SDS 385: Exercise 01

August 27, 2016

Jennifer Starling

Exercises 1: Preliminaries

Linear regression

Consider the simple linear regression model

$$y = X\beta + e\,,$$

where $y = (y_1, \ldots, y_N)$ is an N-vector of responses, X is an $N \times P$ matrix of features whose ith row is x_i, and e is a vector of model residuals. The goal is to estimate β, the unknown P-vector of regression coefficients.

Let's say you trust the precision of some observations more than others, and therefore decide to estimate β by the principle of weighted least squares (WLS):

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

where w_i is the weight for observation i. (Higher weight means more influence on the answer; the factor of 1/2 is just for convenience, as you'll see later.)

(A) Rewrite the WLS objective function[1] above in terms of vectors and matrices, and show that $\hat{\beta}$ is the solution to the following linear system of P equations in P unknowns:

$$(X^TWX)\hat{\beta} = X^TWy\,,$$

where W is the diagonal matrix of weights.

[1] That is, the thing to be minimized.

(B) One way to calculate $\hat{\beta}$ is to: (1) recognize that, trivially, the solution to the above linear system must satisfy $\hat{\beta} = (X^TWX)^{-1}X^TWy$; and (2) to calculate this directly, i.e. by inverting X^TWX. Let's call this the "inversion method" for calculating the WLS solution.

Numerically speaking, is the inversion method the fastest and most stable way to actually solve the above linear system? Do some independent sleuthing on this question. Summarize what you find, and provide pseudo-code for at least one alternate method based on matrix factorizations—call it "your method" for short. (Note: our linear system is not a special flower; whatever you discover about general linear systems should apply here.)

(C) Code up functions that implement both the inversion method and your method for an arbitrary X, y, and weights W. Obviously you shouldn't write your own linear algebra routines for doing things like multiplying or decomposing matrices, but don't use a direct model-fitting function like R's "lm" either. Your actual code should look a lot like the pseudo-code you wrote for the previous part. Note: be attentive to how you multiply a matrix by a diagonal matrix, or you'll waste a lot of time multiplying stuff by zero.

Now simulate some silly data from the linear model for a range of values of N and P. (Feel free to assume that the weights w_i are all 1.) It doesn't matter how you do this—e.g. everything can be Gaussian if you want. (We're not concerned with statistical principles yet, just with algorithms, and using least squares is a pretty terrible idea for enormous linear models, anyway.) Just make sure that you explore values of P up into the thousands, and that $N > P$. Benchmark the performance of the inversion solver and your solver across a range of scenarios. (In R, a simple library for this purpose is microbenchmark.)

(D) Now what happens if X is a highly sparse matrix, in the sense that most entries are zero? Ideally we'd realize some savings by not doing a whole bunch of needless multiplication by zero in our code.

It's easy to simulate an X matrix that looks like this. A quick-and-dirty way is to simulate a mask of zeros and ones (but mostly zeros), and then do pointwise multiplication with your original feature matrix. For example:

```
N = 2000
P = 500
X = matrix(rnorm(N*P), nrow=N)
mask = matrix(rbinom(N*P,1,0.05), nrow=N)
X = mask*X
X[1:10, 1:10]  # quick visual check
```

Again assume that the weights w_i are all 1. Repeat the previous benchmarking exercise with this new recipe for simulating a sparse X, except add another solver to the mix: one that can solve a linear system $Ax = b$ in a way that exploits the sparsity of A. To do this, you'll need to actually represent the feature matrix X in a sparse format, and then call the appropriate routines for that format. (Again, do some sleuthing; in R, the Matrix library has data structures and functions that can do this; SciPy will have an equivalent.)

Benchmark the inversion method, your method, and the sparse method across some different scenarios (including different sparsity levels in X, e.g. 5% dense in my code above).

Generalized linear models

As an archetypal case of a GLM, we'll consider the binomial logistic regression model: $y_i \sim \text{Binomial}(m_i, w_i)$, where y_i in an integer number of "successes," m_i is the number of trials for the ith case, and the success probability w_i is a regression on a feature vector x_i given by the inverse logit transform:

$$w_i = \frac{1}{1 + \exp\{-x_i^T \beta\}} .$$

We want to estimate β by the principle of maximum likelihood. Note: for binary logistic regression, $m_i = 1$ and y_i is either 0 or 1.

