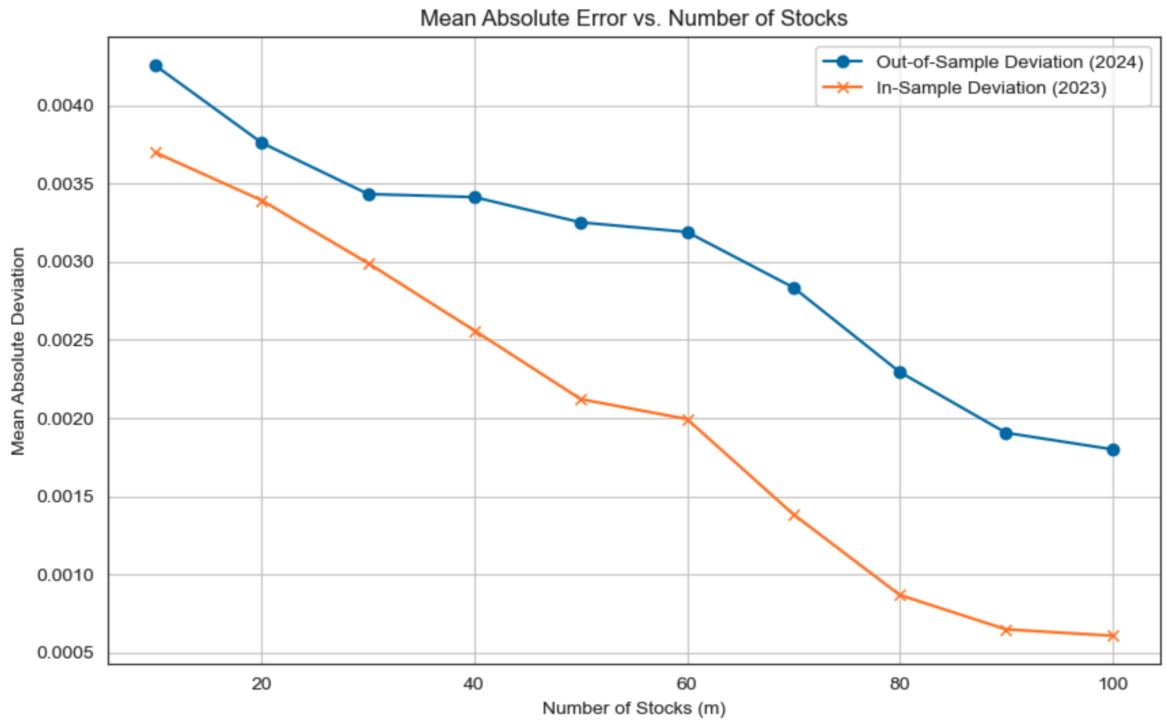


Figure D

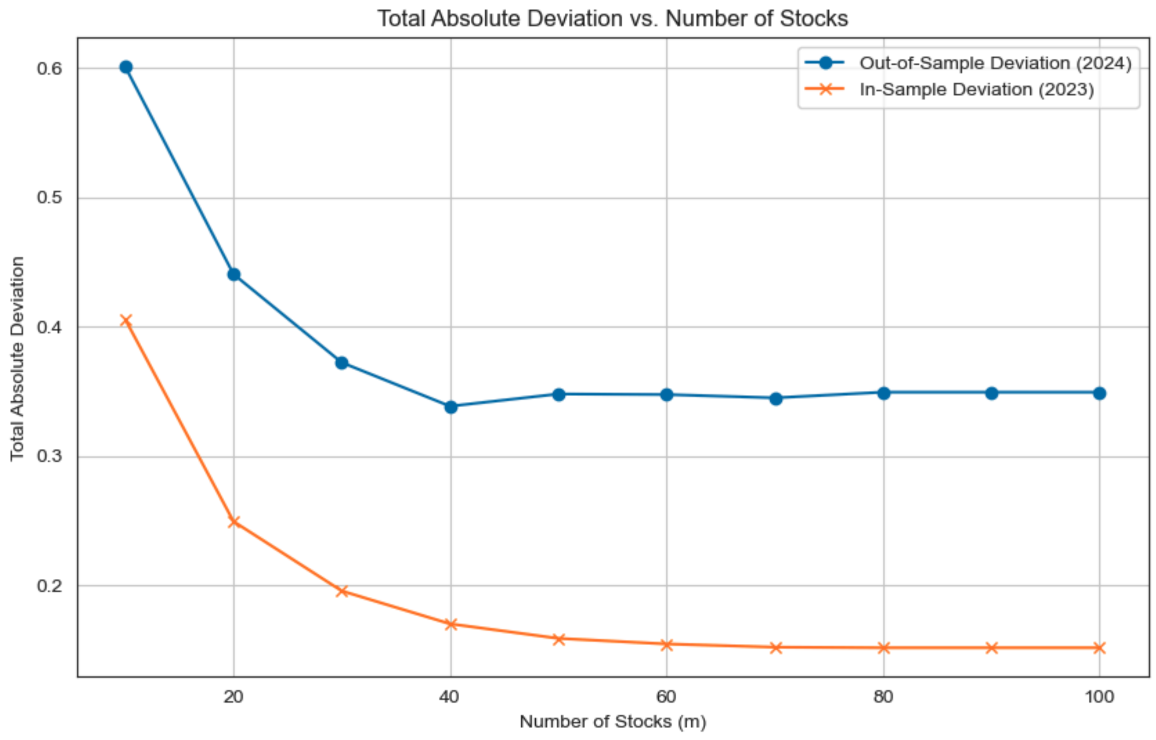


Figure E

