MIE1621 – Non-Linear Programming

Course Project

Fall 2017

Jonathan Smith

smithj70

999694521

Dec 7, 2017

Part 1: Problem formulation

The problem at hand is the optimization of the risk adjusted return of a financial portfolio of n assets. The risk adjusted return is governed by the following equations:

$$\sum_{i=1}^{n} \mu_i x_i - \frac{\delta}{2} \sum_{i=1}^{n} \sum_{i=1}^{n} \sigma_{ij} x_i x_j$$
$$\text{subject to}$$
$$\sum_{i=1}^{n} x_i = 1$$

Where:
μ_i is the return of asset i,
σ_{ij} is the covariance of asset I and j,
x_i is the fraction of wealth invested in asset i,
δ is the risk tolerance of the investor, set to a value between 3.5 and 4.5.

The goal of this optimization will be to maximize the expected return, given some risk tolerance. The equation is therefore multiplied by -1 in order to apply conventional minimization techniques.

This equation can be re-written in matrix form for scalability:

$$f = \frac{\delta}{2} x^T \sigma x - x^T \mu \qquad \text{s.t.} \qquad x^T e = 1 \qquad \text{where } e = diag(I_n)$$

In order to reconfigure this equation as an unconstrained optimization, the Lagrangian of this problem is constructed, ensuring that the optimal solutions meet the constraint:

$$L(x, \pi) = \frac{\delta}{2} x^T \sigma x - x^T \mu + \pi(x^T e - 1)$$

In order to satisfy the FONC, the following partial derivatives of L are found:

$$\frac{\partial L}{\partial x} = \delta \sigma x - \mu + \pi e = 0$$
$$\frac{\partial L}{\partial \pi} = x^T e - 1 = 0$$

In order to solve these simultaneously, the following equation is constructed, where σ is an nxn matrix, and x, e, μ are an n-dimensional column vectors:

$$\nabla L = \begin{pmatrix} \delta\sigma & e \\ e & 0 \end{pmatrix} \begin{pmatrix} x \\ \pi \end{pmatrix} - \begin{pmatrix} \mu \\ 1 \end{pmatrix}$$

Lastly, the hessian can be shown as:

$$\mathrm{H} = \begin{pmatrix} \delta\sigma & e \\ e & 0 \end{pmatrix}$$

The gradient for a simple 2-stock portfolio is shown as below, however, this generalizes for n stocks.

$$\nabla L = \begin{pmatrix} \delta\sigma_{11} & \delta\sigma_{12} & 1 \\ \delta\sigma_{21} & \delta\sigma_{22} & 1 \\ 1 & 1 & 0 \end{pmatrix} \begin{pmatrix} x_1 \\ x_1 \\ \pi \end{pmatrix} - \begin{pmatrix} \mu_1 \\ \mu_2 \\ 1 \end{pmatrix}$$

Part 2: Naïve Newton, SDM, BFGS Solution Methods

This portion of the report details attempts to solve the above problem using Newton's Method, Steepest Descent Method, and BFGS Quasi-Newton Method, with a step length of 1. The functions to execute this code are shown in Appendix A. Note that these functions cannot be run independently of other accompanying functions. The full python notebook is submitted along with this report, or available at this link: https://github.com/jonsmith359/MIE1621-Project.

These methods all follow the same basic form of iterations:

$$x^{k+1} = x^k - \alpha^k d^k$$

where α^k is a step length and d^k is a descent direction. For this portion of the project, $\alpha^k = 1$. The way that these three methods differ is in their calculation of descent direction.

Newton: $d = \mathrm{H}^{-1}\nabla L$

SDM: $d = \nabla L$

BFGS: $d = \mathrm{H}^{-1}\nabla L$

In the case of BFGS, an approximation of the inverse hessian is used, based on the estimation formula:

$$H^{-1}{}_{k+1} = (\mathrm{I} - \frac{s_k y_k^T}{y_k^T s_k}) H^{-1}{}_{k+1} \left(\mathrm{I} - \frac{y_k s_k^T}{y_k^T s_k}\right) + \frac{s_k s_k^T}{y_k^T s_k}$$

These three methods were performed on the sample dataset given, with starting portfolio weights of [1,1,1] and risk factor 3.5. Iterations are performed until the current solution vector is sufficiently close to the previous solution vector. Each element must be within some user defined tolerance of the previous. The results are summarized below.

