Abhinandan Shrivastava

Individual Project

Imperial College Business School

MSc Risk Management and Financial Engineering

Word Count: 2667

26th May 2025

Contents

List of Tables

List of Figures

1 Introduction

The aim of this report is to explain how the Markowitz model was used to create portfolios targeting specific returns. As the entire framework was developed and executed in C++, this report will also explain the software structure which was followed while developing the code to tackle this problem. The rest of the report is structured in the following way:

Section 1 describes the general setting of the exercise, and explains the theoretical constructs of the model. Section 2 explains the entire software structure which has been developed to execute the entire framework. Section 3 evaluates the results of the model and discusses the conclusions. The Appendix explains how to use the files which have been submitted along with this report.

1.1 Data

The dataset used in this exercise contains the daily returns of 83 out of the 100 companies listed in the Financial Times Stock Exchange (FTSE). Each company has a time series of 700 daily returns, resulting in a data matrix of size 700×83, where rows represent time points and columns represent individual companies.

1.2 Markowitz Model

The Markowitz model is a very common model used by the industry to quantify the trade-off between risk and returns. This model converted a portfolio construction problem to an optimization problem. For a given return target, you can construct a portfolio which achieves this target while undertaking minimum risk. This is solved in the following way:

1. A set of investable assets (all the 83 companies in our case), have their own mean returns denoted by the vector $\boldsymbol{\mu} \in \mathbb{R}^n$, and have a covariance matrix $\boldsymbol{\Sigma} \in \mathbb{R}^{n \times n}$ which captures the way each asset moves with each other.
2. Given this, we can assign weights to each of these companies using a weight vector $\boldsymbol{w} \in \mathbb{R}^n$ to create a portfolio such that the variance of the constructed portfolio, $\boldsymbol{w}^\top \boldsymbol{\Sigma} \boldsymbol{w}$, is minimized.
3. This minimization problem is subject to constraints. In our case, we have two constraints:
   - The portfolio must target a return $\hat{r}_{\text{target}}$, such that $\boldsymbol{w}^\top \boldsymbol{\mu} = \hat{r}_{\text{target}}$.
   - The weights must sum to one: $\boldsymbol{w}^\top \mathbf{1} = 1$.

   Short-selling is allowed in our case, so the weights $\boldsymbol{w}$ can take negative values.
4. This converts the optimization problem into a linear one, and solving this system gives us the optimal weights $\boldsymbol{w}^*$ that we assign to each company to build the optimal portfolio.

If we remove the short-selling constraint and impose $\boldsymbol{w} \geq \mathbf{0}$, the optimization problem becomes quadratic in nature due to inequality constraints. But for the purposes of this report, short-selling is allowed, and the system solved in this report is linear.

1.3 Rolling Window Evaluation

In order to evaluate the performance of the Markowitz Model, a rolling window approach is implemented. For each window, in-sample returns are first sliced and used to calculate the optimal weights using the Markowitz Model. These optimal weights are then applied to the subsequent out-of-sample returns to evaluate how well the model tracks the given target. Given that we have 700 days of data, the rolling logic proceeds as follows:

- In-sample window size: 100 days

- Out-of-sample window size: 12 days
- Rolling step: 12 days

$$\begin{aligned} \text{In-sample window:} \quad & [12k : 12k + 100) \\ \text{Out-of-sample window:} \quad & [12k + 100 : 12k + 112) \\ & k = 0, 1, \ldots, 49 \end{aligned}$$

Thus, the rolling window process looks like:

Table 1: Rolling Window Logic

Loop	**In-sample Window**	**Out-of-sample Window**
1	[0 : 100)	[100 : 112)
2	[12 : 112)	[112 : 124)
3	[24 : 124)	[124 : 136)
⋮	⋮	⋮
50	[588 : 688)	[688 : 700)

For each in-sample window, the Markowitz Model is used to solve for the optimal weight vector for a given target return. These weights are then applied to the 12-day out-of-sample period to compute the portfolio return, which reflects the model's true out-of-sample performance.

