About a few solvers for Markowitz portfolio selection problem

Bogusz Jelinski

Septemper 24th, 2016

Abstract

The most commonly suggested tool to solve Markowitz portfolio selection problem is Excel, which might be unwieldy for bigger models. The goal was to verify if other modelling tools can be used for this particular purpose, both commercial and open-source ones were reviewed. Some slight differences in numerical solutions provided by these tools are presented. As some developments in the modelling of capital markets are attributed to the computational complexity of the classical Markowitz model, the ease of use and performance were in focus.

1 Introduction

Markowitz portfolio selection model [1] was in itself quite simple but the computational part of the problem was so troublesome that it lead to different simplifications [2]. The development of microcomputers gave mass access to nonlinear solvers allowing the solution to the original quadratic problem. Let us remind it, there are two values - expected return of a portfolio (E) and variance of the portfolio (V) being its risk measure:

$$E = \sum_{i=1}^{N} x_i \mu_i$$

$$V = \sum_{i=1}^{N} \sum_{j=1}^{N} \sigma_{ij} x_i x_j$$

$$\sum_{i=1}^{N} x_i = 1$$

$$x_i \geqslant 0 \; for \; i = 1..N$$

As we can see, to solve the problem we have to calculate $n(n-1)/2$ covariance pairs and n variances. Two variants are usually considered - maximize expected return with variance not greater than an assumed value v_{max}:

$$E \rightarrow MAX$$

$$V \leqslant v_{max}$$

or minimize variance with expected return not lesser than an assumed value e_{min}:

$$V \to MIN$$

$$E \geqslant e_{min}$$

Below you will find implementations of the second variant (minimize the risk) in four popular languages. This issue seems to be trivial today in terms of complexity.

Numerical examples are based on stock prices of Bank of America Corporation, Ford Motor Company, General Electric and AT&T. Returns include both price gains and dividends and were calculated for twelve consecutive quarters starting on Jan 1st 2010.

2 AMPL

AMPL is a commercial tool [3], but it is available for students and teachers at no charge. Input for the solver is split into the model and the data file. Let us name the model file `mpt.mod`:

```
set A; # assets
set T := {1..12};
param Rets {T,A}; # returns on assets
param mean {j in A} = (sum {i in T} Rets[i,j]) / card(T);
param Diff {i in T, j in A} = Rets[i,j] - mean[j];
param Covar {i in A, j in A} =
   sum {k in T} (Diff[k,i]*Diff[k,j] / (card{T}-1));
var X {A} >=0;
var Mean = sum {j in A} mean[j] * X[j];
minimize Risk: sum {i in A} (X[i]*(sum {j in A} Covar[i,j]*X[j]));;
subject to TotalOne: sum {j in A} X[j] = 1;
subject to Reve: Mean >= 0.035;
```

and the data file `mpt.dat`:

```
set A := BAC F GE T ;
param Rets: BAC F GE T :=
1 0.1383 0.2228 0.1845 -0.0812
2 -0.2029 -0.2019 -0.2079 -0.0574
3 -0.0649 0.158 0.1593 0.1923
4 0.0038 0.3695 0.1265 0.0344
5 -0.0599 -0.1357 0.1045 0.0462
6 -0.1795 -0.0904 -0.0654 0.0398
```

```
7 -0.4472 -0.3103 -0.1995 -0.0862
8 0.0072 0.1483 0.2308 0.0891
9 0.6517 0.1258 0.1024 0.0425
10 -0.1539 -0.2361 0.0522 0.1479
11 0.0981 0.0554 0.1166 0.0536
12 0.2969 0.3092 -0.0715 -0.0954
```

You can now use AMPL graphical user interface - `amplide` - to run the following commands:

```
model mpt.mod;
data mpt.dat;
solve;
display X
```

You will get the following result:

```
MINOS 5.51: optimal solution found.
5 iterations, objective 0.01062155067
Nonlin evals: obj = 11, grad = 10.
ampl: display X;
X [*] :=
BAC  0
  F  0.0428628
 GE  0.437821
  T  0.519316
;
```

3 Julia

> Julia is a high-level, high-performance dynamic programming language for technical computing [4].

