Linear Regression

(A) Rewrite the WLS objective functions in terms of vectors and matrices as follows:

$$\hat{\beta}=\arg\min_{\beta\in R^P}\sum_{i=1}^{N}\frac{w_i}{2}(y_i-x_i^T\beta)^2=\arg\frac{1}{2}\min_{\beta\in R^P}(y-X\beta)^T W(y-X\beta) \tag{1}$$

Since W is a diagonal matrix of weights (so symmetric), then

$$\begin{aligned}&(y-X\beta)^T W(y-X\beta)\\&=(y^T-\beta^T X^T)W(y-X\beta)\\&=y^T Wy-y^T X\beta-\beta^T X^T Wy+\beta^T X^T WX\beta\\&=y^T Wy-(X^T Wy)^T\beta-\beta^T X^T Wy+(X\beta)^T WX\beta\end{aligned}$$

To get the minimum of the above equation, take derivative of β with the following formulas

$$\frac{\partial x^T a}{\partial x}=\frac{\partial a^T x}{\partial x}=a,\frac{\partial x^T bx}{\partial x}=(b+b^T)x$$

$$\Rightarrow\frac{\partial[\ (y-X\beta)^T W(y-X\beta)]}{\partial\beta}=-2X^T Wy+2X^T WX\beta=0$$

$$\Rightarrow(X^T WX)\hat{\beta}=X^T Wy$$

(B) Numerically speaking, I do not think inversion method is the fastest and most stable way to solve the linear system. Computing and applying the inverse matrix or pseudoinverse is a tremendously bad idea, since it is much more expensive and often numerically less stable than applying other algorithms, especially for high dimensional matrices.

Here are several ways to solve a system of linear equations Ax = b which provide more stability and are computationally efficient compared to inversion method.

(a) Matrix decomposition: Gaussian or Gauss-Jordan elimination (considered as LU decomposition), Cholesky decomposition, QR decomposition, or SVD, and
(b) Iterative method: conjugate gradient method.

Which method is optimal depends on the size and properties of the system matrix.

(a) LU requires A to be square and performs well when A is sparse; it can be used for most linear systems.
(b) Cholesky performs well for ***Hermitian positive-definite matrix*** A (A=LL*, where L is a lower triangular matrix with real and positive diagonal entries, and L* denotes the conjugate transpose of L)
(c) QR decomposition requires A has linearly independent columns.
(d) Conjugate gradient requires A to be symmetric and positive definite; it performs well when

A is sparse and too large to be inverted directly for Cholesky.

(C) Since $X^TWX = X^T(W^{1/2})^T W^{1/2} X \overset{set\ W^{1/2}=V}{=} (VX)^T VX$ is positive-definite, my method will be the Cholesky decomposition. To solve $(X^TWX)\hat{\beta} = X^TWy$, here is the pseudo-code:

Function inputs:
X: N x P matrix
Y: N x 1 vector of responses
W: N x N diagonal matrix of weights

Function outputs:
$\hat{\beta}$: P x 1 vector of coefficient estimates

Pseudo-code:
1) Set $A = X^TWX$;
2) Set $b = X^TWy$;
3) Set U = Cholesky decomposition of A. Only the upper triangular part of A is used in R so that $A = U^TU$ when A is symmetric.
4) Solve $U^Tz = b \Rightarrow z = (U^T)^{-1}b$;
5) Solve $Ux = z \Rightarrow x = U^{-1}z$;
6) Return $\hat{\beta}$ as $\hat{\beta} = x$.

From the following R output, the inverse method is fastest for very small N and P values, but as N and P increase, LU and Cholesky methods perform much more quickly than inverse. LU is the most efficient method of the three and it is a partial Gaussian elimination method.

```
$`N=20, P=5`
Unit: milliseconds
                expr      min        lq      mean    median       uq       max neval
  inv_method(X, W, y) 0.081536 0.0849575 0.1931034 0.0892335 0.091799  5.458343   100
   lu_method(X, W, y) 0.177897 0.1818880 0.3844447 0.1898705 0.195287 18.898650   100
 chol_method(X, W, y) 0.172195 0.1778970 0.3056914 0.1830280 0.193292 10.990810   100

$`N=100, P=25`
Unit: milliseconds
                expr      min        lq      mean   median       uq      max neval
  inv_method(X, W, y) 0.625489 0.6323310 0.6990878 0.679086 0.689349 2.409014   100
   lu_method(X, W, y) 0.271977 0.2845205 0.3290688 0.315025 0.346385 0.679086   100
 chol_method(X, W, y) 0.319301 0.3355515 0.3833101 0.367482 0.380596 2.109670   100

$`N=400, P=100`
Unit: milliseconds
                expr       min        lq      mean    median        uq        max neval
  inv_method(X, W, y) 30.104417 30.618435 33.188846 31.425812 32.949621 157.426978   100
   lu_method(X, W, y)  4.234735  4.344780  4.814803  4.610484  4.897285   7.127263   100
 chol_method(X, W, y)  5.469176  5.583782  6.206546  5.826680  6.420238   8.372538   100

$`N=1200, P=300`
Unit: milliseconds
                expr       min       lq     mean   median       uq       max neval
  inv_method(X, W, y) 771.87970 794.7408 885.7643 819.2258 878.1566 1739.3503   100
   lu_method(X, W, y)  97.14744 100.9289 122.3051 104.6884 113.5510  281.4505   100
 chol_method(X, W, y) 122.67388 128.7483 148.1157 133.4865 140.8815  322.5651   100
```