	Return	var_1	var_2	var_3
0	0.1073	0.02778	0.00387	0.00021
1	0.0731	0.00387	0.01112	-0.0002
2	0.0621	0.00021	-0.0002	0.00115

Table 1: Sample Data Set

Method	Convergence	# Iterations	Time
Newton	Yes	1	4.05 µs
SDM	No	100	5.01 µs
BFGS	Yes	17	5.03 µs

Table 2: Part 2 Results

From these results, we can see that Newton's method performs the best, reaching the optimal portolio: x = [0.45526555 0.1745838 0.37015065] , Return: 7.25% within a single iteration. This is because of the use of an exact gradient and hessian at initial point x. The steepest descent method diverges for this, and other starting points. This occurs because the Lagrangian function is non-convex. Steepest descent is unable to guarantee a solution under these conditions. It should be noted however, that the iterations come close to the optimum, but then diverge from it. The BFGS method converges to the solution after 17 iterations. The added iterations from Newton are because the initial hessian approximation is based on the identity matrix. Overall, these methods perform as expected.

Part 3 Newton, SDM, BFGS Solution Methods with Backtracking

Backtracking is implemented in order to find an optimal step length in the equation

$$x^{k+1} = x^k - \alpha^k d^k$$

The backtrack lines search works by assuming a step length of 1, and decreasing it by half until the following condition is satisfied:

$$f(x^{k+1}) \leq f(x^k) + \gamma * \alpha^k * d^k * \nabla f(x^k)$$

This is intended to ensure that the step length is not too large so as to miss the critical point, but still large enough to make meaningful progress towards the point. An additional constraint is implemented in the code to ensure that the step length does not approach zero on each iteration

The methods in part 2 were repeated with this line search to achieve the following results:

Method	Convergence	# Iterations	Time
Newton	Yes	1	3.81 µs
SDM	No	100	7.15 µs
BFGS	Yes	26	4.05 µs

Table 3: Part 3 Results

As was the case in part 2, Newton converged in one step due to the exact gradient and hessians used in the computation. SDM still diverges under these conditions due to the non-convexity of the function. As was the case with part 2, The BFGS method still converges to the correct answer, but this happens in more iterations when backtracking is used. This occurs because a smaller step length is used, leading to a more conservative step between iterations.

Part 4: Scaling up

The following dataset was created using daily data for the stocks: AAPL, GOOG, SAF.PA, AIR.DE, LMT, BA, UTC1.DE, BBD-B.TO, GE, RR.L in the year leading up to Dec 5 2017. The return is the average daily return throughout the year, while the covariance's are calculated based on the daily changes of each stock with respect to one another.

	Return	var_1	var_2	var_3	var_4	var_5	var_6	var_7	var_8	var_9	var_10
0	0.1465	0.0118	0.0082	0.0098	0.0033	0.0073	0.0174	-0.0014	0.0071	-0.0132	0.0110
1	0.9031	0.0082	0.0071	0.0082	0.0040	0.0055	0.0130	-0.0006	0.0059	-0.0101	0.0087
2	0.0769	0.0098	0.0082	0.0110	0.0038	0.0074	0.0177	-0.0014	0.0064	-0.0128	0.0111
3	0.0940	0.0033	0.0040	0.0038	0.0341	0.0018	0.0045	0.0003	0.0043	-0.0054	0.0031
4	0.2831	0.0073	0.0055	0.0074	0.0018	0.0058	0.0139	-0.0017	0.0051	-0.0097	0.0074
5	0.2056	0.0174	0.0130	0.0177	0.0045	0.0139	0.0346	-0.0046	0.0137	-0.0243	0.0174
6	0.1037	-0.0014	-0.0006	-0.0014	0.0003	-0.0017	-0.0046	0.0016	-0.0015	0.0028	-0.0009
7	0.0024	0.0071	0.0059	0.0064	0.0043	0.0051	0.0137	-0.0015	0.0153	-0.0122	0.0057
8	0.0270	-0.0132	-0.0101	-0.0128	-0.0054	-0.0097	-0.0243	0.0028	-0.0122	0.0192	-0.0120
9	0.8350	0.0110	0.0087	0.0111	0.0031	0.0074	0.0174	-0.0009	0.0057	-0.0120	0.0135

Table 4: Scaled-Up Data

When these results are fed into the same models described above, with starting weights of 1's the following results are obtained:

Method	Convergence	# Iterations	Time
Newton	Yes	1	4.05 µs
SDM	No	100	11.00 µs
BFGS	Yes	70	4.05 µs

Table 5: Part 4 Results without backtracking

Method	Convergence	# Iterations	Time
Newton	Yes	558	8.70 µs
SDM	No	100	5.52 µs
BFGS	Yes	497	5.25 µs

Table 6: Part 4 Results with backtracking

As we see from these results, the newton and quasi-newton methods still converge to the optimal portfolio:

x = [-185.74551335, 494.79125851, -372.1543059, -6.20940063, 200.18461618, -40.70871509, -173.04802813, -29.32457964, 6.62083792, 106.59275841]

In this case, some weights are negative because short selling is not restricted in this optimization model.