2 Software Structure

The entire framework consists of the following files:

1. `main.cpp`
2. `csv.h`, `csv.cpp`
3. `read_data.h`, `read_data.cpp`
4. `tools.h`, `tools.cpp`
5. `solver.h`, `solver.cpp`
6. `backtest.h`, `backtest.cpp`

Each file (and its associated header, if any) serves a specific purpose. The structure has been developed to follow the logical flow of the model being implemented in this exercise. Object-Oriented Programming (OOP) has been used to make the entire framework modular and robust.

The `main.cpp` file defines a dynamic array called `returnMatrix`. For the given dimensions of the underlying data, it uses the `new` functionality to dynamically allocate memory to hold this dataset at runtime.

The underlying dataset is in a `.csv` format, which is read and stored using READ_DATA and CSV in the matrix created above.

Next, a vector of targeted returns is initialized using the CLASS:BACKTEST and is used to solve the Markowitz Model for each target return in this vector. Each target return corresponds to a specific portfolio, and each portfolio has 50 loops worth of out-of-sample (OOS) data. All of this

information is retained by the developed framework, and the results are stored in another `.csv` file.

The flowchart shown in Figure 1 explains how the functions and classes are interlinked with each other, and a detailed description of the workings of each class is provided in the following sections.

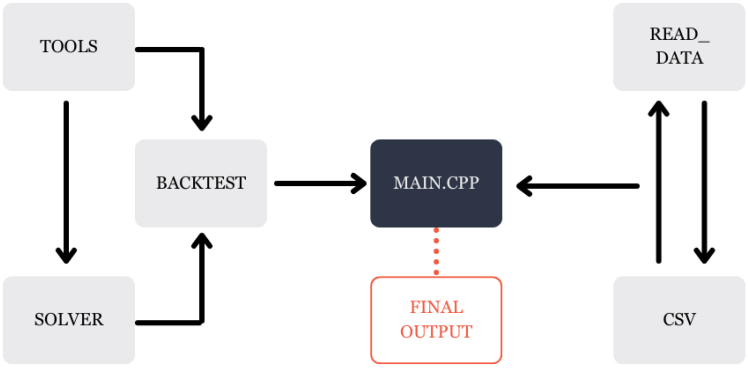

Figure 1: Workflow of the entire framework

2.1 Read_data & Class: Csv

Csv is the first class of the software structure. This is the class which was provided along with the coursework, to help read and parse data which is stored in `.csv` files. This class has not been changed at all in the framework developed in this coursework.

READ_DATA is not a class, it is just a file which stores functions which are utilized for actually reading and writing data from `.csv` files. The first two functions, namely `read_data: string_to_double` and `read_data:readData` were also provided with the coursework. The last function, `read_data: writetoCSV` is a new function, which has been added to write the final results to another `.csv` file. This function takes the 3-D vector of final results, the vector of target returns from Class:Backtest, and for a given `filename`,writes the output in a `filename.csv` file.

2.2 Class: Tools

As the Markowitz Model involves calculating the parameters Mean (μ) and Covariance (Σ), and the Conjugate Gradient Method (CGM) utilizes vector and matrix multiplications, Class:Tools houses all these functions which are related to either statistical properties or mathematical functions.

However, it is important to note that only `Tools::matMultVec` and `Tools::dotProduct` are static functions in this class. This means that an object of Class:Tools need not be created if (Matrix $\times$ Vector) or (Vector $\cdot$ Vector) operations need to be performed anywhere.

The other functions can only be used when an object of Class:Tools is created. To create this object, the following inputs have to be passed:

- `returnMatrix` ($\mathbf{R}$) — the matrix of asset returns
- `numberAssets` (N) — the number of assets in the universe

- `start` (t_s) — the starting index for the rolling window
- `end` (t_e) — the ending index for the rolling window

This is because, further in our classes, the related Mean and Covariance vectors are always associated with a starting and ending index (as they will be called within the rolling window loop), and hence these related features are only calculated for the specific loop within which the object of this class is created.