Figure 1: Performance benchmarking results

(D) Since both LU and Cholesky perform well for sparse matrices, we benchmarked both of these methods and sparse Cholesky decomposition against inverse method for a sparse matrix with various sparsity level (1%, 5%, 25%). Here, *theta* represents the density of X, i.e., the proportion of entries which are non-zero. In the benchmark below, we can see more noticeable efficiency increase with higher sparsity and LU again performed the most efficiently.

```
$`theta=0.01`
Unit: milliseconds
                    expr        min         lq      mean    median        uq       max neval
    inv_method(X, W, y) 12853.4133 14131.1496 15649.875 16316.100 16981.361 17780.898    10
     lu_method(X, W, y)  2746.3071  2832.3491  3188.309  3182.086  3551.914  3700.431    10
   chol_method(X, W, y)  3326.3916  3956.6246  4783.463  4965.668  5419.242  6048.513    10
 sparse_method(X, W, y)   682.8665   822.4416  1046.696  1036.324  1190.962  1570.341    10

$`theta=0.05`
Unit: milliseconds
                    expr        min         lq       mean     median        uq       max neval
    inv_method(X, W, y) 11427.7201 14825.9745 15919.8157 16308.1549 17487.184 19604.344    10
     lu_method(X, W, y)  3183.9826  3483.1195  3573.6130  3548.6364  3695.268  3954.929    10
   chol_method(X, W, y)  3827.7877  4200.4905  4823.9148  4891.4851  5404.459  5574.472    10
 sparse_method(X, W, y)   805.8619   826.8298   977.1668   967.8372  1074.542  1215.062    10

$`theta=0.25`
Unit: milliseconds
                    expr        min         lq      mean    median        uq       max neval
    inv_method(X, W, y) 13071.8719 15303.1995 15945.594 16223.690 16977.776 17819.406    10
     lu_method(X, W, y)  3014.7278  3067.4558  3395.305  3287.320  3604.779  4155.951    10
   chol_method(X, W, y)  4581.1142  4645.9313  5255.740  5144.875  5574.287  6454.916    10
 sparse_method(X, W, y)   697.6045   951.5397  1046.293  1091.212  1146.796  1240.047    10
```

Figure 2: Benchmarking for various values of N, P, and density level

Generalized Linear Regression

(A) The negative log likelihood is

$$l(\beta) = -\log\{\prod_{i=1}^{N} p(y_i \mid \beta)\} = -\log\{\prod_{i=1}^{N} \binom{m_i}{y_i} w_i^{y_i} (1-w_i)^{m_i - y_i}\}$$

$$= -\sum_{i=1}^{N} \log\{\binom{m_i}{y_i} w_i^{y_i} (1-w_i)^{m_i - y_i}\}$$

$$= -\sum_{i=1}^{N} \{\log\binom{m_i}{y_i} + y_i \log(w_i) + (m_i - y_i)\log(1-w_i)\}$$

With $w_i = \dfrac{1}{1+\exp(-x_i^T \beta)} \Rightarrow \exp(-x_i^T \beta) = \dfrac{1}{w_i} - 1$

Taking derivative of $l(\beta)$ w.r.t β :

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

Since $\dfrac{\partial w_i}{\partial \beta} = -(1+\exp(-x_i^T\beta))^{-2} \dfrac{\partial}{\partial \beta}\exp(-x_i^T\beta)$

$$= \frac{-\exp(-x_i^T\beta)(-x_i)}{(1+\exp(-x_i^T\beta))^2} = \frac{x_i \exp(-x_i^T\beta)}{(1+\exp(-x_i^T\beta))^2} = x_i w_i^2 \left(\frac{1}{w_i} - 1\right) = x_i w_i (1-w_i)$$