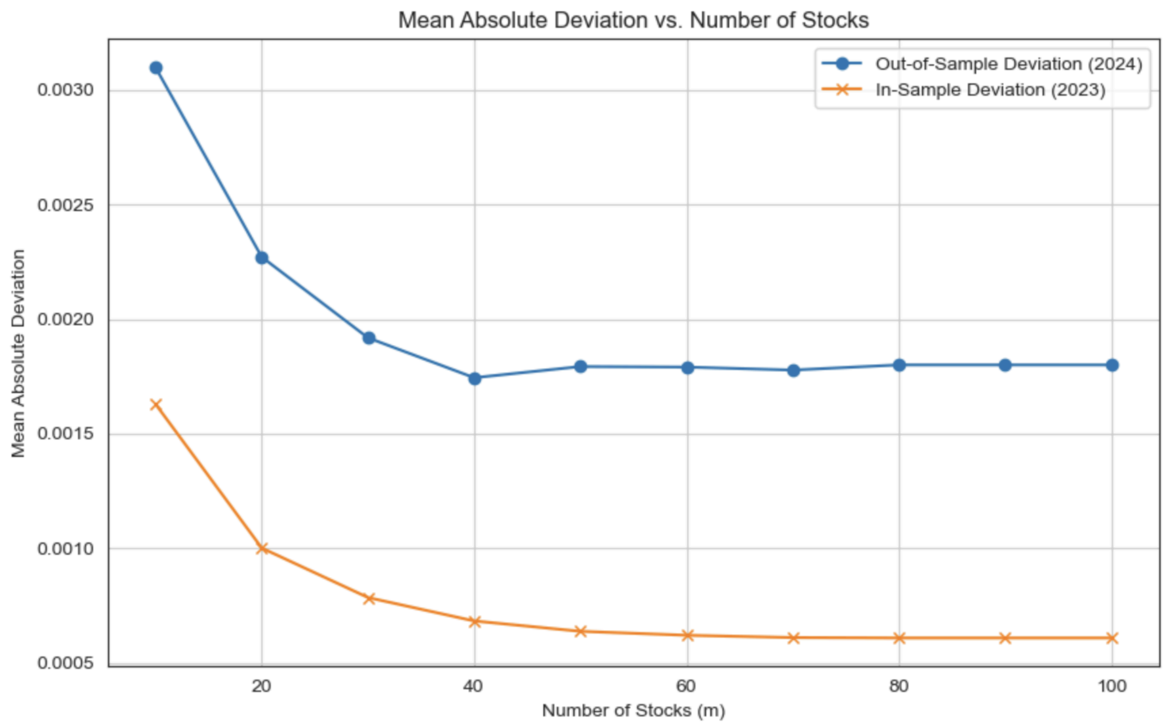


Figure F

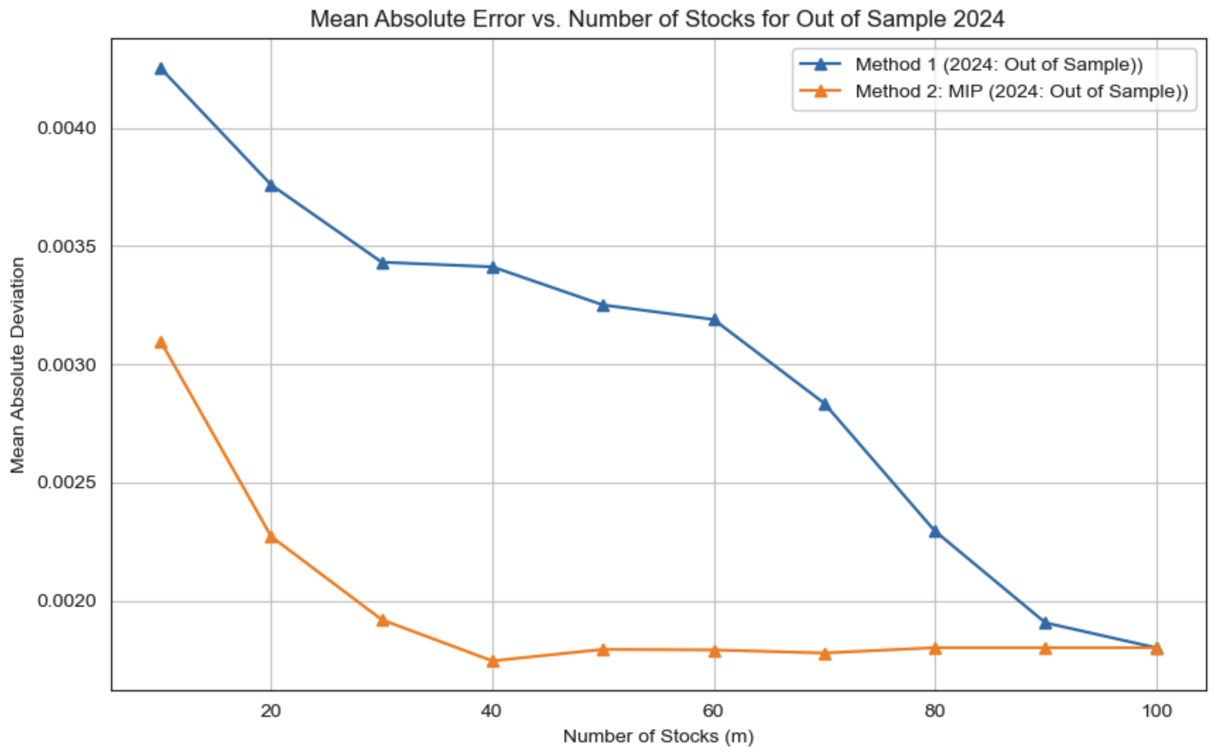


Figure G

