

SDS 385: Exercise 01

August 27, 2016

Jennifer Starling

Linear Regression

Part A

The WLS objective function, rewritten in matrix form, is:

$$\hat{\beta} = \arg \min_{\beta \in R^P} \frac{1}{2} (Y - X'\beta)'W(Y - X'\beta) = \arg \min_{\beta \in R^P} \frac{1}{2} (Y' - X'\beta')W(Y - X'\beta)$$

To satisfy the 'arg min' part of the expression, take the derivative of $\hat{\beta}$ with respect to β , set equal to zero, and solve as follows.

$$\frac{\delta}{\delta\beta} [\frac{1}{2} (Y' - X'\beta')W(Y - X'\beta)] = (\frac{1}{2}) \frac{\delta}{\delta\beta} [(Y'WY - 2Y'WX\beta + \beta'XWX'\beta)] = 0$$

The derivatives of each term are as follows.

- (a) $\frac{\delta}{\delta\beta} [Y'WY] = 0$ since this term is constant with respect to β .
- (b) $\frac{\delta}{\delta\beta} [-2Y'WX\beta] = -2Y'WX$, since derivative has form $\frac{\delta}{\delta\beta} c\beta = c$, where $c = -2Y'WX$. Then since W diagonal (and so symmetric) and X and Y vectors, $-2Y'WX = -2X'WY$.
- (c) $\frac{\delta}{\delta\beta} [\beta'XWX'\beta] = 2XWX'\beta$, since derivative has quadratic form $\frac{\delta}{\delta\beta} [\beta'V\beta] = (V + V')\beta$, where $V = XWX'$. When V symmetric, this further simplifies to $2V\beta$.

Then the derivative, subbing in $\hat{\beta}$ for β , is

$$\frac{1}{2} [-2X'WY + 2XWX'\hat{\beta}] = 0 \rightarrow XWX'\hat{\beta} = X'WY \rightarrow \hat{\beta} = (XWX')^{-1}X'WY.$$

Therefore $\hat{\beta} = (XWX')^{-1}X'WY$.

To show that $\hat{\beta} = (XWX')^{-1}X'WY$ is the solution to the linear system:

$$(X'WX)\hat{\beta} = (X'WX)(X'WX)^{-1}X'WY = IX'WY = X'WY.$$

Part B

Numerically speaking, I do not believe that inversion is the fastest and most stable way to solve the linear system. There are several matrix factorization methods which provide more stability and are computationally efficient compared to inversion. Inverting a matrix directly is computationally intensive, especially as N and P become large.

Some of the methods I discovered for solving linear equations of form $Ax=B$ without inverting the A matrix directly are: LU Decomposition, Gaussian elimination, Cholesky decomposition, QR decomposition, RRQR factorization, and the conjugate gradient method. There was not a strict consensus as to which method is universally superior; the key to know the characteristics of the matrix A, so that you can choose an optimal method. Different characteristics lend themselves to different methods.

- (a) Cholesky performs well for Hermitian matrices (symmetric positive definite).
- (b) LU performs well when A is sparse, and A is only required to be square.
- (c) Conjugate gradient requires A to be symmetric positive definite, but is a good iterative algorithm for scenarios where A is sparse and too large to be inverted directly or for Cholesky.

My method will be the Cholesky decomposition. My pseudo-code is as follows.

Goal: Solve $Ax = b$ where $A = X'WX$, $b = X'WY$, and x is the vector of β coefficient estimates.

Function inputs:

- (a) X , an $N \times P$ matrix
- (b) W , a diagonal matrix of weights
- (c) Y , an $N \times 1$ vector of responses

Function outputs:

- (a) $\hat{\beta}$, a vector of coefficient estimates

Code Steps:

- (1) Set $A = X'WX$
- (2) Set $b = X'WY$
- (3) Set R = Cholesky decomposition of A . (R gives the R (upper) instead of L (lower).)

Now we have $R'R = A$.

- (4) Solve $R'z = b$ for z by finding $z = (R')^{-1}b$
- (4) Solve $Rx = z$ by finding $x = R^{-1}z$
- (4) Return $\hat{\beta}$ estimate as $\hat{\beta} = x$

I also included the LU decomposition, which is solved by similar steps.

