A Markowitz Portfolio Optimiser

Computational Finance with C++

CID: 01805027

10 June 2020

1 Introduction

Since their inception, economists, mathematicians, physicists and computer scientists the like have sought to 'beat' the markets using developments in their respective fields.

One such forum in which attempts have been made is the problem of portfolio choice. It is widely accepted that one of the goals of investing in financial markets is to make as much excess return as possible (above the standard market return; often characterised by the risk free rate) for a given level of risk taken on (idiosyncratic and non-idiosyncratic).

This is characterised well by the *Efficient Frontier* as seen in Figure (1).

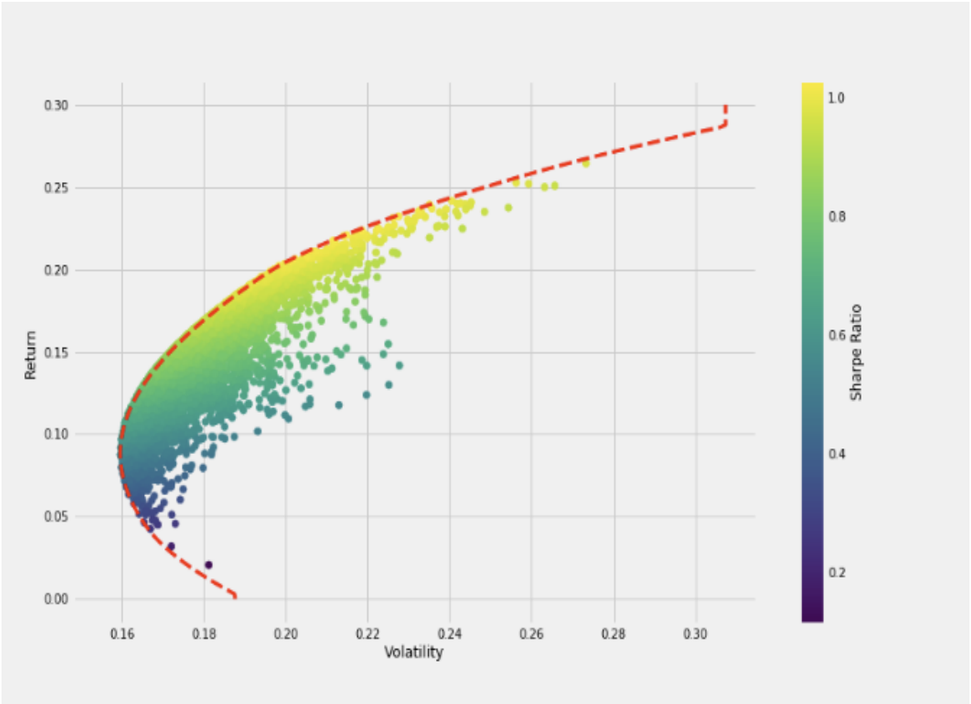

Figure 1: Expected return against volatility. The red dashed line characterises the efficient frontier itself, on which the optimal portfolios lie. [1] Note here the term *volatility* is used to represent the variance of the portfolios, not the standard deviation as is traditional.

Optimal portfolios lie on the dashed red line, named the *Efficient Frontier* as it characterises the portfolios that make as much excess return as possible for a given risk appetite, denoted by its position along the horizontal axis. The points in the centre of the concave function are examples of suboptimal portfolios. These portfolios carry more risk than the theory requires for their level of expected excess return, or similarly do not return enough above the market rate, for the amount of risk they take on.

No portfolios exist outside of the frontier since this scope corresponds to portfolios that achieve a theoretically impossibly large return for a given risk appetite, or conversely and impossibly small level of risk for a quoted level of remuneration.