As an aside, if you have a favorite data set or problem that involves a different GLM—say, a Poisson regression for count data—then feel free to work with that model instead throughout this entire section. The fact that we're working with a logistic regression isn't essential here; any GLM will do.

(A) Start by writing out the negative log likelihood,

$$l(\beta) = -\log \left\{ \prod_{i=1}^{N} p(y_i \mid \beta) \right\} .$$

Simplify your expression as much as possible. This is the thing we want to minimize to compute the MLE. (By longstanding convention, we phrase optimization problems as minimization problems.)

Derive the gradient of this expression, $\nabla l(\beta)$. Note: your gradient will be a sum of terms $l_i(\beta)$, and it's OK to use the shorthand

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

in your expression.

(B) Read up on the method of steepest descent, i.e. gradient descent, in Nocedal and Wright (see course website). Write your own function that will fit a logistic regression model by gradient descent. Grab the data "wdbc.csv" from the course website, or obtain some other real data that interests you, and test it out. The WDBC file has information on 569 breast-cancer patients from a study done in Wisconsin. The first column is a patient ID, the second column is a classification of a breast cell (Malignant or Benign), and the next 30 columns are measurements computed from a digitized image of the cell nucleus. These are things like radius, smoothness, etc. For this problem, use the first 10 features for X, i.e. columns 3-12 of the file. If you use all 30 features you'll run into trouble.

Some notes here:

1. You can handle the intercept/offset term by either adding a column of 1's to the feature matrix X, or by explicitly introducing an intercept into the linear predictor and handling the intercept and regression coefficients separately, i.e.

$$w_i(\beta) = \frac{1}{1+\exp\{-(\alpha + x_i^T\beta)\}} \, .$$

2. I strongly recommend that you write a self-contained function that, for given values of β, y, X, and sample sizes m_i (which for the WDBC data are all 1), will calculate the gradient of $l(\beta)$. Your gradient-descent optimizer will then call this function. Modular code is reusable code.
3. Make sure that, at every iteration of gradient descent, you compute and store the current value of the log likelihood, so that you can track and plot the convergence of the algorithm.
4. Be sensitive to the numerical consequences of an estimated success probability that is either very near 0, or very near 1.
5. Finally, you can be as clever as you want about the gradient-descent step size. Small step sizes will be more robust but slower; larger step sizes can be faster but may overshoot and diverge; step sizes based on line search (Chapter 3 of Nocedal and Wright) are cool but involve some extra work.

(C) Now consider a point $\beta_0 \in \mathcal{R}^P$, which serves as an intermediate guess for our vector of regression coefficients. Show that the second-order Taylor approximation of $l(\beta)$, around the point β_0, takes the form

$$q(\beta;\beta_0) = \frac{1}{2}(z - X\beta)^T W(z - X\beta) + c \, ,$$

where z is a vector of "working responses" and W is a diagonal matrix of "working weights," and c is a constant that doesn't involve β. Give explicit expressions for the diagonal elements W_{ii} and for z_i (which will necessarily involve the point β_0, around which you're doing the expansion).[2]

[2] Remember the trick of completing the square, e.g. `https://justindomke.wordpress.com/completing-the-square-in-n-dimensions/`.

(D) Read up on Newton's method in Nocedal and Wright, Chapter 2. Implement it for the logit model and test it out on the same data set you just used to test out gradient descent.[3] Note: while you could do line search, there is a "natural" step size of 1 in Newton's method.

[3] You should be able to use your own solver for linear systems from the first section.

(E) Reflect broadly on the tradeoffs inherent in the decision of whether to use gradient descent or Newton's method for solving a logistic-regression problem.

Linear Regression

Part A

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

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

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) $\frac{\delta}{\delta\beta}[\beta'X'WX'\beta] = 2XWX'\beta$, 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. Difference 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 NxP matrix

(b) W, a diagonal matrix of weights

(c) Y, an Nx1 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 $z = R^{-1}z$