Iterations typically took longer with this data, since there are more degrees of freedom to optimize, and extra variables that have to meet the stopping criteria. Interestingly, as the dataset increases in size, the BFGS model converges faster than the regular newton method when backtracking is implemented.

A1 Newton Method Code

See https://github.com/jonsmith359/MIE1621-Project for full code

```
def newton (df,d,init,tol=0.001,limit=100, backtrack=False, Rho=0.5, gamma=0.01):
  '''
  Perform Newton Method to find optimal solution for some function f
  df-dataframe containing return and covariance
  d-risk tolerance factor
  init-initial guess
  tol-stopping tolerance between interations
  limit-maximum number of iterations
  backtrack-perform backtrack line search if true, use step length 1 otherwise
  Rho,gamma-backtrack parameters
  '''
  ret,var=split(df)
  ret_kkt, var_kkt = kkt_convert(df)
  x=init
  rho=Rho
  delta=[tol]
  i=0
  print('Initial guess: ',x)
  while (np.any(np.absolute(delta)>=tol))&(i<limit):
    alpha=1
    d=np.matmul(inv(var_kkt),df_kkt(x)) # From definition of newton
    x_k = x - alpha*d
    while (alpha>0.01)&(backtrack==True)&(f(x_k[:-1])>(f(x[:-1])+alpha*gamma*np.matmul(d,df_kkt(x)))):
      alpha=alpha*rho
      x_k = x - alpha*d
      print('alpha: ',alpha)
    x_k = x - alpha*d
    delta = x_k-x
    x=x_k
    i+=1
    print('Iteration',i,':',x)
  print('slope:',df_kkt(x))
  x=x[0:-1]
  print('Solution: x =',x,', Return:',f(x))
```

```
def newton (df,d,init,tol=0.001,limit=100, backtrack=False, Rho=0.5, gamma=0.01):
    '''
    Perform Newton Method to find optimal solution for some function f
    df-dataframe containing return and covariance
    d-risk tolerance factor
    init-initial guess
    tol-stopping tolerance between interations
    limit-maximum number of iterations
    backtrack-perform backtrack line search if true, use step length 1 otherwise
    Rho,gamma-backtrack parameters
    '''
    ret,var=split(df)
    ret_kkt, var_kkt = kkt_convert(df)
    x=init
    rho=Rho
    delta=[tol]
    i=0
    print('Initial guess: ',x)
    while (np.any(np.absolute(delta)>=tol))&(i<limit):
        alpha=1
        d=np.matmul(inv(var_kkt),df_kkt(x)) # From definition of newton
        x_k = x - alpha*d
        while (alpha>0.01)&(backtrack==True)&(f(x_k[:-1])>(f(x[:-1])+alpha*gamma*np.matmul(d,df_kkt(x)))):
            alpha=alpha*rho
            x_k = x - alpha*d
            print('alpha: ',alpha)
        x_k = x - alpha*d
        delta = x_k-x
        x=x_k
        i+=1
        print('Iteration',i,':',x)
    print('slope:',df_kkt(x))
    x=x[0:-1]
    print('Solution: x =',x,', Return:',f(x))
```

A2 SDM Method Code

```
def steepest_descent (df,d,init,tol=0.001,limit=100, backtrack=False, rho=0.5, gamma=0.01):
    '''
    Perform Steepest Descent Method to find optimal solution for some function f
    df-dataframe containing return and covariance
    d-risk tolerance factor
    init-initial guess
    tol-stopping tolerance between interations
    limit-maximum number of iterations
    backtrack-perform backtrack line search if true, use step length 1 otherwise
    Rho,gamma-backtrack parameters
    '''
    ret,var=split(df)
    ret_kkt, var_kkt = kkt_convert(df)
    x=init
    delta=[tol]
    i=0
    print('Initial guess: ',x)
    while (np.any(np.absolute(delta)>=tol))&(np.all(df_kkt(x)<0))&(i<limit):
        alpha=1
        d=df_kkt(x) # From definition of steepest descent
        x_k = x - alpha*d
        while (alpha>0.01)&(backtrack==True)&((f(x_k[:-1]))>(f(x[:-1])+alpha*gamma*np.matmul(d,(df_kkt(x))))):
            alpha=alpha*rho
            x_k = x - alpha*d
            print('alpha: ',alpha)
        x_k = x-alpha*d
        delta = x_k-x
#       print(delta)
        x=x_k
        i+=1
        print('Iteration',i,':',x)
        print('slope:',df_kkt(x))
    print('slope:',df_kkt(x))
    x=x[0:-1]
    print('Solution: x =',x,', Return:',f(x))
```


```
def steepest_descent (df,d,init,tol=0.001,limit=100, backtrack=False, rho=0.5, gamma=0.01):
    '''
    Perform Steepest Descent Method to find optimal solution for some function f
    df-dataframe containing return and covariance
    d-risk tolerance factor
    init-initial guess
    tol-stopping tolerance between interations
    limit-maximum number of iterations
    backtrack-perform backtrack line search if true, use step length 1 otherwise
    Rho,gamma-backtrack parameters
    '''
    ret,var=split(df)
    ret_kkt, var_kkt = kkt_convert(df)
    x=init
    delta=[tol]
    i=0
    print('Initial guess: ',x)
    while (np.any(np.absolute(delta)>=tol))&(np.all(df_kkt(x)<0))&(i<limit):
        alpha=1
        d=df_kkt(x) # From definition of steepest descent
        x_k = x - alpha*d
        while (alpha>0.01)&(backtrack==True)&((f(x_k[:-1]))>(f(x[:-1])+alpha*gamma*np.matmul(d,(df_kkt(x))))):
            alpha=alpha*rho
            x_k = x - alpha*d
            print('alpha: ',alpha)
        x_k = x-alpha*d
        delta = x_k-x
        x=x_k
        i+=1
        print('Iteration',i,':',x)
        print('slope:',df_kkt(x))
    print('slope:',df_kkt(x))
    x=x[0:-1]
    print('Solution: x =',x,', Return:',f(x))
```