Julia and its packages [5] [6] are licensed under MIT, GPL, MPL, BSD and EPL licenses. Let us put the Julia code into `mpt.jl`. I omitted covariance calculation for the sake of clarity:

```
using JuMP
using Ipopt
N=4
r_min=0.035
m = Model()
```

```
C = [0.076900 0.039968 0.018111 -0.000288 ; 0.039968 0.050244 0.019033
   -0.000060 ; 0.018111 0.019033 0.021381 0.007511 ; -0.000288 -0.000060
   0.007511 0.008542]
Mean = [0.0073 0.0346 0.0444 0.0271]
@variable(m, 0 <= x[i=1:N] <= 1)
@objective(m,Min,sum{x[j]*sum{x[i]*C[i,j],i=1:N},j=1:N})
@constraint(m, sum{x[i]*Mean[i],i=1:N} >=r_min)
@constraint(m, sum{x[i],i=1:N} ==1.0)
status = solve(m)
for i= 1:N
 println(getvalue(x[i]))
end
```

Then after running `julia mpt.jl` we will get:

```
EXIT: Optimal Solution Found.
1.3095345121754e-7
0.043278618483219226
0.4378847609266113
0.5188364896367182
```

4 Python

Let us put the Python code into `mpt.py`[1]:

```
from cvxopt import matrix
from cvxopt.solvers import qp
import numpy as np
n = 4
Cov = matrix([[0.076900,  0.039968, 0.018111, -0.000288 ],
  [ 0.039968,  0.050244, 0.019033, -0.000060 ],
  [ 0.018111,  0.019033, 0.021381,  0.007511 ],
  [-0.000288, -0.000060, 0.007511,  0.008542 ]])
Mean = matrix([0.0073, 0.0346, 0.0444, 0.0271])
r_min = 0.035
G = matrix(np.concatenate((-np.transpose(Mean), -np.identity(n)), 0))
h = matrix(np.concatenate((-np.ones((1,1))*r_min, np.zeros((n,1))), 0))
```

[1]Code based on example found in [7]

```
A = matrix(1.0, (1,n))
b = matrix(1.0)
q = matrix(np.zeros((n, 1)))
sol = qp(Cov, q, G, h, A, b)
print(sol['x'])
```

Then after having executed `python mpt.py` we will get:

```
Optimal solution found.
[ 8.47e-07]
[ 4.33e-02]
[ 4.38e-01]
[ 5.19e-01]
```

Cvxopt and NumPy are licensed respectively under GPL and BSD.

5 C

The Ipopt library [6] is distributed with an example of usage — the `hs071_c.c` file — with four variables, two constraints and goal minimisation, which perfectly fits our needs.

The covariance matrix is symmetric so the gradient of the objective is given by

$$\begin{bmatrix} 2 * \sum_{i=1}^{N} x_i c_{i1} \\ 2 * \sum_{i=1}^{N} x_i c_{i2} \\ 2 * \sum_{i=1}^{N} x_i c_{i3} \\ 2 * \sum_{i=1}^{N} x_i c_{i4} \end{bmatrix}$$

and the Jacobian of the constraints $g(x)$ is

$$\begin{bmatrix} \mu_1 & \mu_2 & \mu_3 & \mu_4 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

and the Hessian of the Lagrangian function is the covariance matrix — see the gradient of the objective above.

Make the following changes to adapt this example file to our optimisation problem. First remove the whole if-block with the second call to IpoptSolve, most probably between lines 165 and 206. Then put covariance and mean matrices somewhere at the beginning making them global:

```
Number c[4][4] =
  {{0.076900, 0.039968, 0.018111, -0.000288},
   {0.039968, 0.050244, 0.019033, -0.000060},
   {0.018111, 0.019033, 0.021381, 0.007511},
   {-0.000288, -0.000060, 0.007511, 0.008542}};
Number MoR[4] = {0.0073, 0.0346, 0.0444, 0.0271};
```

Secondly move the assignment of the number of stocks/variables to the beginning of the `main` function and change the hardcoded values of nonzeros in the Jacobian and the Hessian:

```
Index n=4;
Index nele_jac = 2*n;
Index nele_hess = n*(n+1)/2;
```

Then change the lower and upper limits of variables:

```
for (i=0; i<n; i++) {
    x_L[i] = 0.0;
    x_U[i] = 1.0;
  }
```

and the limits of constraints - the minimum return and the sum of variables:

```
g_L[0] = 0.035;
g_U[0] = 1e19;
g_L[1] = 1;
g_U[1] = 1;
```

