portfolio selection cookbook

Jiazhou Wang

Jan. 2017

Contents

1 Introduction

This handbook is written under Matlab code and python code, for the convex optimization part in python, this cookbook use the python package CVXOPT, a self-supported optimizer will be an interesting topic in the later version.

2 Markowitz portfolio selection

2.1 One period multiple instruments case

The basic Markowitz portfolio selection is trying to minize the following problem:

$$\underset{w}{minimize} \quad \frac{1}{2} w^T \Sigma w - r^T w$$

which the answer is fairly simple:

$$w_{opt} = \Sigma^{\dagger} r$$

where $\Sigma^{\dagger}$ is the pseudo-inverse of Σ

The main issue here is computation issue, i.e.: how to compute $\Sigma^{\dagger} r$ correctly? Notice that here the problem is "compute $\Sigma^{\dagger} r$ correctly", not compute $\Sigma^{\dagger}$ correctly, which means one does not necessarily compute $\Sigma^{\dagger}$ directly. This leads to the first method:

Algorithm 1: Solving Markowitz portfolio selection

Result: Given Σ and r, compute w_{opt}
Input Σ, r;
On Matlab:;
  $w_{opt} = \Sigma \backslash r$;
On Python:;
  w_{opt} = scipy.linalg.solve(Σ, r);
  w_{opt} = numpy.linalg.lstsq(Σ, r)[0];
Using Cholesky decomposition on Python:;
  L = numpy.linalg.cholesky(Σ);
  solver = scipy.linalg.solve_triangular;
  w_{opt} = solver(L, solver(L.T, r, lower = False), lower = True);

Comments: numpy.linalg.solve has a potential issue. It may not work as expected under MKL(intel Math Kernel Library), scipy.linalg.solve may not have this problem, but it is not confirmed yet. If Σ is positive definite, the least square method and cholesky decomposition is recommended. Solving normal equation (which is $w = (\Sigma^T \Sigma)^{-1} \Sigma^T r$) is not in the list because this method will double the condition number of Σ, which usually has some potential numerical issue.

2.2 Multiple periods multiple instruments case

The multiple periods version of Markovitz portfolio selection is trying to solve the following problem:

$$\underset{w_i}{minimize} \quad \sum_{i=1}^{n}(\frac{1}{2}w_i^T\Sigma_i w_i - r_i^T w_i)$$

An equivalent representation of this problem is:

$$\begin{aligned} \underset{\tilde{w}}{minimize} \quad & \frac{1}{2}\tilde{w}^T\tilde{\Sigma}\tilde{w} - \tilde{r}^T\tilde{w} \\ \text{such that} \quad & \tilde{w} = append(w_1, w_2, ..., w_n) \\ & \tilde{\Sigma} = block\_diag(\Sigma_1, \Sigma_2, ..., \Sigma_n) \end{aligned}$$

Notice that in this case the minimum of the sum is the sum of the minimum, i.e.:

$$\min(\sum_{i=1}^{n}(\frac{1}{2}w_i^T\Sigma_i w_i - r_i^T w_i)) = \sum_{i=1}^{n}\min(\frac{1}{2}w_i^T\Sigma_i w_i - r_i^T w_i))$$

This shows that one can optimize each period individually and then sum the solution together.

3 Markowitz portfolio selection with linear transaction cost

In reality, one cannot always get into the desired position for free. Usually this part is modeled as what people called "transaction cost". As a good starting point, linear (or proportional) transaction costs is very easy to implement into Markovwitz portfolio selection framework.