It must be noted that mean-variance efficiency is a corner-stone of modern portfolio choice theory, even appearing as an assumption to the renowned Capital Asset Pricing Model [2] (CAPM); it is assumed that 'all market participants are mean-variance efficient optimisers'. Though the subtle but important point must be made that the CAPM states that all market participants are rewarded for the amount of *non-diversifiable, systematic*

risk taken on, and says there shall be no remuneration for *diversifiable, idiosyncratic risk* which are generally included in the measures of portfolio volatility used for the efficient frontier plot above.

As a result, this project seeks to solve the portfolio choice problem by an implementation of Markowitz Portfolio Theory [3] in C++ using numerical methods and a backtesting framework to test the resulting portfolio.

1.1 Theory

The formal mathematical derivation below largely following notes provided by Parpas, P [4] are included for the context of the problem.

We define a target return, $\overline{r_p}$, for the portfolio and we wish for the portfolio to return this level of compensation for as little risk (portfolio variance) as possible.

Therefore a system of Lagrange Multipliers are used to define the requirements on the portfolio. Namely, to:

1. **Minimise portfolio variance,** σ_p^2 - Equation (1)
2. **Ensure the expected portfolio return is equal to the target return,** $\overline{r_p}$ - Equation (2)
3. **Constrain the allocation of funds** - to ensure that the portfolio does not see an over or under-allocation of the funds available to it; it must have invested all resources available to it at any one time - Equation (3)

Note that there have been no constraints placed on the individual values of the portfolio weight of asset i, w_i. In other similar studies constraints such as ensuring the positivity of w_i are employed to forbid the system from short selling a security. This constraint has not been imposed here; the model is freely able to short sell a security should it see fit.

In Equation (1), we define the variance of the Markowitz portfolio, σ_p^2, with a preceding factor of $\frac{1}{2}$ for convenience. σ_{ij} is the covariance between stocks i and j.

DO WE WANNA ADD THE DERIVATION?

$$\frac{1}{2}\sigma_p^2 = \frac{1}{2}\sum_{i,j=1}^{n} w_i \sigma_{ij} w_j \tag{1}$$

The variance of the portfolio is minimised via the Lagrange method while ensuring the following two constraints are also upheld:

$$\sum_{i=1}^{n} w_i \overline{r_i} - \overline{r_p} = 0 \tag{2}$$

Above, we mandate that the portfolio return is equal to the value target return $\overline{r_p}$. By setting $\overline{r_p}$, we have effectively chosen the portfolio return we desire and thus the weights the optimisation method yields.

Below, is simply the constraint that the sum of the portfolio weights must be equal to unity. This ensures that the portfolio does not over or under-allocate the funds available to it.

$$\sum_{i=1}^{n} w_i - 1 = 0 \tag{3}$$

$$L(w, \lambda, \mu) = \frac{1}{2}\sum_{i,j=1}^{n} w_i \sigma_{ij} w_j - \lambda\left(\sum_{i=1}^{n} w_i \overline{r_i} - \overline{r_p})\right) - \mu\left(\sum_{i=1}^{n} w_i - 1\right) \tag{4}$$

Finally, Equation(4) shows the Lagrangian we seek to minimise, with λ and μ the lagrange multipliers for points

For convenience when scaling to a large number of assets and to code in the possibility of introducing further constraints, we introduce the following matrix notation:

- $\boldsymbol{w} = (w_1, ..., w_n)' \in \Re^n$ - a column vector of n weights for n assets
- $\overline{\boldsymbol{r}} = (\overline{r_1}, ..., \overline{r_n})' \in \Re^n$ - a column vector of n expected asset returns
- $\boldsymbol{e} = (1, ..., 1)' \in \Re^n$ - the unit column vector of length n
- $\boldsymbol{0} = (0, ..., 0)' \in \Re^n$ - the zero column vector of length n

The Lagrangian in matrix form is shown below in Equation (5) where $\Sigma \in \Re^{n \times n}$ is the covariance matrix of the assets, explicitly; $\Sigma_{ij} = \sigma_{ij}$ as defined above.