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

$$= -\sum_{i=1}^{N} (y_i x_i (1-w_i) - (m_i - y_i) w_i x_i) = -\sum_{i=1}^{N} (y_i - m_i w_i) x_i$$

$$\overset{\text{In matrix form}}{=} -X^T (y - mw) = X^T (mw - y)$$

(B) This is a one trial case (sample size is 1), then the log likelihood function is

$$l(\beta) = -\log[\prod_{i=1}^{N} p(y_i \mid \beta)] \overset{m_i = 1}{=} -\log[\prod_{i=1}^{N} w_i^{y_i} (1-w_i)^{m_i - y_i}]$$

$$= -\sum_{i=1}^{N} \log[w_i^{y_i} (1-w_i)^{m_i - y_i}]$$

$$= -\sum_{i=1}^{N} [y_i \log(w_i) + (m_i - y_i)\log(1-w_i)]$$

Here are a few notes about the R code:

(a) Step size is fixed at stepsize = 0.01;

(b) To handle probabilities close to 0 or 1, and a constant 10^{-5} to each log term in the log likelihood function;

(c) Convergence is determined by using $\frac{|l(\beta^{(n)})-l(\beta^{(n-1)})}{|l(\beta^{(n-1)})|+0.001}<\varepsilon$, where $\varepsilon=10^{-10}$.

From the following table, we see that two sets of estimates are close.

	Gradient descent method	R: glm
[1,]	0.47078994	0.48701675
[2,]	-6.39258011	-7.22185053
[3,]	1.65539473	1.65475615
[4,]	-2.49935840	-1.73763027
[5,]	13.87418518	14.00484560
[6,]	1.06741134	1.07495329
[7,]	-0.03285503	-0.07723455
[8,]	0.68188209	0.67512313
[9,]	2.60331111	2.59287426
[10,]	0.44515708	0.44625631
[11,]	-0.48552211	-0.48248420

Figure 3: Comparison of results from Newton's method and gl

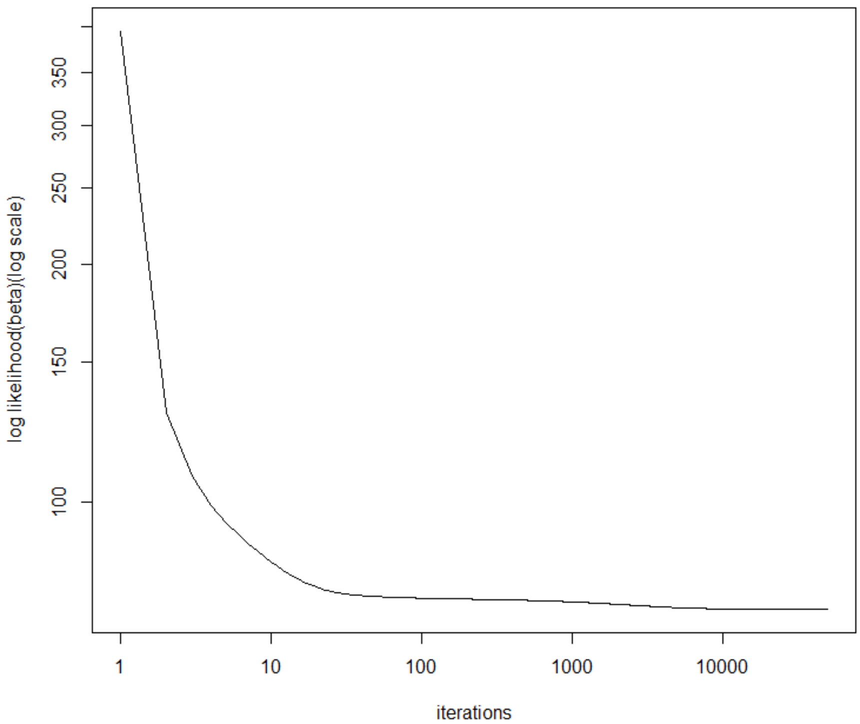


Figure 4: Log likelihood trace plot for gradient descent

(C) First, we need to calculate the Hessian matrix of log likelihood function $\nabla^2 l(\beta)$, which is a key part of Taylor series expansion.

From (A), we have $\nabla l(\beta) = -\sum_{i=1}^{N}(y_i - m_i w_i)x_i = -X^T(y - mw)$ and $\frac{\partial w_i}{\partial \beta} = x_i w_i (1 - w_i)$

$$\Rightarrow \nabla^2 l(\beta) = -\frac{\partial}{\partial \beta}\sum_{i=1}^{N}(y_i - m_i w_i)x_i = \sum_{i=1}^{N} m_i x_i \frac{\partial w_i}{\partial \beta} = \sum_{i=1}^{N} m_i x_i x_i w_i (1 - w_i)$$

Written in matrix form, $\nabla^2 l(\beta) = X^T W X$

where $W = diag[m_1 w_1 (1 - w_1), \ldots, m_N w_N (1 - w_N)]$

Let $v = y - mw$, then $\nabla l(\beta) = -X^T v$

Recall Taylor series second-order expansion in general form:

$$q(x;a) = f(a) + h(a)^T (x - a) + \frac{1}{2}(x - a)^T H(a)(x - a)$$

Where, $h(a)$ gradient evaluated at point a.