Figure 2: Functional Flow for `Tools`

Figure 2 shows the hierarchical structure of function calls within the Class:Tools. For a given object:

- `CovMatrix()` calls `Cov()`
- `Cov()` internally calls `Mean()`

This chaining ensures that the covariance matrix is constructed using the correct intermediate mean and covariances calculated from the return data.

2.3 Class: PortfolioSolver

The files `solver.h` and `solver.cpp` contains the code for Class:PortfolioSolver. The member functions `buildQ`, `initialX`, and `buildB` are functions which build all the required inputs for using the Conjugate Gradient Method (CGM) to solve the system of linear equations.

The `PortfolioSolver::buildQ` function actually takes two inputs, the `CovMatrix` and `mean Vector`, and uses them to create the Q matrix of dimensions $[(N+2)\times(N+2)]$, where N: number of assets in the universe. This is done because, apart from the covariance matrix of size $(N \times N)$, Q also contains two extra entries: one for the vector of returns, and one for a vector of ones.

The `PortfolioSolver::initialX` creates a vector of size $[(N+2)\times 1]$, and initializes all the weights as equal. This is one important feature of this model, as CGM requires a good initializing guess for it to work optimally. Initializing the weights as 0 leads to problems with the matrix Q being positive semi-definite, which can be combated through adding a very small regularizing term ($eg: 1e-4$) to the diagonal entries of Q. For this exercise, the weights are being initialized to have the same value, with the code for regularizing term being commented out. However, if the user wanted to initialize the weigths as 0, the regularizing code can be included in the `buildQ` function, and the system will work as intended.

Similarly, `PortfolioSolver::buildB` creates a vector of size $[(N+2)\times 1]$, with the first N values being 0, and the last two values are the $\hat{r}_{\text{target}}$, and -1.

Finally, the function `PortfolioSolver::solvebyCGM` takes all these matrices and vectors as input, and using the CGM, solves for the optimal portfolio weights. The optimal weights are returned by this function. Whenever required, the `static` functions `Tools:matMultVec` and `Tools::dotProduct` are called. The method exactly follows the algorithm highlighted in the problem set. Figure 3 can help illustrate the workflow better:

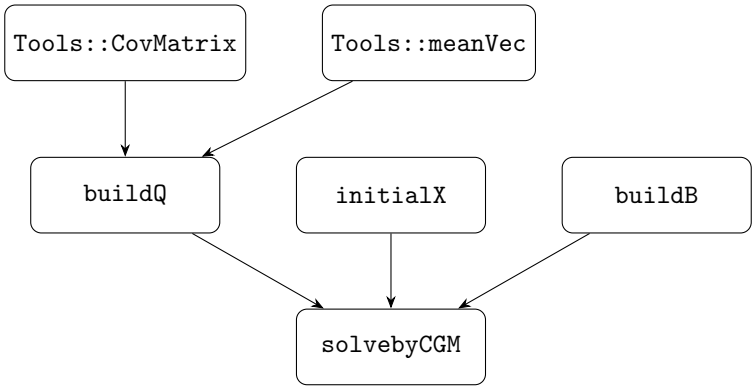

Figure 3: Functional Flow for `PortfolioSolver`

2.4 Class: Backtest

This is the final class of the framework, where all components are integrated and the final output is generated.

Class:Backtest includes a static function named `targetVector`, which can be called without instantiating an object. This function takes as input two endpoints representing the range of target returns, along with an `increment` value. It returns a vector containing all target returns lying between the endpoints, spaced by the specified increment.

In our case, the start and end points are 0.0% and 10.0% respectively, and with increment=0.5%, the `targetVector` houses 20 target returns.

To create an object of the `Backtest` class, the following inputs are required:

- `returnMatrix` ($\mathbf{R}$) — the matrix of asset returns
- `numberAssets` (N) — the number of assets in the universe
- `numberReturns` (T) — the total number of available daily returns
- `targetVector` ($\hat{R}_{\text{target}}$) — the vector of target returns
- `is_window` — the in-sample (training) window size
- `os_window` — the out-of-sample (testing) window size

These input variables must be supplied when instantiating a `Backtest` object. This is also the only class related to the model which is directly used in the `main.cpp` file by the user.

