Constrained quadratic optimisation

[Nematrian website page: ConstrainedQuadraticOptimisation, © Nematrian 2015]

Abstract

These pages set out the mathematics behind constrained quadratic optimisation, as implemented using a variant of the Simplex algorithm, and this sort of optimisation relates to traditional Markowitz (i.e. mean-variance) portfolio optimisation.

Contents

1. The canonical problem
2. Solving the canonical problem
3. Setting up the 'super'-problem
4. Updating the tableau
5. Finishing the algorithm
6. Portfolio optimisation

Tools
References

1. The canonical problem

Constrained quadratic optimisation involves finding the value of a vector $x = (x_1, \ldots x_n)^T$ that minimises a given penalty function $L(x)$ (or maximises it, the two are interchangeable by replacing L with – L) subject to some (normally linear) constraints, where:

$$L(x) = B + Cx + x^T Dx$$

The constraints, in canonical form, are normally of two types. There are n lower limit constraints of the form $x \geq 0$ (by which we mean that each $x_i \geq 0$ for $i = 1, \ldots n$) and there are m further constraints of the form $Ax \leq P$ (by which we mean that each $\sum_j A_{ij} x_j \leq P_i$ for $i = 1, \ldots m$).

In these definitions B is a scalar, C is a vector (or $1 \times n$ matrix) and D is an $n \times n$ symmetric matrix, which is assumed to be non-positive definite if the problem is a minimisation, and non-negative definite if the problem is a maximisation. A non-negative definite (symmetric) matrix is one whose eigenvalues are all at least zero.

The value of B is irrelevant to the optimal value of x so without loss of generality can be taken as zero.

Problems with lower limits on each x_i that are non-zero, say $x \geq q$ can be restated into the above form by a change of variables to $y = x - q \geq 0$ resulting in a new penalty function:

$$\hat{L}(y) = B + C(y - q) + (y - q)^T D(y - q)$$
$$\hat{L}(y) = \hat{B} + \hat{C}y + y^T \hat{D} y$$

Here $\hat{L}(y) = \hat{B} + \hat{C}y + y^T \hat{D} y$ where $\hat{B} = B + Cq$, $\hat{C} = C - 2q^T D$ and $\hat{D} = D$. We therefore merely need to alter C appropriately, and unwind the change of variables at the end of the optimisation process.

Problems that involve 'greater than' or 'equals' type constraints, e.g. $Ax \geq P$ or $Ax = P$ as well as (or instead of) 'less than' type constraints can be converted into the above canonical form by:

(a) converting each 'equality' constraint into two equivalent constraints, one being the corresponding 'greater than' constraint and one being the corresponding 'less than' constraint, altering m accordingly (since if $Ax = P$ then $Ax \geq P$ and $Ax \leq P$, and then

(b) inverting each 'greater than' constraint present into a 'less than' constraint by noting that if $\sum_j A_{ij} x_j \geq P_i$ then $-\sum_j A_{ij} x_j \leq -P_i$.

2. Solving the canonical problem

We can solve the canonical problem using a variant of the Simplex algorithm, as follows:

(a) We use the method of *Lagrange multipliers*, where $\lambda = (\lambda_1, \dots, \lambda_m)^T$ and $\mu = (\mu_1, \dots, \mu_n)^T$ are the Lagrange multipliers corresponding to the two sets of constraints $Ax - P \leq 0$ and $x \geq 0$ respectively.

(b) We introduce further 'slack' variables, $S = (S_1, \dots, S_m)^T$ for the constraints $Ax \leq P$, i.e. we define $S = P - Ax \geq 0$.

(c) Application of the Kuhn-Tucker conditions then implies that the canonical problem can be solved using a variant of the Simplex algorithm, using a Simplex 'tableau' (bearing in mind that $D^T = D$) as follows, see e.g. Taha (1976):

$$\begin{pmatrix} -2D & -I & 0 & A^T \\ A & 0 & I & 0 \end{pmatrix} \begin{pmatrix} x \\ \mu \\ S \\ \lambda \end{pmatrix} = \begin{pmatrix} C^T \\ P \end{pmatrix}$$

subject to $\mu_i x_i = 0 = \lambda_j x_j$ for $i = 1, \dots n, j = 1, \dots, m$ and $\lambda, \mu, x, S \geq 0$.