$H(a)$ Hessian evaluated at point a.

Then, we have $f(x) = c + bx + \frac{1}{2}x^T ax = \frac{1}{2}(x^T ax + 2bx + c) = \frac{1}{2}(x - u)^T a(x - u)$

$$= \frac{1}{2}(x^T ax - 2u^T ax + u^T au)$$

$$\Rightarrow b = -u^T a, c = \frac{1}{2}u^T au \Rightarrow u = -(ba^{-1})^T, c = \frac{1}{2}b(a^{-1})^T b^T$$

So $q(\beta;\beta_0) = l(\beta_0) + (-X^T v)^T (\beta - \beta_0) + \frac{1}{2}(\beta - \beta_0)^T X^T W X(\beta - \beta_0)$

$$= \frac{1}{2}[2c - 2v^T X(\beta - \beta_0) + (X(\beta - \beta_0))^T WX(\beta - \beta_0)] + l(\beta_0) - c$$

$$= \frac{1}{2}[(u - X(\beta - \beta_0))^T W((u - X(\beta - \beta_0))] + l(\beta_0) - c$$

Where $u = -(-v^T W^{-1})^T = W^{-1}v, c = \frac{1}{2}(-v^T)(W^{-1})^T(-v^T)^T = \frac{1}{2}v^T W^{-1} v$

$$\Rightarrow q(\beta;\beta_0) = \frac{1}{2}[(W^{-1}v + X\beta_0 - X\beta)^T W(W^{-1}v + X\beta_0 - X\beta)] + l(\beta_0) - c$$

$$= \frac{1}{2}[(z - X\beta)^T W(z - X\beta)] + c^*$$

Where
$$z = W^{-1}v + X\beta_0 = W^{-1}(y - mw) + X\beta_0$$
$$c^* = l(\beta_0) - c = l(\beta_0) - \frac{1}{2}v^T W^{-1} v = l(\beta_0) - \frac{1}{2}(y - mw)^T W^{-1}(y - mw)$$

c* is a constant that doesn't involve β.

(D) Here we use Newton's method to estimate. This iterative process requires far fewer iterations to achieve convergence since we are taking the curvature of the objective function $l(\beta)$ into account. We actually only use 10 iterations and achieve estimates which are exactly in line with estimates from ***glm***.

Newton's Method: $\hat{\beta}_{t+1} = \hat{\beta}_t - (\nabla^2 l(\hat{\beta}_t))^{-1} \nabla l(\hat{\beta}_t)$

	Newton's method	R: glm
	0.48701675	0.48701675
V3	-7.22185053	-7.22185053
V4	1.65475615	1.65475615
V5	-1.73763027	-1.73763027
V6	14.00484560	14.00484560
V7	1.07495329	1.07495329
V8	-0.07723455	-0.07723455
V9	0.67512313	0.67512313
V10	2.59287426	2.59287426
V11	0.44625631	0.44625631
V12	-0.48248420	-0.48248420