The `Backtest::rollingWindow` function is where the actual model is executed. It utilizes all the previously developed classes and is arguably the most important function in the entire framework. This function takes a single target return, denoted by $\hat{R}_{\text{target}}$ as input and contains a main for-loop that implements the rolling window logic discussed earlier.

For each iteration within this loop, `Backtest::rollingWindow` performs the following steps:

1. Initializes an object from the `Tools` class using the in-sample window. This is used to compute the sample mean vector $\bar{\boldsymbol{\mu}}_{\text{IS}}$ and the sample covariance matrix $\bar{\boldsymbol{\Sigma}}_{\text{IS}}$, which are required by the `PortfolioSolver` class.
2. Using $\bar{\boldsymbol{\mu}}_{\text{IS}}$, $\bar{\boldsymbol{\Sigma}}_{\text{IS}}$, and the target return $\hat{r}_{\text{target}}$, initializes an object of the `PortfolioSolver` class to compute the optimal portfolio weights $\boldsymbol{w}^*$ for the current in-sample period.

3. Creates another object from the `Tools` class, now using the out-of-sample (OOS) window. This yields the OOS mean vector $\bar{\boldsymbol{\mu}}_{\text{OOS}}$ and covariance matrix $\bar{\boldsymbol{\Sigma}}_{\text{OOS}}$.

4. Using the optimal weights $\boldsymbol{w}^*$ from Step 2, calculates the true OOS portfolio mean return:

$$\mu_{\text{portfolio}}^{\text{OOS}} = \boldsymbol{w}^{*T} \bar{\boldsymbol{\mu}}_{\text{OOS}}$$

and portfolio variance:

$$\sigma_{\text{portfolio}}^{2,\,\text{OOS}} = \boldsymbol{w}^{*T} \bar{\boldsymbol{\Sigma}}_{\text{OOS}} \boldsymbol{w}^*$$

After completing all iterations of the rolling window, the function appends the out-of-sample statistics from each loop and returns a matrix that contains the OOS performance metrics for the given $\hat{r}_{\text{target}}$.

The `Backtest::backtestResults` function is relatively straightforward. Given a vector of target returns, $\hat{\boldsymbol{R}}_{\text{target}}$, this function iterates over each value $\hat{r}_{\text{target}} \in \hat{\boldsymbol{R}}_{\text{target}}$, and for each one, it calls the `Backtest::rollingWindow` function.

The output is a 3-dimensional vector, where for each target return $\hat{r}_{\text{target}}$, the corresponding out-of-sample (OOS) portfolio mean and variance values obtained from each rolling window iteration are stored. This is the actual result of the entire model, this is the function which is called by the user, and the output from this function is actually written into the output `assignment_results.csv` file. Figure 4 can help illustrate the workflow better:

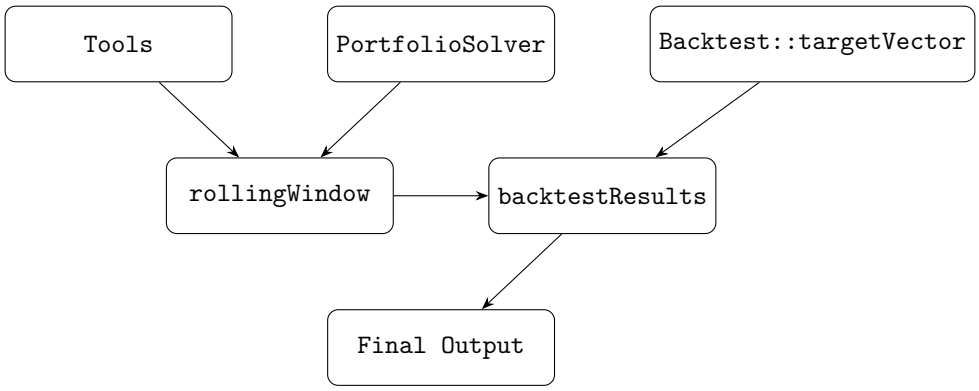

Figure 4: Functional Flow for `Backtest`

3 Model Evaluation

As explained in the previous section, the final output is a 3-D vector. Specifically, each $\hat{r}_{\text{target}} \in \hat{\boldsymbol{R}}_{\text{target}}$ has OOS results of dimension 50×2, where the rows represent the time loops, and the columns represent the OOS portfolio Mean $\mu_{\text{portfolio}}^{\text{OOS}}$ and OOS portfolio variance $\sigma_{\text{portfolio}}^{2,\,\text{OOS}}$ respectively.