Loosely speaking, the algorithm works as follows:

(i) We start with a feasible solution (i.e. a solution that satisfies all of the constraints), which will be characterised by some value of L;
(ii) We then identify a methodical way of improving on L (if possible), whilst staying within the set of feasible solutions; and
(iii) We iterate (ii) until we run out of ability to improve L.

Conveniently, because of the convex nature of the feasible solution set, the Simplex algorithm is in this case guaranteed to converge. Moreover, because the loss function is quadratic and positive definite, it converges to the global optimum and not just a local optimum.

The relative ease with which it is possible to solve quadratic optimisation problems is one reason why quadratic utility functions are so commonly used in quantitative finance, see Kemp (2010).

However, there is one complication. For the algorithm to work properly we need to start with a 'feasible' solution – if we start with one that isn't feasible then the algorithm sometimes converges to a feasible solution but sometimes it doesn't. We do not necessarily end up with a feasible solution merely by starting with $\lambda, \mu, x, S = 0$, since the left hand side of the tableau then equates to zero,

but the right hand side doesn't. What we therefore need to do is to add additional variables to create a superset of the original optimisation problem for which it is easy to identify a (basic) feasible solution, the additional variables being set up in such a way that when we have converged on the solution to the superset problem we will then either:

(1) Have reached a feasible solution to the original problem, in which case we have the solution; or

(2) We still have an infeasible solution, in which case we can conclude that there was no feasible solution to the original problem.

3. Setting up the 'super'-problem

To set up the superset of the original problem as described in Solving the canonical problem, we multiply through each row in the above tableau (both left and right hand sides) by -1 if the relevant element of C^T or P is negative. This means that the right-hand side of the tableau is all now non-negative. Simultaneously we introduce a further series of 'artificial' variables $R = \left(\bar{R}_1, \dots, \bar{R}_n, \hat{R}_1, \dots, \hat{R}_m\right)^T$, all elements of which are ≥ 0, the first n elements of which, $\bar{R}$, are artificial variables corresponding to the x_i and the remaining m elements are artificial variables corresponding to the S_j. To be more precise, if we want the speediest algorithm, we only introduce only such elements of these variables that are needed to achieve a starting feasible solution.

The starting solution is then given by $x, \mu, \lambda = 0$ and:

$$\bar{R}_i = abs((C^T)_i)$$
$$\left(S_j, \hat{R}_j\right) = \begin{cases} (P_i, 0), & P_i \geq 0 \\ (0, -P_i), & P_i < 0 \end{cases}$$

The starting 'basic' variables, i.e. those whose opening values are greater than zero are each of the $\bar{R}_i$ and whichever of the S_j and the $\hat{R}_j$ is non-zero in the above formulae.

The complete starting tableau is then:

$$\begin{pmatrix} -2D^* & -I^* & 0 & (A^T)^* & I & 0 \\ A^{**} & 0 & I^{**} & 0 & 0 & \tilde{I} \end{pmatrix} \begin{pmatrix} x \\ \mu \\ S \\ \lambda \\ \bar{R} \\ \hat{R} \end{pmatrix} = \begin{pmatrix} (C^T)^* \\ P^{**} \end{pmatrix}$$

Subject to $\mu_i x_i = 0 = \lambda_j x_j$ for $i = 1, \dots n$, $j = 1, \dots, m$ and $\lambda, \mu, x, S \geq 0$ where $(.)^*$ means multiply through relevant row of the matrix by -1 if $(C^T)_i < 0$, $(.)^{**}$ means multiply through relevant row of the matrix by -1 if $P_j < 0$ and $\tilde{I}$ is the $m \times m$ identity matrix with each 1 replaced by a zero if the corresponding value of P_j is greater than zero.

N.B. We have ignored for the purposes of this discussion the possibility that any of the entries on the right-hand side of the tableau are identically zero. This degenerate case can be handled by adding a very small number to make them positive.

We need to know which variables 'correspond' to each other (i.e. appear jointly in the constraints $\mu_i x_i = 0 = \lambda_j x_j$, although these correspondences do not change as the iteration progresses.