Part C

The R code for implementing and benchmarking the functions is as follows.

```

5  ### SDS 385 - Exercises 01 - Part A
   #This code compares various matrix decomposition
   #methods to the inversion method, and benchmarks
   #performance of the Cholesky and LU methods versus
   #inversion at various sample/parameter sizes,
   #and various sparsity levels of the X matrix.

   #Jennifer Starling
   #22 August 2016

10  library(Matrix) #For matrix decomposition.

   ### PART C:

15  #-----
   #Inversion Method function:
   #   Inputs: X = vector of x values, Y = vector of y values,
   #           W = diag matrix of weights.

```

```

#   Outputs: B = beta-hat vector; the WLS solution for
20 #       estimating the beta vector of coefficients.

#   Matrix requirements:
#   1. Length X = Length Y = Dim(W)
#   2. t(X) %*% W %*% X must be invertible
25
inv_method <- function(X,W,y){
  B_hat <- solve(t(X) %*% W %*% X) %*% t(X) %*% W %*% y
  return(B_hat)
}

30
#-----
#LU Decomposition to solve linear system Ax=b.
#   Inputs: X = vector of x values, Y = vector of y values,
#           W = diag matrix of weights.
35 #   Outputs: B_hat_LU, an estimate of the 'x' in Ax=b.

lu_method <- function(X,W,y){

  #Solves linear system Ax=b.
40 #Since we have (X'WX)B=X'Wy, B (beta) acts as x, with A and b as follows.

  #Finding B (beta_hat) in equation
  A = (t(X) * diag(W)) %*% X #Efficient way of A = t(X) %*% W %*% X as W diag.
                                #Avoids mult by 0's.
45 b = (t(X) * diag(W)) %*% y #b'Wy

  #Obtain LU matrix decomposition of A.
  decomp <- lu(A) #Calculates matrix decomposition object..
  L <- expand(decomp)$L #Upper triangular matrix
50 U <- expand(decomp)$U #Lower triangular lower triangular matrix
  #Note: Uses partial pivoting. $P shows pivot matrix.

  #Now we replace Ax=b with LUX=b.
  #Introduce Ld=b, giving us two linear equation systems: Ld=b and Ux=d.
55 #So we will solve in two steps.

  #1. Solve Ld=b for d. This is d=inv(L)b
  d <- solve(L) %*% b

  #2. Substitutde d into Ux=d to solve for x. This is x = inv(U)d. (x=beta_hat)
60 B_hat_LU <- solve(U) %*% d

  return(B_hat_LU) #Returns function output.
}

65
#-----
#Cholesky decomposition function:
#   Inputs: X = vector of x values, Y = vector of y values,
#           W = diag matrix of weights.
70 #   Outputs: B = beta-hat vector; the WLS solution for
#       estimating the beta vector of coefficients.

```

```

cholesky_method <- function(X,W,y){
  #Solves linear system Ax=b.
  #Since we have  $(X'WX)B=X'Wy$ , B (beta) acts as x, with A and b as follows.

  #Finding B (beta_hat) in equation
  A = (t(X) * diag(W)) %*% X #Efficient way of  $A = t(X) \%*\% W \%*\% X$  as W diag.
                                #Avoids mult by 0's.
  b = (t(X) * diag(W)) %*% y #b'Wy
  R <- chol(A) #Find right/upper cholesky decomposition of A.

  #Now we have  $R'R=A$ .

  #1. Solve  $R'z=b$  for z. This is  $z = \text{inv}(R')b$ .
  z = solve(t(R)) %*% b

  #2. Solve  $Rx=z$  for x. This is  $x = \text{inv}(R)z$ . ( $x = \text{beta\_hat}$ )
  B_hat_chol <- solve(R) %*% z

  return(B_hat_chol)
}

#-----
#BENCHMARKING:

#Simulate data from the linear model for a range of values of N and P.
#(Assume weights are all 1, data are gaussian.)
#Carry out performance testing of the two methods.

library(microbenchmark)

N <- c(10,100,500,1000)
P <- N/2 #Setting up so that  $N > P$ . This is an arbitrary choice.

perf_results <- list()

for (i in 1:length(N)){
  n <- N[i]
  p <- P[i]

  print(n)

  #Set up matrices of size N, P parameters: (dummy data)
  X <- matrix(rnorm(n*p),nrow=n,ncol=p)
  y <- rnorm(n)
  W <- diag(1,nrow=n)

  #Perform benchmarking:
  perf_results[[i]] <- microbenchmark(
    inv_method(X,W,y),
    lu_method(X,W,y),
    cholesky_method(X,W,y), unit='ms'
  )
}

```

125

```

    )
  }

names(perf_results) <- (c('N=10,P=5', 'N=100,P=50', 'N=500,P=250', 'N=1500,P=500'))
perf_results      #Display benchmarking results.