As each portfolio is built to target a specific return, model evaluation will consist of quantifying how closely did the portfolios tracked their targets, and how much variance did they take on while doing so. For this purpose, we will be using the Mean Absolute Error (MAE) statistic, which is given by the following formula:

$$\text{MAE} = \frac{1}{T} \sum_{t=1}^{T} \left| r_t^{\text{actual}} - r_t^{\text{target}} \right| \tag{1}$$

Here, r_t^{actual} denotes the out-of-sample return realized at time t, while r_t^{target} denotes the target return that the portfolio aimed to achieve.

Apart from that, each $\hat{r}_{\text{target}}$ has 50 means. To better evaluate this metric, a boxplot is used to plot the distribution of these means for each target. This will help illustrate how exactly the means are distributed, as we go from 0.5% to 10.0% as target returns.

As observable from Table 2, the out-of-sample (OOS) performance of the Markowitz model is not particularly strong. As the target returns increase, the mean OOS return also increases, although it remains well below the expected targets. At the same time, the average variance also increases linearly. This may occur due to the following reasons:

1. As target returns increase, the optimizer attempts to achieve these higher returns by taking on greater variance.
2. A higher variance causes the actual returns to be spread over a wider range, but they are spread around 0.0%, not around the target mean.
3. Additionally, a larger number of returns fall on the right-hand side of 0.0%. Consequently, when we take an average over these values, the mean OOS return keeps increasing steadily.

Table 2: Performance Metrics Across Target Returns (%)

Target Return	**Mean OOS Return**	**MAE OOS Mean**	**Avg OOS Variance**
0.5	0.252152	0.433508	0.044070
1.0	0.353944	0.774304	0.091190
1.5	0.429962	1.137814	0.146902
2.0	0.543018	1.542878	0.239498
2.5	0.591190	1.982362	0.364316
3.0	0.596474	2.485954	0.524300
3.5	0.721756	2.872344	0.711586
4.0	0.847674	3.352790	0.934562
4.5	0.927530	3.779814	1.188868
5.0	0.988318	4.268754	1.456446
5.5	1.019632	4.761720	1.826376
6.0	1.062820	5.242800	2.224440
6.5	1.209806	5.648650	2.709134
7.0	1.170964	6.173924	3.293936
7.5	1.272500	6.659664	4.035834
8.0	1.310244	7.086696	4.888886
8.5	1.359502	7.530902	5.631376
9.0	1.504658	7.988654	6.355346
9.5	1.481080	8.535508	7.411674
10.0	1.552870	9.155690	8.320676

These observations can be better illustrated by Figure 5:

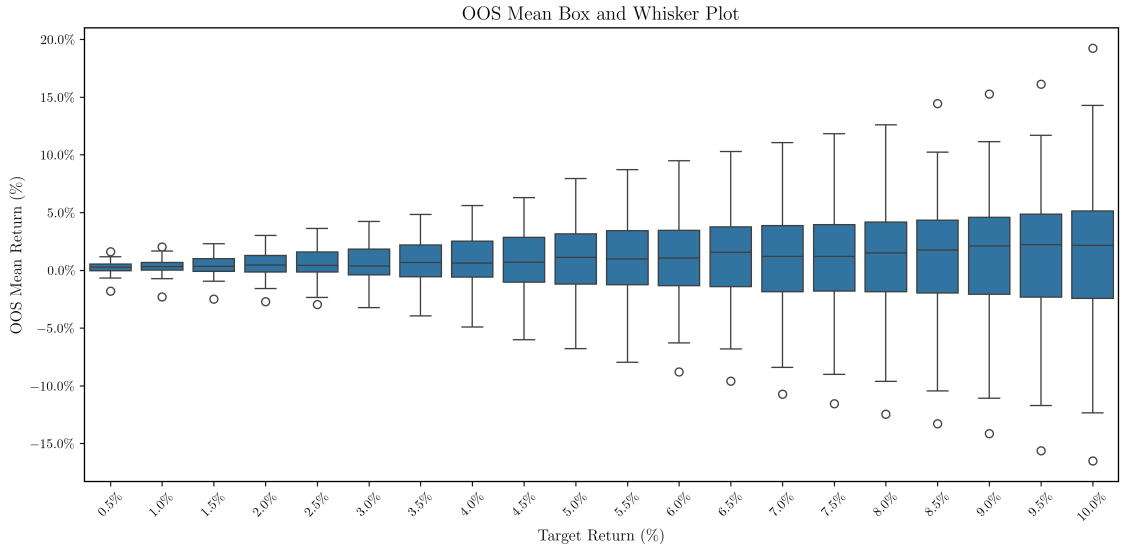

Figure 5: Plotting Boxplot of OOS Mean Returns

As observable by the increasing length of the Inter-Quartile Range (IQR) and the whiskers, the mean OOS returns are distributed with more variance as the target returns increase. However, the increase in the IQR is not as drastic as the minimum ($Q_1 - 1.5 \times \text{IQR}$) and maximum ($Q_3 + 1.5 \times \text{IQR}$) values, indicated by the fences of the whiskers. The circles for each target return represent outlier values, and all these observations suggest that the OOS returns generated within the loops have been very volatile, and such outliers are causing the average mean to stay close to 0.0%.

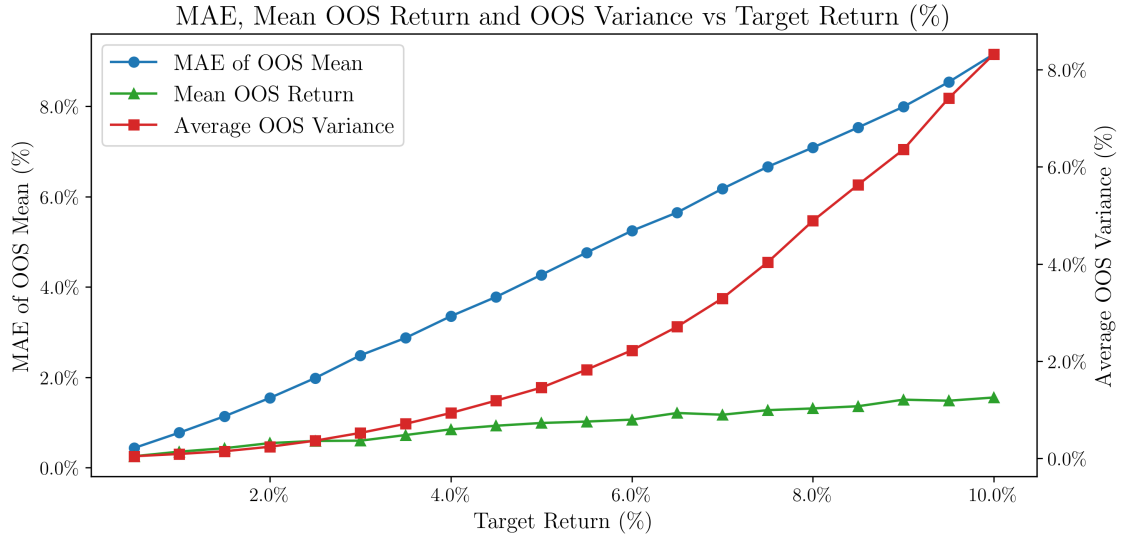

Figure 6: Plotting Performance Statistics

Figure 6 plots the performance statistics, and the strong linear trend in the average OOS variance and the MAE of the mean returns is clearly visible here.