In addition to the *Tableau*, which is an $(n+m) \times (3n+3m)$ array we also need to keep track of the following as the iteration proceeds:

Two rows:

(a) *BasicRow*, indicates with, say, a 1 whether the variable in question is 'basic'

(b) *ObjectiveRow*, initially calculated as:

$$ObjectiveRow(k) = \begin{cases} \sum_{q=1}^{n+m} Tableau(q,k), & k = 1, \dots, 2n+2m \\ 0, & k = 2n+2m+1, \dots 3n+3m \end{cases}$$

Two columns:

(c) *SolutionColumn*, contains the current feasible solution, i.e. the right hand side of the above tableau 'equation'; and

(d) *BasicColumn*, contains integers indicating to which variables the entries in the *SolutionColumn* currently apply (and thus which variables are basic, so there is some overlap here with *BasicRow*). *BasicColumn* starts as:

$$BasicColumn(i) = 2n+2m+i \quad i = 1, \dots, n$$
$$BasicColumn(n+j) = \begin{cases} 2n+j, & P_j \geq 0 \\ 3n+2m+j, & P_j < 0 \end{cases} \quad j = 1, \dots m$$

4. Updating the tableau

We update the tableau iteratively (probably only up to some upper limit of number of iterations, in case there is an error in the computation), and we stop when there is no change to the tableau at a given iteration.

To do this we need to identify which variable ought ideally to enter the feasible solution, i.e. to become a 'basic' variable, and which variable it should replace, i.e. which one is ceasing to be a basic variable. We need to do these simultaneously, since the constraints mean that only certain combinations of variables can enter and leave at the same time.

This may be done by identifying the largest positive value of the ObjectiveRow for a column (variable) which is not currently basic (but only if either the current corresponding variable is non-basic, so that the joint constraint of the form value $\mu_i x_i = 0 = \lambda_j x_j$ is still satisfied, or if the corresponding variable is basic then the two can be swapped over and still improve the objective function) *and* if there is another basic variable, which if removed from the feasible set at the same time improves the objective function.

As long as we identify a one entering basic variable (column) and one exiting basic variable (column) as above, we pivot the *Tableau*, *ObjectiveRow* and *SolutionColumn* around their intersection and we update the *BasicRow* and *BasicColumn* accordingly.

As mentioned above, sometimes we need to replace zeros with very small positive numbers to avoid the tableau becoming degenerate.

5. Finishing the algorithm

The algorithm stops when there is no longer any valid combination of entering and leaving basic variables that improves the objective function. If the corresponding solution to the original problem is still not feasible (i.e. there are still some of the additional artificial variables introduced when setting up the super-problem that are greater than zero) then the original problem didn't have a feasible solution. Otherwise, the solution to the original problem is the same as that for the super-problem. If we introduced a change of variables $y = x - q$ at the start of the problem because the lower limits on x were not 0 then we need to unwind this change of variables, as the super-problem solution is defined in terms of y not x.

6. Portfolio optimisation

In a (mean-variance) portfolio optimisation context, the objective that we typically want to maximise is the following (or some monotonic equivalent):

$$U(x) = r.x - \lambda(x - b)^T V(x - b)$$

Here x are the portfolio weights (so typically we impose at least the following constraint $\sum_i x_i = 1$), b is the benchmark (or 'minimum risk' portfolio), r is a vector of assumed returns on each asset and V is the covariance matrix ($= s^T C s$, where s is the vector of risks on each asset class, here assumed to be characterised by their volatilities, as this approach is merely a mean-variance one, and C their correlation matrix).

Nematrian website tools

The main tools that the Nematrian website makes available for constrained quadratic optimisation are:

(a) Constrained Quadratic Optimiser. All purpose constrained quadratic optimiser.

(b) Constrained Quadratic Portfolio Optimiser. Equivalent tool with inputs specifically tailored to the portfolio optimisation problem.

(c) Reverse Quadratic Optimiser. Works out the 'implied alphas', i.e. the return assumptions that need to be held for a portfolio to be optimal (ignoring constraints), given active positions, standard deviations, a correlation matrix and a trade-off factor (i.e. risk aversion factor) that corresponds to the investor's chosen trade-off between return and risk. Please bear in mind that if a given set of returns, r, is optimal in this context then so are $ar + b$ for any constant (asset class independent) values for a and b.

(d) Tools for plotting efficient frontiers, including MnPlotQuadraticEfficientFrontier and MnPlotQuadraticEfficientPortfolios, which plot the efficient frontier (in risk-return space) and the portfolios making up the efficient frontier.

References

Kemp, M.H.D. (2010). *Extreme Events: Robust Portfolio Construction in the Presence of Fat Tails*. John Wiley & Sons [see ExtremeEvents for further details]

Taha, H.A. (1976). *Operations Research – An Introduction*. Macmillan