```

The performance benchmarking results are as follows. The inverse method was fastest for very small N and P values, but as N and P increased, LU and Cholesky performed more quickly than inverse. LU was the fastest method of the three.

```

> perf_results
$`N=10,P=5`
Unit: milliseconds
      expr      min      lq      mean      median      uq      max neval cld
inv_method(X, W, y) 0.038848 0.0442330 0.1654871 0.0482625 0.053931 10.973424 100 a
lu_method(X, W, y) 0.097958 0.1094925 0.1618572 0.1168010 0.134071 2.297167 100 a
cholesky_method(X, W, y) 0.089010 0.1019180 0.1291046 0.1108950 0.122683 0.445684 100 a

$`N=100,P=50`
Unit: milliseconds
      expr      min      lq      mean      median      uq      max neval cld
inv_method(X, W, y) 1.341858 1.3730145 1.5708283 1.464907 1.6235910 3.893104 100 c
lu_method(X, W, y) 0.389776 0.4115185 0.5163842 0.443505 0.5392765 2.367135 100 a
cholesky_method(X, W, y) 0.621448 0.6396920 0.8082234 0.709129 0.7825670 3.414895 100 b

$`N=500,P=250`
Unit: milliseconds
      expr      min      lq      mean      median      uq      max neval cld
inv_method(X, W, y) 138.48695 143.65306 147.27360 145.88347 148.95680 236.5789 100 c
lu_method(X, W, y) 27.08385 28.66972 31.97459 29.76773 31.06965 123.7989 100 a
cholesky_method(X, W, y) 52.35322 54.36095 56.99425 55.57933 57.59481 143.0048 100 b

$`N=1500,P=500`
Unit: milliseconds
      expr      min      lq      mean      median      uq      max neval cld
inv_method(X, W, y) 1159.7687 1181.4272 1204.1996 1192.2028 1210.0858 1318.4425 100 c
lu_method(X, W, y) 217.5088 226.2489 236.3857 231.3792 237.0129 366.1290 100 a
cholesky_method(X, W, y) 425.7139 435.8642 450.1871 441.6242 450.4200 561.5078 100 b

```

Figure 1: Performance benchmarking with dense X matrix

Part D

Since both LU and Cholesky are good for sparse matrices, I benchmarked both of these methods against the inverse method for a sparse matrix X.

I performed two types of benchmarking:

- Benchmarking various N and P at 10% sparse.
- Benchmarking various sparsity levels (5%, 10%, 20%, 50%).

For benchmarking at various N and P levels, results were similar to the results above. Inverse was superior for small N and P, and as N and P increased, Cholesky and LU were superior. LU again performed the most efficiently.

For benchmarking at various sparsity levels (with N=100, P=50 for all levels), the LU method performed most efficiently again, followed by Cholesky. For all methods, performance slowed as the matrix became less sparse.

```
> perf_results #Display benchmarking results.
$`N=10,P=5`
Unit: microseconds
      expr      min      lq      mean      median      uq      max neval cld
inv_method(X, W, y) 195.270 230.6890 250.7306 246.817 263.0400 436.975 100 ab
lu_method(X, W, y) 142.961 185.6895 233.2497 208.937 236.2485 794.388 100 a
cholesky_method(X, W, y) 184.360 223.1060 269.7928 244.267 278.2675 708.212 100 b

$`N=100,P=50`
Unit: microseconds
      expr      min      lq      mean      median      uq      max neval cld
inv_method(X, W, y) 1340.657 1362.3990 1461.4733 1452.5095 1491.339 2091.966 100 c
lu_method(X, W, y) 393.144 429.8705 557.8016 506.2400 530.219 5921.085 100 a
cholesky_method(X, W, y) 631.044 655.0500 754.6730 739.4485 773.096 2299.500 100 b

$`N=500,P=250`
Unit: milliseconds
      expr      min      lq      mean      median      uq      max neval cld
inv_method(X, W, y) 139.28952 147.35422 152.22056 150.50908 153.99406 254.34373 100 c
lu_method(X, W, y) 27.87053 30.68015 32.11990 31.90189 33.13519 39.19118 100 a
cholesky_method(X, W, y) 52.88206 56.24804 58.58854 58.02186 60.73400 71.86568 100 b

$`N=1000,P=500`
Unit: milliseconds
      expr      min      lq      mean      median      uq      max neval cld
inv_method(X, W, y) 1140.2651 1206.4823 1407.2283 1266.9112 1527.1099 2560.0583 100 c
lu_method(X, W, y) 220.1672 234.8487 299.8574 251.6809 300.5480 782.2592 100 a
cholesky_method(X, W, y) 419.3958 446.5509 497.6972 469.8177 524.3121 883.4575 100 b
```