A3 BFGS Method Code

```
def BFGS (df,d,init,tol=0.001,limit=100, backtrack=False, rho=0.5, gamma=0.0001):
    '''
    Perform BFGS Quasi-Newton Method to find optimal solution for some function f
    '''
    ret,var=split(df)
    ret_kkt, var_kkt = kkt_convert(df)
    x=init
    H=inv(np.eye(1+len(var)))
    I=np.eye(1+len(var))
    delta=[tol]
    i=0
    print('Initial guess: ',x)
    while (np.any(np.absolute(delta)>=tol))&(i<limit):
        alpha=1
        d=np.matmul(np.asarray(H),df_kkt(x))
        x_k = x - alpha*d
        while (alpha>0.1)&(backtrack==True)&(f(x_k[:-1])>(f(x[:-1])+alpha*gamma*np.matmul(d,df_kkt(x)))):
            alpha=alpha*rho
            x_k = x - alpha*d
            print('alpha: ',alpha)
        x_k = x - alpha*d
        delta = x_k-x
#       find udpated H inverse
        s=np.asmatrix(x_k-x)
        y=np.asmatrix(df_kkt(x_k)-df_kkt(x))
        a=np.asscalar(1/(y*np.transpose(s)))
        A=a*(np.transpose(s)*y)
        B=a*(np.transpose(y)*s)
        C=a*(np.transpose(s)*(s))
        H=np.matmul(np.matmul((I-A),H),(I-B))+C
        x=x_k
        i+=1
        print('Iteration',i,':',x)
    print('slope:',df_kkt(x))
    x=x[0:-1]
    print('Solution: x =',x,', Return:',f(x))
```


```
def BFGS (df,d,init,tol=0.001,limit=100, backtrack=False, rho=0.5, gamma=0.0001):
    '''
    Perform BFGS Quasi-Newton Method to find optimal solution for some function f
    '''
    ret,var=split(df)
    ret_kkt, var_kkt = kkt_convert(df)
    x=init
    H=inv(np.eye(1+len(var)))
    I=np.eye(1+len(var))
    delta=[tol]
    i=0
    print('Initial guess: ',x)
    while (np.any(np.absolute(delta)>=tol))&(i<limit):
        alpha=1
        d=np.matmul(np.asarray(H),df_kkt(x))
        x_k = x - alpha*d
        while (alpha>0.1)&(backtrack==True)&(f(x_k[:-1])>(f(x[:-1])+alpha*gamma*np.matmul(d,df_kkt(x)))):
            alpha=alpha*rho
            x_k = x - alpha*d
            print('alpha: ',alpha)
        x_k = x - alpha*d
        delta = x_k-x
#         find udpated H inverse
        s=np.asmatrix(x_k-x)
        y=np.asmatrix(df_kkt(x_k)-df_kkt(x))
        a=np.asscalar(1/(y*np.transpose(s)))
        A=a*(np.transpose(s)*y)
        B=a*(np.transpose(y)*s)
        C=a*(np.transpose(s)*(s))
        H=np.matmul(np.matmul((I-A),H),(I-B))+C
        x=x_k
        i+=1
        print('Iteration',i,':',x)
    print('slope:',df_kkt(x))
    x=x[0:-1]
    print('Solution: x =',x,', Return:',f(x))
```