The `eval_f` goal function should contain:

```
Number sum;
int i,j;
*obj_value=0.0;
for (i=0; i<n; i++)
  { sum=0.0;
    for (j=0; j<n; j++)
       sum += x[j] * c[j][i];
    *obj_value += x[i]*sum;
  }
```

The `eval_grad_f` gradient calculation shall change to:

```
int i,j;
for (i=0; i<n; i++)
```

```
  { grad_f[i] =0;
    for (j=0; j<n; j++)
      grad_f[i] += x[j]*c[j][i];
    grad_f[i] = grad_f[i] * 2;
  }
```

The two contraints are calculated within `eval_g`:

```
int i;
g[0]=0.0;
for (i=0; i<n; i++) g[0] += x[i]*MoR[i];
g[0] += my_data->g_offset[0];

g[1]=0;
for (i=0; i<n; i++) g[1] += x[i];
g[1] += my_data->g_offset[1];
```

Finanaly there are two functions left - the Jacobian, `eval_jac_g`:

```
int i=0;
if (values == NULL) {
  for (i=0; i<n; i++) {
      iRow[i] = 0;
      jCol[i] = i;
  }
  for (i=0; i<n; i++) {
      iRow[n+i] = 1;
      jCol[n+i] = i;
  }
}
else {
  for (i=0; i<n; i++)
       values[i] = MoR[i];
  for ( ; i<n+n; i++)
       values[i] = 1;
}
```

and the Hessian, `eval_h`:

```
if (values == NULL) {
  idx=0;
  for (row = 0; row < n; row++) {
    for (col = 0; col <= row; col++) {
        iRow[idx] = row;
        jCol[idx] = col;
        idx++;
    }
  }
}
else {
  idx=0;
  for (row = 0; row < n; row++) {
    for (col = 0; col <= row; col++) {
        values[idx] = obj_factor * (2*c[row][col]);
        idx++;
    }
  }
}
```

You have to remove -pedantic-errors from the Makefile to compile this if you have not placed variable definitions at the beginning of each function. Running the program will give the following result:

```
x[0] = 0.000000e+00
x[1] = 4.327805e-02
x[2] = 4.378846e-01
x[3] = 5.188373e-01
```

6 An example with twenty stocks

Let us see the results for twenty stocks selected out of S&P100 for twelve consecutive quarters starting on Jan 1st 2010. The same input data gave slightly different results for four tools (Julia also uses Ipopt):

Software	AMPL	Python (Ipopt)	C (Ipopt)	MS Excel
Objective (risk)	0.002900	0.002874	0.002898	0.003178
AIG			0.0003	0.0209
BAC	0.0067		0.0067	
IBM	0.1892	0.1890	0.1890	0.2743
KO	0.0718	0.0721	0.0719	0.0989
MCD	0.4291	0.4290	0.4290	0.4852
S	0.0966	0.0965	0.0965	0.1183
VZ	0.2066	0.2060	0.2066	0.0019

Table 1: Portfolio structures computed by different tools

As we can see, Ipopt found the best solution - the portfolio with the lowest risk. MS Excel excludes Verizon (XOM with 0.0005 is not shown in the table). All tools provide results in the blink of an eye — a model with 200 stocks gets solved by AMPL within less than one second.

7 Conclusions

The Markowitz portfolio selection problem is no longer a computational trouble. There are open-source tools to solve models with hundreds of variables within seconds on a personal computer. You can use the language which suites you best. Especially Python deserves recommendation as it gets more and more popular in the scientific community. These tools can be fed with data in a much more convenient way than editing formulas in an office spreadsheet, which moreover has the limit of number of variables.

It is found that different tools can suggest different portfolio weights and different portfolio variance values with the same constraints (see table 1) - another argument to use open-source tools.

References

[1] Markowitz H.: Portfolio Selection. The Journal of Finance, Vol. 7, No. 1., pp. 77-91, (March, 1952)

[2] Sharpe W.F.: A Simplified Model for Portfolio Analysis, Management Science, Vol. 19, No. 2., pp. 277-293, (January, 1963)

[3] http://ampl.com/resources/the-ampl-book/

[4] http://julialang.org/

[5] https://jump.readthedocs.io/en/latest/

[6] https://projects.coin-or.org/Ipopt

[7] https://wellecks.wordpress.com/2014/03/23/portfolio-optimization-with-python/

[8] http://cvxopt.org/

[9] http://www.numpy.org/