$$L(\mathbf{w}, \lambda, \mu) = \frac{1}{2}\mathbf{w}'\Sigma\mathbf{w} - \lambda\left(\mathbf{w}'\bar{\mathbf{r}} - \overline{r_p}\right) - \mu\left(\mathbf{w}'\mathbf{e} - 1\right) \tag{5}$$

Now, in order to minimise the Lagrangian it must be differentiated with respect to the weights, $\mathbf{w}$. Allowing us to find the rate of change of the system's Lagrangian with respect to the portfolio weights and setting the resulting equation to zero will yeild the turning point of the function in parameter space.

$$\frac{\mathrm{d}L(\mathbf{w}, \lambda, \mu)}{\mathrm{d}\mathbf{w}'} = \Sigma\mathbf{w} - \lambda\bar{\mathbf{r}} - \mu\mathbf{e} = 0 \tag{6}$$

It may be seen by inspection that the second derivative of the Lagrangian with respect to $\mathbf{w}$ is indeed positive for all values of $\mathbf{w}$ meaning that this is indeed a minimum and that the function itself is concave. Note that this is aligned with our expectation that the efficient frontier is concave, as in Figure (1).

With Equations (2) and (3) also required for optimality, we may write the system of $n + 2$ simultaneous equations as a single matrix equation written in the form $Ax = b$, below in Equation (7). A is an $n + 2$ square matrix and the two column vectors both have dimensions $(n + 2) \times 1$ respectively. We seek to solve this numerically via the Quadratic Conjugate Method to find the vector of weights w that yield the desired portfolio return, r_p, the solution to Equation (8).

$$\begin{bmatrix} \Sigma & -\bar{\mathbf{r}} & -\mathbf{e} \\ -\bar{\mathbf{r}}' & 0 & 0 \\ -\mathbf{e}' & 0 & 0 \end{bmatrix} \begin{bmatrix} \mathbf{w} \\ \lambda \\ \mu \end{bmatrix} = \begin{bmatrix} \mathbf{0} \\ -\overline{r_p} \\ -1 \end{bmatrix} \tag{7}$$

$$\begin{bmatrix} \mathbf{w} \\ \lambda \\ \mu \end{bmatrix} = \begin{bmatrix} \Sigma & -\bar{\mathbf{r}} & -\mathbf{e} \\ -\bar{\mathbf{r}}' & 0 & 0 \\ -\mathbf{e}' & 0 & 0 \end{bmatrix}^{-1} \begin{bmatrix} \mathbf{0} \\ -\overline{r_p} \\ -1 \end{bmatrix} \tag{8}$$

2 Implementation

We test the portfolio composed of assets weighted by the weights forming the solution of Equation (8). This is done for 83 of the FTSE 100 assets over a period of 700 days in total. Each in sample window is 100 days long, used to calibrate the weights to generate a portfolio with the lowest viable risk exposure for a predetermined target portfolio return: $\overline{r_p}$.

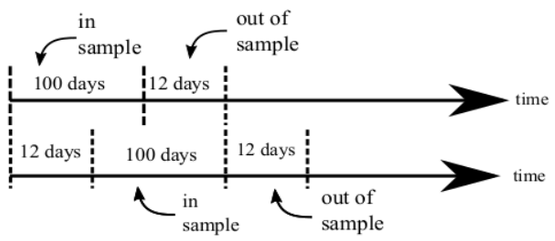

Figure 2: Organisation of the rolling balancing and backtesting windows. [4]

More precisely, for day t_i with i in the following ranges, the backtest is in the corresponding balancing or evaluation phase:

- **In sample (balance) window 1** - $t_i \,\forall\, i \in [0, 99]$
- **Out of sample (evaluation) window 1** - $t_i \,\forall\, i \in [100, 111]$
- **In sample(balance) window 2** - $t_i \,\forall\, i \in [12, 111]$
- **Out of sample (evaluation) window 2** - $t_i \,\forall\, i \in [112, 121]$

This continues until the end of the 700 day dataset is reached. In total, the application balances 50 times, generates 50 new sets of weights, and evaluates these 50 times **for each target return**.