Figure 5: Comparison of results from Newton's method and glm

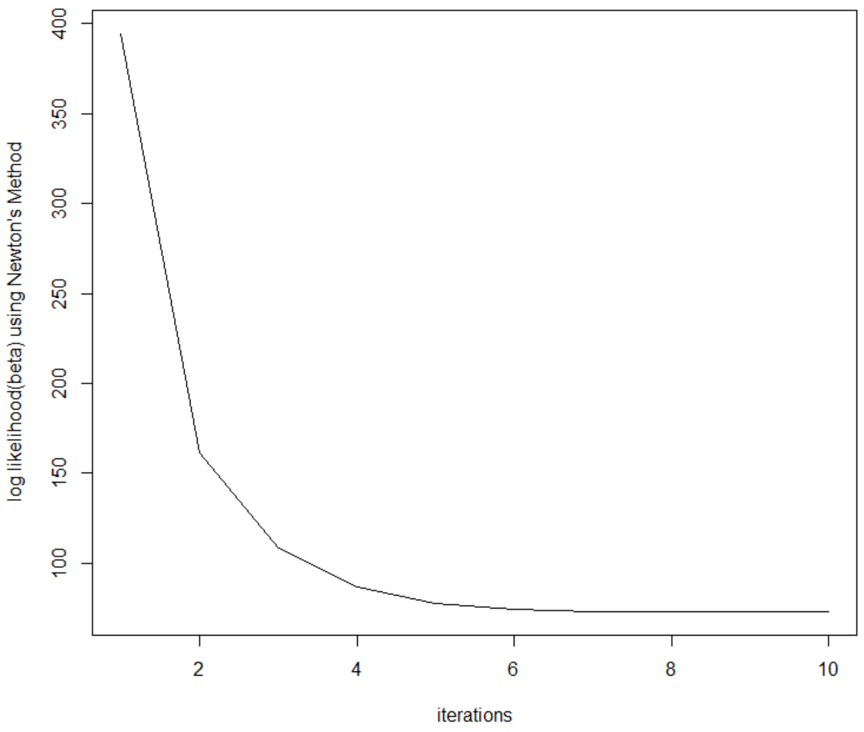


Figure 6: Log likelihood trace plot for Newton's method

exercise_1_LR_part_C_D.R

Shuchen

Thu Sep 21 21:47:46 2017

```r
# SDS385 Exercise 1 Linear Regression: Part C
# compare various matrix decomposition methods with inversion method
# and benchmark peformance of the Cholesky and LU vesus inversion

library (Matrix)
library (microbenchmark) # For benchmarking
```

```r
# Inversion method function
inv_method <- function(X,W,y){

  beta_hat_inv <- solve (t(X) %*% W %*% X) %*% t(X) %*% W %*% y

  return(beta_hat_inv)

}
```

```r
# Cholesky decomposition
chol_method <- function (X,W,y){

  A <- t(X) %*% (X*diag(W))    # Efficient way of A = t(X) %*% W %*% X as W is diagonal
  b <- t(X) %*% (y*diag(W))    # Avoid multiply by 0's

  # Cholesky decomposition of A
  U <- chol(A)   # Upper Cholesky decomposition of A and U'U = A

  # Replace Ax=b with U'Ux=b, solve U'z=b for z
  z <- solve(t(U)) %*% b
  # Solve Ux = z for x (x=beta_hat)
  beta_hat_chol <- solve(U) %*% z

  return(beta_hat_chol)

}
```

```r
# LU method function
lu_method <- function (X,W,y){

  A <- t(X) %*% (X*diag(W))    # Efficient way of A = t(X) %*% W %*% X as W is diagonal
  b <- t(X) %*% (y*diag(W))    # Avoid multiply by 0's

  # LU decomposition of A
  decomp <- lu(A)
  L <- expand(decomp)$L  # Upper triangular matrix
  U <- expand(decomp)$U  # Lower triangular matrix

  # Replace Ax=b with LUx=b, solve Lz=b for z
  z <- solve(L) %*% b
  # Solve Ux=z for x (x=beta_hat)
```

```r
  beta_hat_lu <- solve(U) %*% z

  return(beta_hat_lu)

}
```

```r
# Simulate data from the linear model for a range of values of N and P
# Assume weights w_i are all 1, data are Gaussian

N <- c(20, 100, 400, 1200)
P <- N/4 # N>P

res <- list() # Performance results

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:
  res[[i]] <- microbenchmark (
    inv_method(X,W,y),
    lu_method(X,W,y),
    chol_method(X,W,y),
    unit ='ms'
  )

}
```

```
## [1] 20
## [1] 100
## [1] 400
## [1] 1200
```

```r
names(res) <- (c('N=20, P=5', 'N=100, P=25', 'N=400, P=100', 'N=1200, P=300'))
res # Display benchmarking results
```

```
## $`N=20, P=5`
## Unit: milliseconds
##                 expr      min        lq      mean    median       uq
##   inv_method(X, W, y) 0.075835 0.0821060 0.1780734 0.0866675 0.091799
##    lu_method(X, W, y) 0.164213 0.1744750 0.3588151 0.1796075 0.192721
##  chol_method(X, W, y) 0.158510 0.1690585 0.2981707 0.1761865 0.188160
##        max neval
##   4.760441   100
##  17.230870   100
##  10.952608   100
##
## $`N=100, P=25`
## Unit: milliseconds
```

```
##                  expr      min        lq      mean    median        uq
##   inv_method(X, W, y) 0.623208 0.6303355 0.6888188 0.6591295 0.6856430
##    lu_method(X, W, y) 0.273116 0.2885120 0.3425420 0.3118890 0.3324155
##  chol_method(X, W, y) 0.317591 0.3272840 0.3759148 0.3489510 0.3680520
##       max neval
##  1.215626   100
##  2.768229   100
##  2.013309   100
##
## $`N=400, P=100`
## Unit: milliseconds
##                  expr       min        lq      mean    median        uq
##   inv_method(X, W, y) 30.318235 31.204012 32.193846 31.790443 32.603521
##    lu_method(X, W, y)  4.228463  4.396380  5.025233  4.729652  5.433540
##  chol_method(X, W, y)  5.474308  5.704092  7.488826  6.007712  6.512322
##         max neval
##   47.043355   100
##    8.072623   100
##  131.037291   100
##
## $`N=1200, P=300`
## Unit: milliseconds
##                  expr      min       lq     mean   median        uq
##   inv_method(X, W, y) 802.8075 817.6188 938.7094 831.1985 1162.4563
##    lu_method(X, W, y) 100.0485 104.0201 119.0944 106.1389  143.6161
##  chol_method(X, W, y) 128.0239 132.5625 151.8183 135.0625  143.3067
##        max neval
##  1254.2928   100
##   224.4404   100
##   327.6830   100
```