Figure 2: Performance benchmarking with sparse X matrix

```
> perf_results
$`5%`
Unit: microseconds
      expr      min      lq      mean      median      uq      max neval cld
inv_method(X, W, y) 1339.781 1359.7520 1469.3973 1383.2195 1479.3050 2624.806 100 c
lu_method(X, W, y) 393.088 421.8870 464.1313 435.1490 460.1350 1081.836 100 a
cholesky_method(X, W, y) 654.567 679.9915 803.9488 693.1765 744.2205 6065.418 100 b

$`10%`
Unit: microseconds
      expr      min      lq      mean      median      uq      max neval cld
inv_method(X, W, y) 1341.014 1358.3260 1452.5574 1388.2360 1483.1140 2098.072 100 c
lu_method(X, W, y) 388.911 411.5860 465.5492 430.5955 463.8015 1295.116 100 a
cholesky_method(X, W, y) 618.995 637.5325 744.7535 648.2635 701.9220 6531.208 100 b

$`25%`
Unit: microseconds
      expr      min      lq      mean      median      uq      max neval cld
inv_method(X, W, y) 1341.192 1361.8625 2424.2772 1384.4855 1446.653 101512.188 100 b
lu_method(X, W, y) 393.414 410.0670 474.5369 425.5910 473.378 1209.219 100 a
cholesky_method(X, W, y) 622.194 638.1095 707.9242 647.6305 694.926 1448.988 100 ab

$`50%`
Unit: microseconds
      expr      min      lq      mean      median      uq      max neval cld
inv_method(X, W, y) 1342.665 1357.659 1432.2028 1368.9885 1467.3945 2316.136 100 c
lu_method(X, W, y) 389.675 414.527 511.9470 432.1205 466.6205 6331.394 100 a
cholesky_method(X, W, y) 619.193 638.728 685.6965 657.8495 705.1420 1335.383 100 b
```

Figure 3: Performance benchmarking with varying sparsity levels of X

Generalized Linear Regression

Part A

The negative log-likelihood function is simplified as follows.

$$\begin{aligned} l(\beta) &= -\log\{\prod_{i=1}^N p(y_i|\beta)\} = -\log\{\prod_{i=1}^N \binom{m_i}{y_i} w_i^{y_i} (1-w_i)^{(m_i-y_i)}\} \\ &= -\sum_{i=1}^N \log\{w_i^{y_i} (1-w_i)^{(m_i-y_i)}\} = -\sum_{i=1}^N \{y_i \log(w_i) + (m_i - y_i) \log(1-w_i)\}; w_i = \frac{1}{1+\exp(x'_i \beta)} \end{aligned}$$

The gradient is found by taking the derivative of $l(\beta)$ with respect to β .

First, the derivative of w_i with respect to β will be useful:

$$\frac{\delta w_i}{\delta \beta} = -(1 + \exp(-x'_i \beta))^{-2} \cdot \frac{\delta}{\delta \beta} (\exp(-x'_i \beta)) = \frac{-\exp(-x'_i \beta)(-x'_i)}{(1+\exp(-x'_i \beta))^2} = \frac{x'_i \exp(-x'_i \beta)}{(1+\exp(-x'_i \beta))^2}$$