How this was achieved efficiently is outlined in the following section.

2.1 Code

With a distinct focus on code readability, maintainability and efficiency, the backtesting frameworks, Quadratic Conjugate Method and other necessary tools to backtest the Markowitz optimised portfolio were written in C++ (which may be seen explicitly in the code subsection of the Appendix, or in the public repository.).

The project code was split into five distinct project directories for the easy of maintenance and cleanliness. The directories and their contents are outlined briefly for context in the following sections.

2.1.1 Utility

The Utility contains the classes and structs *VectorUtil, Matrix, RunConfig* and *Results*. The first two housed small building-block-esque functions upon which the entirety of the application is based. As such, it was vital that these methods used to perform arithmetic operations on the rank 1 and 2 tensors were well tested and efficient. Creating these classes first greatly increased the speed and accuracy of development of the more complex classes that followed. It was imperative that the arithmetic functions in *VectorUtil* were as generic as possible. As a result, generic types were used throughout, though a condition was imposed in the typename signature that ensured the types being passed were numeric, as can be seen in the snippet below:

```
// VectorUtil.h

template<typename T,
   typename = typename enable_if< is_arithmetic<T>::value, T> ::type>
vector<T> vectorLinearCombination(T aCoeff, vector<T> *a, T bCoeff, vector<T> *b)
{
   int aSize = a->size();
   int bSize = b->size();
   if (aSize != bSize) {
      cout << "Vectors aren't the same size and so cannot be subtracted/added
          from/to one another." << endl;
      cout << "\t Sizes: " << aSize << " and " << bSize;
      exit(12);
   }
   vector<double> result = vector<double>(aSize);
   for (int i = 0; i < aSize; i++) {
      result[i] = aCoeff * a->at(i) + bCoeff* b->at(i);
}

return result;
```

Also seen in the above method is the helpful error message when it receives data incompatible with the operation, and an easily searchable error code for ease of debugging.

RunConfig and *Results* are both C++ structs used to group pieces of information in a convenient manor. The former contained parameters that were pertinent to a particular run of the backtest. It was useful to run the application on small and medium sized subsets of the data for debugging purposes and so with each of these came a set of parameters (in/out of sample window lengths, for example) that were specific to the dataset in question. It became cleaner and more convenient to group these parameters together into one object (a *RunConfig* struct) and pass this to *Portfolio::backtest()*. The same philosophy was behind the use of the *Results* struct as a return type from the aforementioned portfolio method as a way to group multiple result outputs together, instead of passing each object by reference or pointer individually.

I found this line of reasoning to be especially pertinent with method signatures since their readability contributes massively to the readability of the application as a whole.

2.1.2 Repository

Repository houses the classes and methods used to load data in from the comma separated value files. The functionality here was mostly provided with few other editions required.

2.1.3 Estimator

This directory holds the *ParamaterEstimator* class and its header, a prime example for the use case of a singleton object design pattern. The class holds public static methods that calculate various parameters such as the means, covariances and standard deviations of the aforementioned vector and self-made Matrix objects passed to it. It maintains no state and

has no attributes; testament to the functional approach to programming commonly used when encoding mathematical functions.

2.1.4 Optimiser

The *PortfolioOptimiser* class and header are held in this directory and are where the Quadratic Conjugate Method is implemented. It was necessary for this class to maintain a state, having attributes on it that would be required across multiple of its methods. These attributes consisted of constant parameters from the *RunConfig* of the current simulation, such as the initial values for the Lagrange Multipliers λ and μ and the tolerance to which we declare the optimisation method as having converged.

In addition, a getter and setter are employed to set the target return for the optimiser, $\overline{r_p}$ as the same instance of the *PortfolioOptimiser* is used to solve for all different target returns.