```r
# SDS385 Exercise 1 Linear Regression: Part D
# Benchmark the inversion method, Cholesky method, LU method,
# and the sparse method across some different scenarios (including different
# sparsity levels in X (0.01, 0.05, 0.25)
```

```r
# Sparse Cholesky factorization
sparse_chol <- function(X,W,y) {

  X <- Matrix(X, sparse = T)
  A <- t(X) %*% (X*diag(W))    # Efficient way of A = t(X) %*% W %*% X as W is diagonal
  b <- t(X) %*% (y*diag(W))    # Avoid multiply by 0's

  # Cholesky decomposition of A
  U <- chol(A)  # Upper Cholesky decomposition of A and U'U = A

  # Replace Ax=b with U'Ux=b, solve U'z=b for z
  z <- forwardsolve(t(U), b)
  # Solve Ux = z for x (x=beta_hat_sparse)
  beta_hat_sparse <- backsolve(U, z)

  return(beta_hat_sparse)

}
```

```r
# Set different sparsity: 0.01, 0.05, 0.25
theta <- c(0.01, 0.05, 0.25)

results <- list() # Performance results
for (i in 1: length(theta)){

  N <- 2000
  P <- 1000

  X <- matrix(rnorm(N * P), nrow = N)
  mask <- matrix(rbinom(N * P, 1, theta), nrow = N)
  X <- mask * X
  y <- rnorm(N)
  W <- diag(rep(1, N))

  # Perform benchmarking:
  results[[i]] <- microbenchmark (
    inv_method(X,W,y),
    lu_method(X,W,y),
    chol_method(X,W,y),
    sparse_chol(X,W,y),
    times=10)
}
names(results) <- (c('theta=0.01', 'theta=0.05', 'theta=0.25'))
results # Display benchmarking results
```

```
## $`theta=0.01`
## Unit: milliseconds
##                  expr        min        lq      mean     median        uq
##   inv_method(X, W, y) 13745.1844 15259.516 15614.799 15676.0385 16280.700
##    lu_method(X, W, y)  3023.9579  3102.272  3189.329  3162.7813  3240.988
##  chol_method(X, W, y)  3494.2124  4777.135  5084.049  4977.4881  5208.347
##  sparse_chol(X, W, y)   920.2551   932.502  1017.857   986.9976  1085.788
##        max neval
##  16541.581    10
##   3471.320    10
##   6775.999    10
##   1213.819    10
##
## $`theta=0.05`
## Unit: milliseconds
##                  expr        min         lq      mean     median         uq
##   inv_method(X, W, y) 14652.7626 15064.6990 15345.165 15172.5969 15611.200
##    lu_method(X, W, y)  3017.4116  3039.8847  3155.543  3136.6986  3244.646
##  chol_method(X, W, y)  4582.3344  4675.4758  4851.868  4711.3638  4823.323
##  sparse_chol(X, W, y)   926.4837   932.6234   983.463   941.1545  1067.369
##        max neval
##  16181.298    10
##   3348.474    10
##   6080.825    10
##   1109.063    10
##
## $`theta=0.25`
## Unit: milliseconds
```

```
##                   expr        min         lq      mean     median        uq
##   inv_method(X, W, y) 14751.3572 14966.8599 15829.914 15150.3468 17061.690
##    lu_method(X, W, y)  3001.8902  3008.1017  3126.951  3048.4249  3165.792
##  chol_method(X, W, y)  4583.1486  4631.7566  4710.708  4678.7540  4815.729
##  sparse_chol(X, W, y)   927.5967   941.1043  1010.472   991.5972  1058.673
##        max neval
##  18141.409    10
##   3619.376    10
##   4908.306    10
##   1148.049    10
```

```
#END
```