Then find the gradient:

$$\begin{aligned} \frac{\delta l(\beta)}{\delta \beta} &= -\sum_{i=1}^N \{y_i \frac{1}{w_i} (\frac{\delta w_i}{\delta \beta}) + (m_i - y_i) \frac{1}{1-w_i} (-\frac{\delta w_i}{\delta \beta})\} = -\sum_{i=1}^N \{y_i \frac{1}{w_i} (\frac{x_i \exp(-x'_i \beta)}{1+\exp(-x'_i \beta)}) + (m_i - y_i) \frac{1}{1-w_i} (\frac{-x_i \exp(-x'_i \beta)}{(1+\exp(-x'_i \beta))^2})\} \\ &= -\sum_{i=1}^N \{y_i \frac{1}{w_i} (w_i^2 x_i \exp(-x'_i \beta)) + (m_i - y_i) \frac{1}{1-w_i} (w_i^2 x_i \exp(-x'_i \beta))\} \\ &= -\sum_{i=1}^N \{y_i w_i \exp(-x'_i \beta) - (m_i - y_i) x_i w_i\} = -\sum_{i=1}^N \{(y_i w_i \exp(-x'_i \beta) - (m_i - y_i) w_i) x_i\} \end{aligned}$$

And since $w_i = \frac{1}{1+\exp(-x'_i \beta)} \rightarrow \exp(-x'_i \beta) = \frac{1}{w_i} - 1$, we can simplify further:

$$\nabla l(\beta) = -\sum_{i=1}^N \{(y_i w_i (\frac{1}{w_i} - 1) - m_i w_i + w_i y_i) x_i\} = -\sum_{i=1}^N (y_i - m_i w_i) x_i$$

In matrix form: $\nabla l(\beta) = -X'(y - mw) = X'(mw - y)$

Part B

The following is my gradient descent code and results. A few notes regarding methodology:

- (a) Step size is fixed at $a = .01$ in this code.
- (b) To handle probabilities close to 1 and 0, .01 is added to each log term in the loglikelihood.
- (c) Intercept was handled by adding a column of 1's to the X matrix.
- (d) Convergence was determined using $||\nabla l(\beta)|| < 1 * 10^{-2}$.

R output results were as follows.

```
[1] "Algorithm has converged."
[1] 20311
>
> #Post-processing steps.
5 > beta_gd <- betas[[iter]] #Save and output estimated beta values.
> beta_gd
              V3              V4              V5              V6              V7              V8
V9
 0.43486932 -5.03700599  1.65564614 -3.63350426 13.54124216  1.05599002  0.02689289
0.69689744
              V10              V11              V12
10 2.61356777  0.44381978 -0.48663758
> beta #Output glm beta values for comparison.
              X              XV3              XV4              XV5              XV6              XV7              XV8
XV9
 0.48701675 -7.22185053  1.65475615 -1.73763027 14.00484560  1.07495329 -0.07723455
0.67512313
              XV10              XV11              XV12
15 2.59287426  0.44625631 -0.48248420
```