2.1.5 Backtest

The Backtest directory is where the *Portfolio* class, header and *main.cpp* were situated. The *main.cpp* acted as the entry point into the application; where the *RunConfigs* were initialised and sent to the aforementioned *Portfolio::backtest()* method.

A matrix containing all returns data required for the simulation (a matrix of 700 rows by 83 columns in its entirety), was spliced with days acting to index the matrix; their values corresponding to the appropriate balancing and evaluating windows as described above.

For eahc window, the balancing window's matrix (in sample) of returns is passed by pointer to *PortfolioOptimiser::calculateWeights()* which in turn calls a series of its own helper functions to return the weights.

Once received *Portfolio::checkWeights()* is called to ensure the sum of the weights is to within a predefined tolerance of unity (tolerance defined in *RunConfig*).

The weights are then passed with the test window's matrix (out of sample) to *Portfolio::evaluate()* to evauluate the return of the portfolio over the out of sample window. The same weights are also passed with the in sample matrix of returns, generating the in sample returns ensuring the target return $\overline{r_p}$ is indeed achieved in sample. This also acts as a useful comparison to the out of sample portfolio returns achieved.

We now move to the discussion of the calculation of the weights themselves as the solution to Equation (8) by the Quadratic Conjugate Method.

2.2 Quadratic Conjugate Method

Recall that the Markowitz Portfolio Choice Problem was reduced to a matrix equation of the form $Ax = b$ and that the Quadratic Conjugate Method solves this for vector x numerically, with x defined as:

$$x = \begin{bmatrix} \mathbf{w} \\ \lambda \\ \mu \end{bmatrix} \tag{9}$$

An initialisation of x_0 is required for the method and so the weights $\mathbf{w}$ are chosen such that the initial portfolio is equally weighted (with a portfolio of $n = 83$ assets this is equivocates to setting all $w_i \approx 0.01205$). Inital values for the Lagrange Multipliers λ and μ are set to 0.5 each though it was discovered that the initialisation of the two multipliers has little to no effect on the subsequent optimisation results.

A value of ϵ is required as our tolerance limit whereby convergence of the algorithm is defined when the length of the error term s_{k+1} squared is less than ϵ. This value is passed through to the private *PortfolioOptimise::conjugateGradientMethod()* from the $RunConfig$ and a suitable value was found to be 10^{-6}.

Though it must be noted that varying ϵ by a factor of 10^2 smaller or larger seemed to have no effect on the rate of convergence. After some further investigation it was found that for the numbers typical to this context, the product $s_k' s_k$ jumps discontinuously from around 10^{-3} to 10^{-10}. More generally, convergence has been proven to occur in fewer than iterations than the number of rows in matrix A [5]. This implies convergence is guaranteed in fewer than $n + 2$ iterations, 85, in this dataset.

Algorithm 1 Quadratic Conjugate Method

Initialization: $k = 0, s_0 \equiv b - Ax_0, p_0 \equiv s_0$;

while $s_k' s_k > \epsilon$ **do**
  $\alpha_k = \frac{s_k' s_k}{p_k' A p_k}$
  $x_{k+1} = x_k + \alpha_k p_k$
  $s_{k+1} = s_k - \alpha_k A p_k$
  if $s_{k+1}' s_{k+1} < \epsilon$ **then**
    exit
  end if
  $\beta_k = \frac{s_{k+1}' s_{k+1}}{s_k' s_k}$
  $p_{k+1} = s_{k+1} - \beta_k p_k$
  $k+ = 1$
end while

All variables apart from the result, x_{k+1} are discarded when the thread leaves the scope of the loop as they are no longer required. Finally, the weights w_i are retrieved from the first n elements of x_{k+1}.

3 Results & Discussion

Since this is an exercise in implementing a numerical optimisation method to financial data, the results of the application are judged by their adherance to the constraints imposed by the theory in section two.

More specifically, the validity application will, in part, be measured by its ability to generate weights that form a portfolio yielding the target return specified.