exercise01_Generalized_Linear_Models.R

Shuchen

Sun Sep 24 20:04:17 2017

```r
### SDS385 Exercise 01 Generalized Linear Models: Part B
# This code implements gradient descent to estimate the
# beta coefficients(MLE) for binomial logistic regression.
```

```r
library(Matrix)
# Read in data
wdbc <- read.csv("D:/2017 UT Austin/Statistical Models for Big Data/R/wdbc.csv", header = FALSE)
y <- wdbc[ ,2]

X <- as.matrix (wdbc [, 3:12]) # Select first 10 features to keep and scale features
X <- scale (X) # Normalize design matrix features
X <- cbind (rep (1, nrow (X)), X)
y <- wdbc [, 2]
y <- y == "M" # Convert y values to 1/0's, response vector
beta <- as.matrix (rep (0, ncol (X)))
m <- rep(1, nrow (X)) # Number of trials is 1
```

```r
# Compute Sigmoid fuunction wi
sigmoid <- function(z){
  w <- 1 / (1 + exp(-z))
  return(w)
}
```

```r
# Function for computing likelihood, which handles the case that -X^T * beta is huge

loglik <- function(X, y, beta, m) {
  w <- sigmoid(X %*% beta)
  loglik <- - sum(y * log(w+1E-5) + (m-y) * log (1-w+1E-5))
  # Adding a constant to resolve issues with probabilities near 0 or 1.
  return(loglik)
}
```

```r
# Function for computing gradient for likelihood
   # Input: design matrix X, response vector Y
   #        Coefficient matrix beta, sample size (all 1's) vector m
   # Output: Returns value of gradient function for binomial logistic fuction

gradient <- function(X, y, beta, m){
  w <- sigmoid(X %*% beta) # Probabilities vector
  gradient <- array(NA, dim = length(beta)) # Initialize the gradient
  gradient <- apply(X * as.numeric(m * w - y), 2, sum)
  return(gradient)
}
```

```r
# Gradient Descent Algorithm
stepsize <- 0.01
n.steps <- 50000
epsi <- 1E-10 # Level for determining convergence
converged <- 0 # 1/0, depending on whether algorithm converged
```

```r
# Initialize vector to hold loglikelihood function
log.lik <- rep(NULL, n.steps)
# Initialize values for first iteration
log.lik[1] <- loglik(X, y, beta, m)

# Initialize matrix to hold gradients for each iteration
grad <- matrix (0, nrow = n.steps, ncol = ncol (X))
# Initialize values for first iteration
grad[1,] <- gradient (X, y, beta, m)

for (step in 2:n.steps) {
  beta <- beta - stepsize * gradient(X, y, beta, m)

  # Calculate log liklihood for each iteration
  log.lik[step] <- loglik(X, y, beta, m)

  # Calculate gradient for beta
  grad[step, ] <- gradient(X, y, beta, m)

  # Check if convergence met: If yes, exit loop
  if (abs(log.lik[step] - log.lik[step-1]) / (abs(log.lik[step-1]) + 1E-3) < epsi){
    converged = 1;
    break ;
  }
}
```

```r
# view the result
beta
```

```
##              [,1]
##  [1,]  0.47078994
##  [2,] -6.39258011
##  [3,]  1.65539473
##  [4,] -2.49935840
##  [5,] 13.87418518
##  [6,]  1.06741134
##  [7,] -0.03285503
##  [8,]  0.68188209
##  [9,]  2.60331111
## [10,]  0.44515708
## [11,] -0.48552211
```

```r
# Fit glm model for comparison (No intercept: already added to X)
fit <- glm(y ~ X[, c(-1)], family = "binomial") # Fits model, obtains beta values.
```

```
## Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred
```