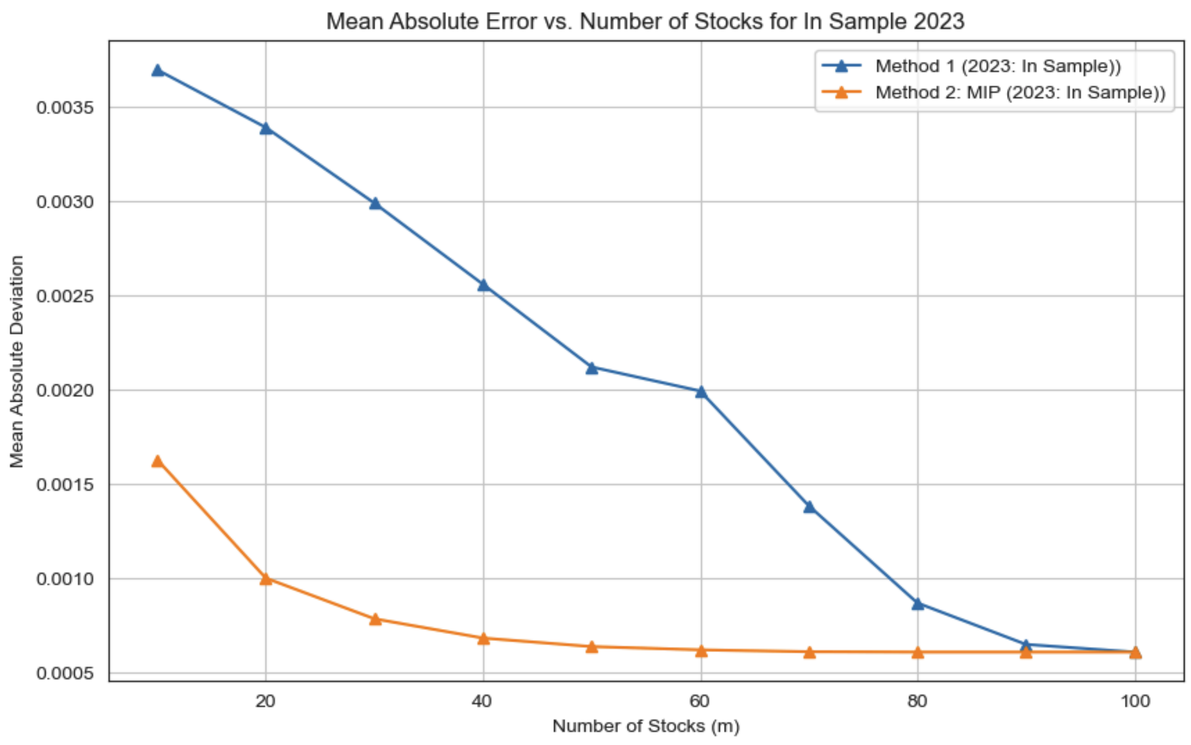
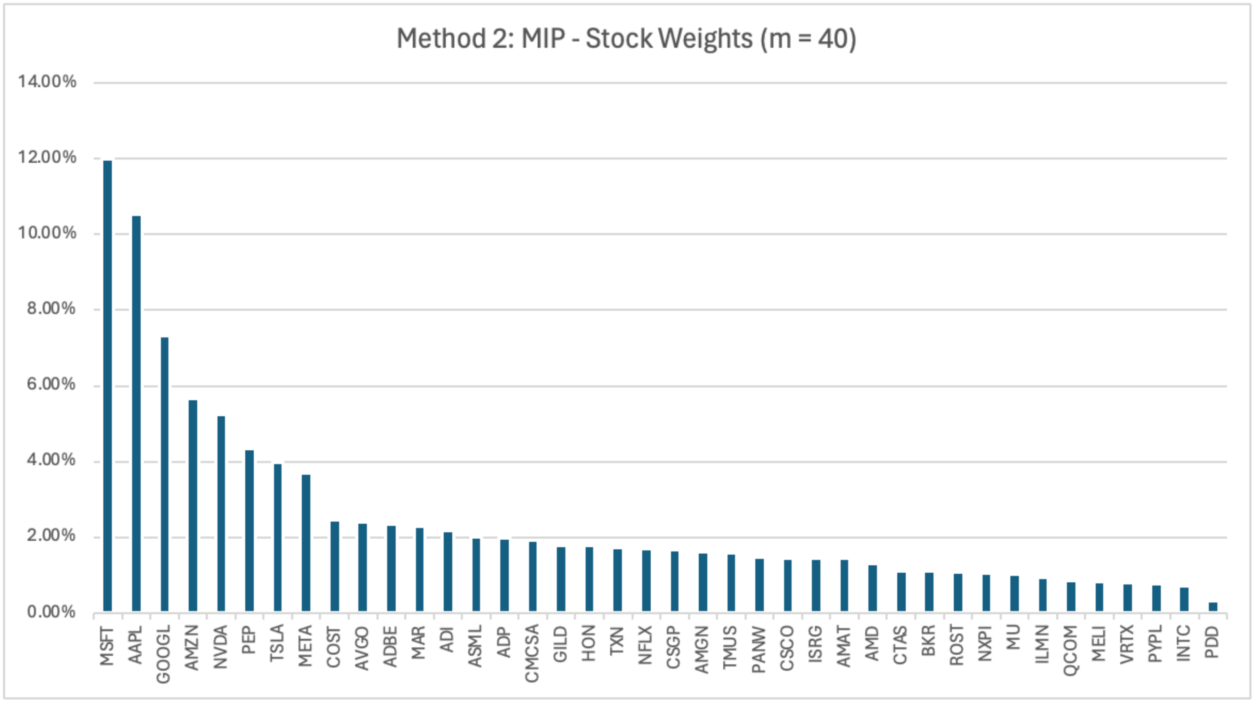


Figure H



References

Cook, Alexis. "Data Leakage." Kaggle, 2019, www.kaggle.com/code/alexisbcook/data-leakage.

OpenAI. *ChatGPT*. OpenAI, 2024.