3.1 One period multiple instruments case

The Markowitz portfolio selection with linear transaction cost is being formed as the following:

$$\underset{w}{minimize} \quad (\frac{1}{2}w^T\Sigma_i w - r^T w + c^T|w - w_0|)$$

3.2 Multiple periods multiple instruments case

The multiple periods version of this problem is as the following:

$$\underset{w_i}{minimize} \quad \sum_{i=1}^{n}(\frac{1}{2}w_i^T\Sigma_i w_i - r_i^T w_i + c^T|w_i - w_{i-1}|)$$

To solve this problem, one can us buy-sell split trick, i.e.:

$$\begin{aligned} w_i &= w_0 + \sum_{k=1}^{k\leq i}(b_k - s_k) \\ b_k &\geq 0 \\ s_k &\geq 0 \end{aligned}$$

then the problem becomes to:

$$\begin{aligned} \underset{b_i,s_i}{minimize} \quad & \sum_{i=1}^{n}(\frac{1}{2}(w_0 + \sum_{k=1}^{k\leq i}(b_k - s_k))^T\Sigma_i(w_0 + \sum_{k=1}^{k\leq i}(b_k - s_k)) - r_i^T(b_i - s_i) + c^T(b_i + s_i)) \\ \text{such that} \quad & b_i \geq 0 \\ & s_i \geq 0 \end{aligned}$$

which is equivalent to:

$$\begin{aligned} \underset{b_i,s_i}{minimize} \quad & \frac{1}{2}(b-s)^T L^T\Sigma L(b-s) + (2L^T\Sigma\tilde{w}_0 - \tilde{r})^T(b-s) + \tilde{c}^T(b+s) \\ & = \begin{bmatrix} b & s \end{bmatrix} \begin{bmatrix} L^T\Sigma L & -L^T\Sigma L \\ -L^T\Sigma L & L^T\Sigma L \end{bmatrix} \begin{bmatrix} b \\ s \end{bmatrix} + \begin{bmatrix} 2L^T\Sigma\tilde{w}_0 - \tilde{r} - \tilde{c} & -2L^T\Sigma\tilde{w}_0 + \tilde{r} + \tilde{c} \end{bmatrix} \begin{bmatrix} b \\ s \end{bmatrix} \\ \text{such that} \quad & b \geq 0 \\ & s \geq 0 \end{aligned}$$

where

$$
\begin{aligned}
L &= L_0 \otimes I \\
&= \begin{bmatrix} 1 & 0 & 0 & ... & 0 & 0 \\ 1 & 1 & 0 & ... & 0 & 0 \\ 1 & 1 & 1 & ... & 0 & 0 \\ ... & ... & ... & ... & ... & ... \\ 1 & 1 & ... & ... & 1 & 0 \\ 1 & 1 & ... & ... & 1 & 1 \end{bmatrix} \otimes \begin{bmatrix} 1 & 0 & 0 & ... & 0 & 0 \\ 0 & 1 & 0 & ... & 0 & 0 \\ 0 & 0 & 1 & ... & 0 & 0 \\ ... & ... & ... & ... & ... & ... \\ 0 & 0 & ... & ... & 1 & 0 \\ 0 & 0 & ... & ... & 0 & 1 \end{bmatrix} \\
\tilde{w_0} &= e \otimes w_0 \\
\Sigma &= block\_diag(\Sigma_i) \\
\tilde{r} &= e \otimes r \\
\tilde{c} &= e \otimes c
\end{aligned}
$$

Although one can just simply compute the hessian and gradient and then send them into the optimizer (quadprob in Matlab or CVXOPT in the python), a self-supported function always has better performance in this case.

In CVXOPT, one can support a KKT solver which solve the following linear system:

$$
\begin{bmatrix} P & A^T & G^T W^{-1} \\ A & 0 & 0 \\ G & 0 & -W^T \end{bmatrix} \begin{bmatrix} u_x \\ u_y \\ u_z \end{bmatrix} = \begin{bmatrix} b_x \\ b_y \\ b_z \end{bmatrix}
$$

where P is the hessian. A is coming from the equality constraints $Ax = b$, G is coming from the inequality constraints $Gx \geq h$. W is Lagrangian multipliers and positive multiples of hyperbolic Householder transformations. Empirically W only has Lagrangian multipliers in this kind of portfolio selection problem, which means W is a diagonal matrix.