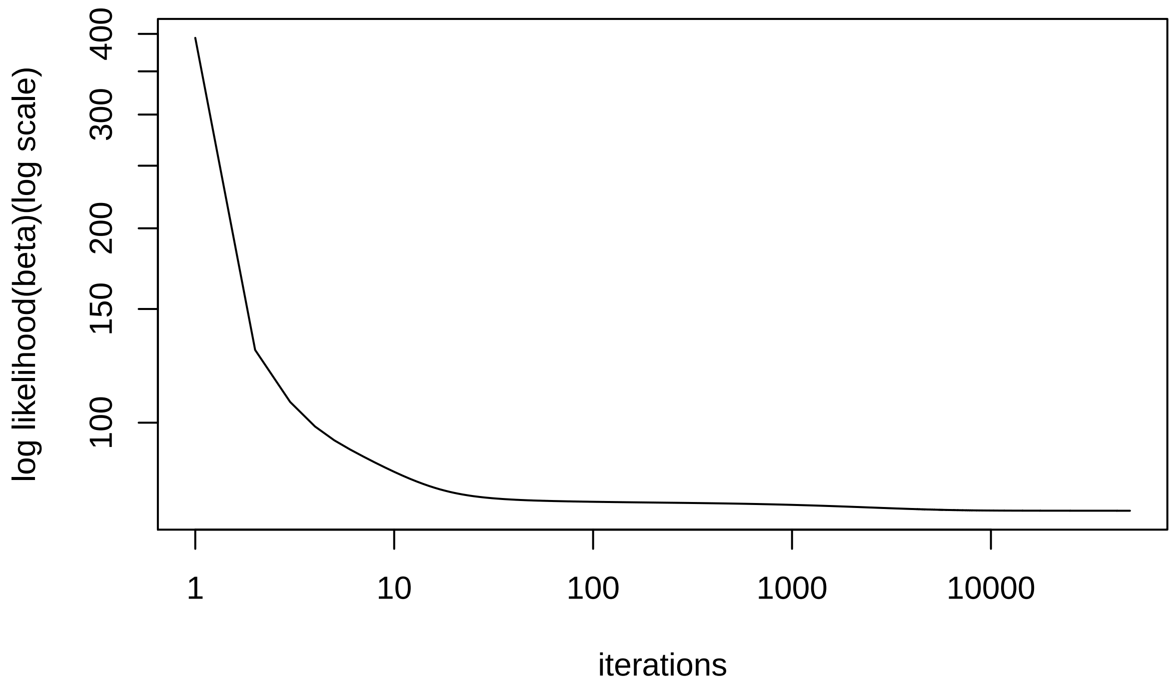
```r
summary(fit)
```

```
##
## Call:
## glm(formula = y ~ X[, c(-1)], family = "binomial")
##
## Deviance Residuals:
##      Min        1Q    Median        3Q       Max
```

```
## -1.95590  -0.14839  -0.03943   0.00429   2.91690
##
## Coefficients:
##               Estimate Std. Error z value Pr(>|z|)
## (Intercept)    0.48702    0.56432   0.863   0.3881
## X[, c(-1)]V3  -7.22185   13.09494  -0.551   0.5813
## X[, c(-1)]V4   1.65476    0.27758   5.961  2.5e-09 ***
## X[, c(-1)]V5  -1.73763   12.27499  -0.142   0.8874
## X[, c(-1)]V6  14.00485    5.89090   2.377   0.0174 *
## X[, c(-1)]V7   1.07495    0.44942   2.392   0.0168 *
## X[, c(-1)]V8  -0.07723    1.07434  -0.072   0.9427
## X[, c(-1)]V9   0.67512    0.64733   1.043   0.2970
## X[, c(-1)]V10  2.59287    1.10701   2.342   0.0192 *
## X[, c(-1)]V11  0.44626    0.29143   1.531   0.1257
## X[, c(-1)]V12 -0.48248    0.60406  -0.799   0.4244
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 751.44  on 568  degrees of freedom
## Residual deviance: 146.13  on 558  degrees of freedom
## AIC: 168.13
##
## Number of Fisher Scoring iterations: 9
```

```r
beta.glm <- fit$coefficients

# Create trace plot of likelihood, check for convergence again
plot (1:length(log.lik), log.lik, type ="l", ylab = "log likelihood(beta)(log scale)",
      xlab ="iterations",log ="xy")
```




```r
#Newton's Methods
beta.nt <- as.matrix(rep(0, ncol(X)))
nsteps <- 10

# Initialize values
log_lik <- rep(NULL, nsteps)
log_lik[1] <- loglik(X, y, beta.nt, m)

for (step in 2:nsteps) {
  w <- as.numeric(sigmoid(X %*% beta.nt))
  Hessian <- t(X) %*% diag(w*(1-w)) %*% X # Compute Hessian matrix
  beta.nt <- beta.nt - solve(Hessian) %*% gradient(X, y, beta.nt, m)
  log_lik[step] <- loglik(X, y, beta.nt, m)

  # Check if convergence met: If yes, exit loop
  if (abs(log_lik[step] - log_lik[step-1]) / (abs(log_lik[step-1]) + 1E-3) < epsi){
    converged = 1;
    break ;
  }
}

# Show estimates from Newton's method
beta.nt
```

```
##            [,1]
##      0.48701675
## V3  -7.22185053
```

```
## V4   1.65475615
## V5  -1.73763027
## V6  14.00484560
## V7   1.07495329
## V8  -0.07723455
## V9   0.67512313
## V10  2.59287426
## V11  0.44625631
## V12 -0.48248420
```

```
log_lik
```

```
##  [1] 394.38937 161.55837 108.22240  86.59502  77.83202  73.99941  73.09228
##  [8]  73.05701  73.05694  73.05694
```

```
# Create trace plot of likelihood, check for convergence
plot (1:length(log_lik), log_lik, type ="l", ylab = "log likelihood(beta) using Newton's Method",
      xlab ="iterations")
```


```
#END
```