3.1 Accuracy

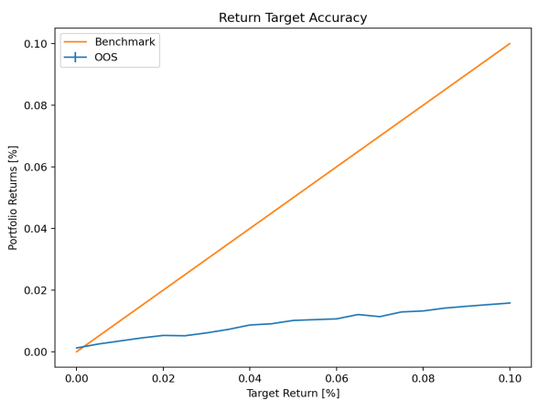

Figure 3: A plot of the efficient frontier as generated by the application.

3.2 Performance

Since C++ was chosen as the language, the application's performance must be mentioned. Running the backtest across the full dataset of 700 days and $n = 83$ assets a total of 20 times, the mean and standard error were calculated yielding a commendable took $14598.6 \pm 10.5\ ms$. EDIT THIS WITH NEW TIMES

BASICALLY TABLE THAT ALREADY GETS PRINTED OUT

TIME TAKEN TO RUN THE SCRIPT

EXPLAIN WHY STANDARD DEVS GETHIGHER AS RETURNS GET HIGHER - BECASUE WE CANT SEE IT IN THE DATA 9SINCE EXCESSIVELY LARGE RETURNS ARE LESS COMMON) -¿ MORE NOISE AND ERATICISM IN WEIGHTS PRODUCTION

METHOD TENDS TO A LIMIT WHERE IT CANT PROCURE MORE RETURN SIMPLY BECAUE PORT RETURN IS WEIGHTS . RETURNS SO PORT RETURN IS LIMITED TO AT BEST, THE SAME RETURN AS THE BEST ASSET SEEN IN THE MARKET. (NO SHORT SELLING) - MAKE UP A SCENARIO WHERE WE SEE IT SHORT SELL A STOCK WITH A NEGATIVE RETURN –¿ METHOD WORKS

4 Conclusions

DECENT WAY TO OPTIMISE GIVEN HOW FAST CONVERGENCE IS
you may like to fix their values at the beginning.
Notice that you should not get (very) different results for different initialization values.
At the end of the day, this is a method that gives you the solution of a linear system!

References

[1] Medium. 2020. Fábio Neves. [ONLINE] Available at: https://towardsdatascience.com/python-markowitz-optimization-b5e1623060f5. [Accessed 8 June 2020].

[2] Black, Fischer., Michael C. Jensen, and Myron Scholes (1972). The Capital Asset Pricing Model: Some Empirical Tests, pp. 79–121 in M. Jensen ed., Studies in the Theory of Capital Markets. New York: Praeger Publishers.

[3] Markowitz, H.M. (March 1952). "Portfolio Selection". The Journal of Finance. 7 (1): 77–91. doi:10.2307/2975974. JSTOR 2975974.

[4] Parpas, P. "Computational Finance with C++, Numerical Methods for Optimization Models" pp. 27-37, Imperial College London

[5] Saad, Yousef (2003). Iterative methods for sparse linear systems (2nd ed.). Philadelphia, Pa.: Society for Industrial and Applied Mathematics. pp. 195. ISBN 978-0-89871-534-7.

5 Appendix

5.1 Additional Figures

Run	Time Taken [ms]
1	14760
2	14519
3	14557
4	14569
5	14551
6	14575
7	14593
8	14601
9	14598
10	14594
11	14588
12	14620
13	14582
14	14602
15	14591
16	14600
17	14609
18	14607
19	14610
20	14646

Table 1: The time taken for full runs of the backtesting application over all 83 assets and 700 days. **Mean ± standard error:** $14598.6 \pm 10.5\ ms$

5.2 Code