In this problem, $A = 0$, so the KKT system is reduced to the following:

$$
\begin{bmatrix} P & 0 & G^T W^{-1} \\ 0 & 0 & 0 \\ G & 0 & -W^T \end{bmatrix} \begin{bmatrix} u_x \\ u_y \\ u_z \end{bmatrix} = \begin{bmatrix} b_x \\ b_y \\ b_z \end{bmatrix}
$$

which yields u_y and b_y are redundant. Then we have a reduced KKT system:

$$\begin{bmatrix} P & G^T W^{-1} \\ G & -W^T \end{bmatrix} \begin{bmatrix} u_x \\ u_z \end{bmatrix} = \begin{bmatrix} b_x \\ b_z \end{bmatrix}$$

Let

$$P = \begin{bmatrix} L^T \Sigma L & -L^T \Sigma L \\ -L^T \Sigma L & L^T \Sigma L \end{bmatrix}$$

$$G = \begin{bmatrix} I & 0 \\ 0 & I \end{bmatrix}$$

we can rewrite the KKT system as the following:

$$\begin{aligned} P u_x + W^{-1} u_z &= b_x \\ u_x - W^T u_z &= b_z \end{aligned}$$

which gives us:

$$\begin{aligned} u_z &= (W^{-1} + P W^T)^{-1} (b_x - P b_z) \\ u_x &= b_z + W^T u_z \end{aligned}$$

And if we take a deeper look into this P we have:

$$P = \begin{bmatrix} L^T \Sigma L & -L^T \Sigma L \\ -L^T \Sigma L & L^T \Sigma L \end{bmatrix}$$

$$\begin{aligned} Px &= \begin{bmatrix} L^T \Sigma L & -L^T \Sigma L \\ -L^T \Sigma L & L^T \Sigma L \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \\ &= \begin{bmatrix} y_1 \\ -y_1 \end{bmatrix} \end{aligned}$$

where:

$$y_1 = L^T \Sigma L (x_1 - x_2)$$

So the algorithm can be written as the following:

Algorithm 2: KKT exploiting structure in CVXOPT

Result: Given P, G, W, b_x, b_y, b_z, compute u_x, u_y, u_z

```
bx1, by1, bz1 = map(lambda x: numpy.array(x).ravel(), [bx, by, bz]);
invd = numpy.array(W['di']).ravel();
d = numpy.array(W['d']).ravel();
L = numpy.linalg.cholesky(numpy.diag(invd) + P * d[:, None]);
solver = scipy.linalg.solve_triangular;
uz = solver(L, solver(L.T, bx1 - Px(bz1), lower = False), lower =
 True);
ux = bz1 + d * uz;
blas.copy(cvxopt.matrix(ux), bx);
blas.copy(cvxopt.matrix(uz), bz);
```

Here mv is a function that computing a vector multiplied by a matrix. Px(P, x) returns the result of Px, which has a fast version as the following:

Algorithm 3: compute Px

Result: Given Σ, $d = \mathrm{diag}(D)$ and x, compute $y = Px$

```
n = x.shape[0]/2;
m = n / d;
y = numpy.zeros(x.shape);
x1 = x[:n];
x2 = x[n:];
y1 = x1 - x2;
for i in xrange(1, m):;
   k = i * d; k0 = k - d; k1 = k + d;;
   y1[k: k1] = y1[k0: k] + y1[k: k1];
for i in xrange(m):;
   y1[i * d: (i+1) * d] = mv(Σ, y1[i * d: (i+1) * d]);
y1 = y1[::-1];
for i in xrange(1, m):;
   k = i * d; k0 = k - d; k1 = k + d;;
   y1[k: k1] = y1[k0: k] + y1[k: k1];
y1 = y1[::-1];
y[:n] = y1;
y[n:] = -y1;
```

And the user can also support two functions to CVXOPT which compute Px and Gx. From above we have seen how Px is computed, and we have $Gx = x$, so these two supporting functions are recommended and not hard to write.

4 Markowitz portfolio selection with linear and quadratic transaction cost

4.1 One period multiple instruments case

Problem description:

$$\underset{w}{minimize} \quad \frac{1}{2}w^T\Sigma w - r^T w + c^T|w - w_0| + (w - w_0)^T D(w - w_0)$$

where D is a diagonal matrix which represents the quadratic transaction cost.

