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.

maybe capm doesnt need to be mentioned? capm states investors and mean varianceefficient optimisers, and so efficient frontier must've come first

A long-standing theory of the markets is the Capital Asset Pricing Model (CAPM) which states that a market participant may be rewarded with returns above the market rate proportionate to the amount of risk taken on. This gave rise

MENTION CAPM BUTCAPM SAYS YOURE REWARDED PROPORTIONAL TO HOW MUCH NON-DIVERSIFIABLE RISK YOU TAKE ON, oN THE FRONTIER GRAPH WE LOOK AT ALL THE VARIANCE ASSOCIATED WITH THE STOCK (DIVERSIFIABLE AND NOT)

1.1 Theory

As aforementioned, we employ Lagrange Multipliers to constrain the portfolio problem, allowing us to adhere to real-world constraints whilst also seeking to minimise the variance of the portfolio.

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

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

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 r_p. By setting r_p, we may effectively 'choose' the portfolio return we desire and thus the weights the optimisation method yields.

Below, is simple the constraint that the sum of the portfolio weights must be equal to unity. This ensures that we do not over or under-allocate the funds available to us.

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

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

Finally, Equation(4) shows the Lagrangian we seek to minimise.

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 vector of n weights for n assets
- $\overline{\boldsymbol{r}} = (\overline{r_1}, ..., \overline{r_n}) \in \Re^n$ - a vector of n expected asset returns
- $\boldsymbol{1} = (1, ..., 1) \in \Re^n$ - the unit vector n
- $\boldsymbol{0} = (0, ..., 0) \in \Re^n$ - the zero 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 themselves and setting the resulting equation to zero will give us the turning point of the function.

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

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.

With Equations (2) and (3) also required for optimality, we may write the system of $n+2$ 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. Later 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

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 code subsection of the Appendix).

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 contained the classes and structs *VectorUtil, Matrix, RunConfig* and *Results*. The first two contained 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 vectors and matrices were well tested and efficient.

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 Data Repository

2.1.3 Parameter Estimation

2.1.4 Portfolio

2.1.5 Backtest

- **Backtest**
- **Parameter Estimation**
- **Portfolio**
- **Data Repository**
- **Utility**

2.2 Quadratic Conjugate Method

3 Results & Discussion

4 Conclusions

References

[1] Watson, P. (2005). Ideas: A History of Thought and Invention from Fire to Freud. New York: HarperCollins Publishers.

5 Appendix

subsectionCode