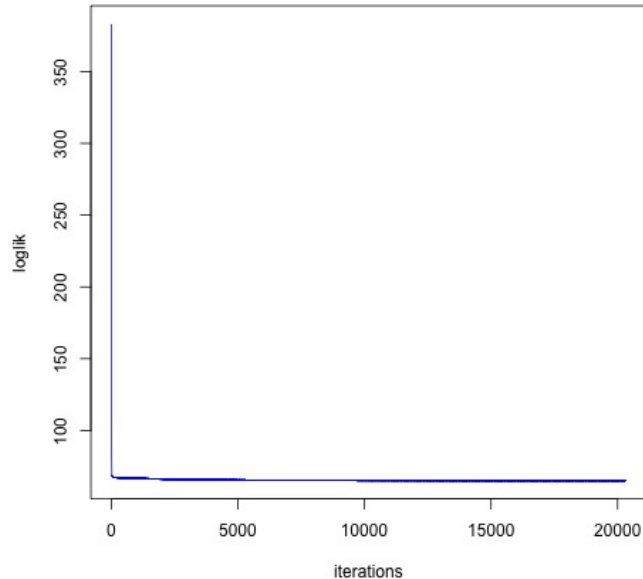


Figure 4: Log-likelihood function

The R code is as follows.

```

### SDS 385 – Exercises 01 – Part B – Problem B
#This code implements gradient descent to estimate the
#beta coefficients for binomial logistic regression.

5 #Jennifer Starling
  #26 August 2016

#PART B:

10 #Read in code.
wdbc = read.csv('/Users/jennstarling/UTAustin/2016_Fall_SDS 385_Stats Models for Big Data/Course
y = wdbc[,2]

15 #Convert y values to 1/0's.
Y = rep(0,length(y)); Y[y=='M']=1
X = as.matrix(wdbc[, -c(1,2)])

#Select features to keep, and scale features.
20 scrub = which(1:ncol(X) %% 3 == 0)
scrub = 11:30
X = X[, -scrub]
X <- scale(X) #Normalize design matrix features.
X = cbind(rep(1,nrow(X)),X)

25 #Set up vector of sample sizes. (All 1 for wdbc data.)
m <- rep(1,nrow(X))

```

```

#-----
30 #Binomial Negative Loglikelihood function.
    #Inputs: Design matrix X, vector of 1/0 vals Y,
    # coefficient matrix beta, sample size vector m.
    #Output: Returns value of negative log-likelihood
    # function for binomial logistic regression.
35 logl <- function(X,Y,beta,m){
    w <- 1 / (1 + exp(-X %*% beta)) #Calculate probabilities vector w_i.
    logl <- - sum(Y*log(w+.01) + (m-Y)*log(1-w+.01)) #Calculate log-likelihood.
    #Adding .01 to resolve issues with probabilities near 0 or 1.
    return(logl)
40 }

#-----
#Function for calculating Euclidean norm of a vector.
norm_vec <- function(x) sqrt(sum(x^2))
45

#-----
#Gradient Function:
    #Inputs: Design matrix X, vector of 1/0 vals Y,
    # coefficient matrix beta, sample size vector m.
50 #Output: Returns value of gradient function for binomial
    # logistic regression.

gradient <- function(X,Y,beta,m){
    w <- 1 / (1 + exp(-X %*% beta)) #SCalculate probabilities vector w_i.
55
    gradient <- array(NA,dim=length(beta)) #Initialize the gradient.
    gradient <- -apply(X*as.numeric(Y-m*w),2,sum) #Calculate the gradient.

    return(gradient)
60 }

#-----
#Gradient Descent Algorithm:

65 #1. Fit glm model for comparison. (No intercept: already added to X.)
glm1 = glm(y~X-1, family='binomial') #Fits model, obtains beta values.
beta <- glm1$coefficients

loglik <- 0 #Initialize vector to hold loglikelihood function.
70 grad <- list() #Initialize list to hold gradients for each iteration.
maxiter <- 50000 #Specify max iterations allowed.
betas <- list() #Initialize list to hold beta vector for each iteration.

a <- .01 #Set step size.
75 iter <- 0 #Track number of iterations until convergence..
conv <- 1*10^-2 #Set convergence level.

betas[[1]] <- rep(0,ncol(X)) #Initialize beta vector to 0 to start.
80 #2. Perform gradient descent.

```

```
for (i in 1:maxiter){  
  
  #Calculate loglikelihood for each iteration.  
  loglik[i] <- logl(X,Y,betas[[i]],m)  
85  
  
  #Calculate gradient for beta.  
  grad[[i]] <- gradient(X,Y,betas[[i]],m)  
  
  #Set new beta equal to beta - a*gradient(beta).  
90  betas[[i+1]] <- betas[[i]] - a * grad[[i]]  
  
  iter <- i + 1  #Track iterations.  
  
  #Check if convergence met: If yes, exit loop.  
95  if (norm_vec(grad[[i]]) < conv){  
    print('Algorithm has converged.')  
    print(iter)  
    break;  
  }  
100  
  
  #Check if max iterations met: If yes, exit loop.  
  if (i >= maxiter){  
    print('Algorithm ending without convergence; max iterations reached.')  
    break;  
105  }  
}  
#End gradient descent iterations.  
  
#Post-processing steps.  
beta_gd <- betas[[iter]]  #Save and output estimated beta values.  
110 beta_gd  
beta  #Output glm beta values for comparison.  
  
#Plot log-likelihood function for convergence.  
plot(1:i,loglik,type='l',xlab='iterations',col='blue')  
115  
#Save plot.  
jpeg(file='/Users/jennstarling/UTAustin/2016_Fall_SDS 385_Stats Models for Big Data/385_Exercise  
plot(1:i,loglik,type='l',xlab='iterations',col='blue')  
dev.off()
```