4.2 Multiple periods multiple instruments case

The multiple periods version is:

$$\underset{w_i}{minimize} \quad \sum_{i=1}^{n}(\frac{1}{2}w_i^T\Sigma_i w_i - r_i^T w_i + c_i^T|w_i - w_{i-1}| + (w_i - w_{i-1})^T D_i(w_i - w_{i-1}))$$

We can write down an equivalent convex optimization problem using buy-sell split:

$$\begin{aligned} \underset{b,s}{minimize} \quad & f = \frac{1}{2}\begin{bmatrix} b & s \end{bmatrix}\begin{bmatrix} I & I \\ -I & I \end{bmatrix}\begin{bmatrix} L^T\tilde{\Sigma}L & 0 \\ 0 & 2D \end{bmatrix}\begin{bmatrix} I & -I \\ I & I \end{bmatrix}\begin{bmatrix} b \\ s \end{bmatrix} + g^T\begin{bmatrix} b \\ s \end{bmatrix} \\ \text{subject to} \quad & b \geq 0 \\ & s \geq 0 \end{aligned}$$

where I is identity matrix, $\otimes$ is Kronecker product, and

$$
\begin{aligned}
L &= L_0 \otimes I \\
&= \begin{bmatrix} 1 & 0 & 0 & ... & 0 & 0 \\ 1 & 1 & 0 & ... & 0 & 0 \\ 1 & 1 & 1 & ... & 0 & 0 \\ ... & ... & ... & ... & ... & ... \\ 1 & 1 & ... & ... & 1 & 0 \\ 1 & 1 & ... & ... & 1 & 1 \end{bmatrix} \otimes \begin{bmatrix} 1 & 0 & 0 & ... & 0 & 0 \\ 0 & 1 & 0 & ... & 0 & 0 \\ 0 & 0 & 1 & ... & 0 & 0 \\ ... & ... & ... & ... & ... & ... \\ 0 & 0 & ... & ... & 1 & 0 \\ 0 & 0 & ... & ... & 0 & 1 \end{bmatrix}
\end{aligned}
$$

and:

$$
\begin{aligned}
\tilde{\Sigma} &= I \otimes \Sigma \\
g &= \begin{bmatrix} (e \otimes w_0)^T \Sigma L - r^T L & \tau^T \end{bmatrix}^T \\
&= \begin{bmatrix} \tau & L^T (r - \Sigma (e \otimes w_0)) \end{bmatrix}
\end{aligned}
$$

Here we assume for every period the covariance matrix Σ, linear transaction cost multiplier c_i and quadratic transaction cost multiplier D_i keep the same.

4.2.1 In Matlab

under construction

4.2.2 In CVXOPT

The support function is as same as the linear transaction cost case, which is the following:
which gives us:

$$
\begin{aligned}
u_z &= (W^{-1} + PW^T)^{-1}(b_x - Pb_z) \\
u_x &= b_z + W^T u_z
\end{aligned}
$$

the difference is here:

$$
P = \begin{bmatrix} L^T \tilde{\Sigma} L + 2D & -L^T \tilde{\Sigma} L + 2D \\ -L^T \tilde{\Sigma} L + 2D & L^T \tilde{\Sigma} L + 2D \end{bmatrix}
$$

and the algorithm for computing Px is written on the following:

Algorithm 4: compute Px

Result: Given Σ, $d = \text{diag}(D)$ and x, compute $y = Px$

```
n = x.shape[0]/2;
m = n / d;
y = numpy.zeros(x.shape);
x1 = x[:n];
x2 = x[n:];
y1 = x1 - x2;
for i in xrange(1, m):;
   k = i * d; k0 = k - d; k1 = k + d;;
   y1[k: k1] = y1[k0: k] + y1[k: k1];
for i in xrange(m):;
   y1[i * d: (i+1) * d] = mv(Σ, y1[i * d: (i+1) * d]);
y1 = y1[::-1];
for i in xrange(1, m):;
   k = i * d; k0 = k - d; k1 = k + d;;
   y1[k: k1] = y1[k0: k] + y1[k: k1];
y1 = y1[::-1];
xx = 2* d * (x1 + x2);
y[:n] = y1 + xx;
y[n:] = -y1 + xx;
```

And $Gx = x$ still holds.

To be continued.