We can also plot a Mean-Variance frontier, but from our OOS results instead, to study the relationship between the portfolio returns and variance.

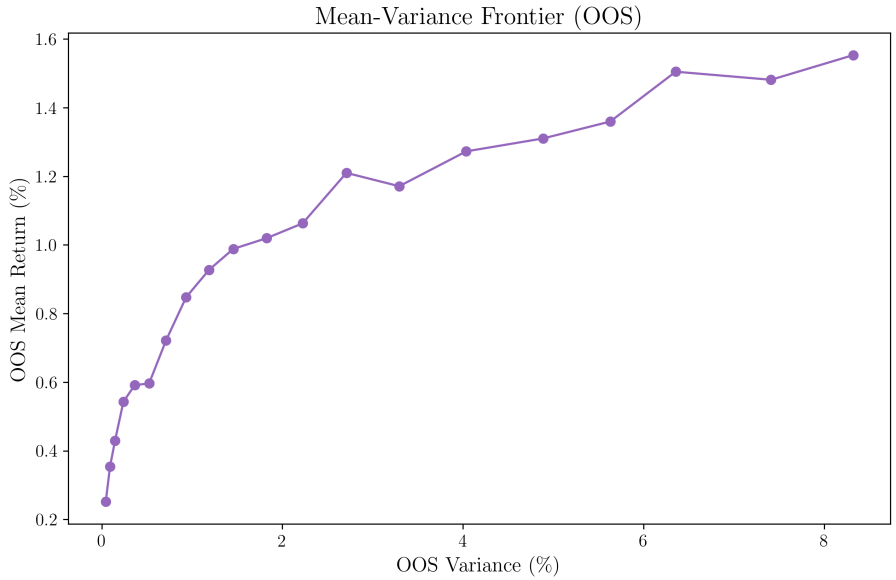

Figure 7: Plotting the OOS Efficient Frontier

As observable from Figure 7, the frontier closely resembles the Mean-Variance frontier that we are used to seeing. It does not perfectly resemble the Efficient Frontier, that is because the Efficient Frontier is plotted for the in-sample portfolios, not out-of-sample portfolios. Regardless, we can see the risk-return tradeoff in action even in the OOS plot, despite the returns being nowhere near the targets.

3.1 Conclusions

In the context of all the observations that have been made, we can draw upon the following points:

1. Traditionally, using the Markowitz Model and plotting the returns against the standard deviation results in a monotonic concave function, which implies that Markowitz assumes:
   - There is a risk-return tradeoff; in order to earn high returns higher levels of risks need to be taken on.
   - This tradeoff is marginally decreasing; taking on more unit risk gives back lower and lower unit returns.
2. In our case, we have been able to observe these theoretical constructs.
3. Additionally, the Markowitz model ends up over-fitting on the sample data (as the in-sample portfolio always has the target returns), and this leads to deteriorating out-of-sample performance.
4. For higher target returns, the optimizer is stretching itself to achieve the targets, and this leads to the model becoming more sensitive to outliers and prone to estimation errors.

In conclusion, while the Markowitz model provides a foundational framework in portfolio theory, it falls short in practice when constructing portfolios that aim to achieve specific target returns. Nonetheless, implementing this framework entirely in C++ serves as an excellent learning experience, offering practical exposure to C++ programming and the use of object-oriented principles to create a modular design. With further refinement, the model can be extended and adapted to better align with real-world investment needs.

Appendix

To successfully run the files which have been submitted along with this report, the user just needs to run the following commands on their terminal:

1. `g++ -c csv.cpp read_data.cpp main.cpp`
2. `g++ -c tools.cpp solver.cpp backtest.cpp`
3. `g++ -o ps3 csv.o read_data.o main.o tools.o solver.o backtest.o`
4. `./ps3`

The same commands have been placed in the `main.cpp` file as well, and they have been commented out.

After the final output was written into `assignment_result.csv`, Python was used to calculate the performance statistics and generate the plots displayed in this report. The `.zip` file also contains a file named `OOS_Eval.ipynb`, which is the Jupyter Notebook used to perform these actions.