(4) Return $\hat{\beta}$ estimate as $\hat{\beta} = x$

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

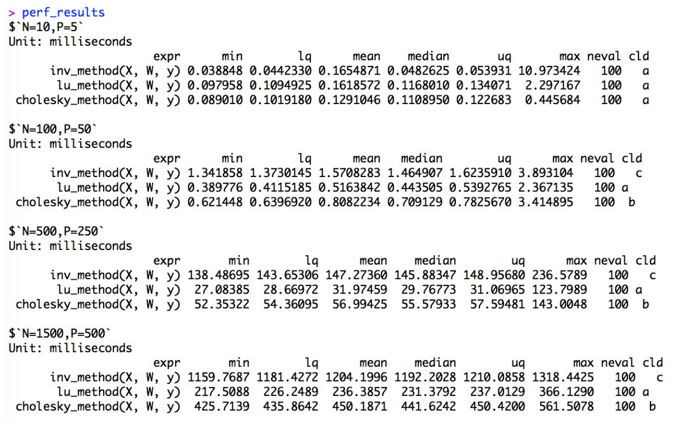
Part C

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

```r
### 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

library(Matrix) #For matrix decomposition.
library(microbenchmark) #For benchmarking

### PART C:

#--------------------------------------------------
#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
#       estimating the beta vector of coefficients.

#   Matrix requirements:
#   1. Length X = Length Y = Dim(W)
#   2. t(X) %*% W %*% X must be invertible

inv_method <- function(X,W,y){
    B_hat <- solve(t(X) %*% W %*% X) %*% t(X) %*% W %*% y
    return(B_hat)
}

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

lu_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

    #Obtain LU matrix decomposition of A.
    decomp <- lu(A)          #Calculates matrix decomposition object..
    L <- expand(decomp)$L    #Upper triangular matrix
    U <- expand(decomp)$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.
    #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)
    B_hat_LU <- solve(U) %*% d

    return(B_hat_LU)        #Returns function output.
}

#---------------------------------------------------
#Cholesky decomposition 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
#   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 = inv(R')b.
    z = solve(t(R)) %*% b

    #2. Solve Rx=z for x.  This is x = inv(R)z. (x = 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)
```

```r
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'
    )
}

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

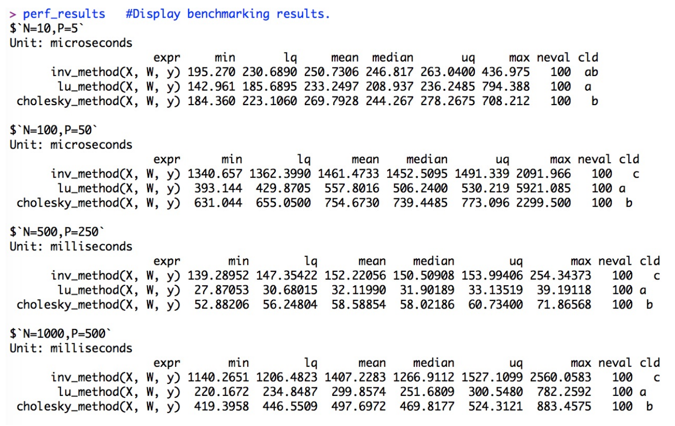
Part D

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:

(a) Benchmarking various N and P at 10% sparse.

(b) Benchmarking various sparsity levels (5%, 10%, 20%, 50%).

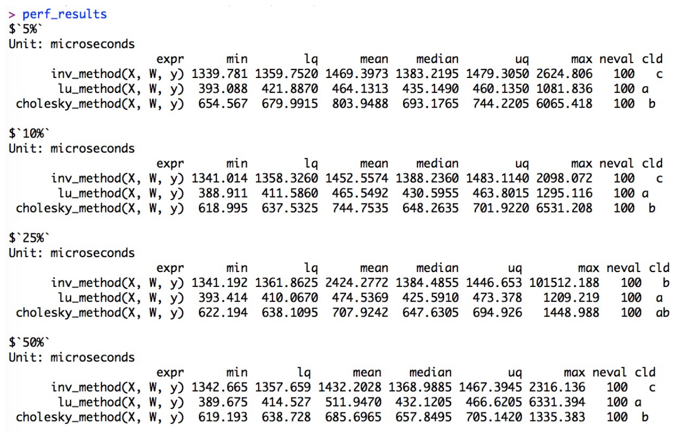
For benchmarking at various N and P levels, results were similar to the results above. Inverse was superior for small N and P, and as N and P increased, Cholesky and LU were superior. LU again performed the most efficiently.



```
> 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

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
$`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.

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

$= -\sum_{i=1}^N log\{{w_i}^{y_i}(1-w_i)^{(m_i-y_i)}\} = -\sum_{i=1}^N \{y_i log(w_i) + (m_i - y_i)log(1-w_i)\}; w_i = \frac{1}{1+exp(x_i'\beta)}$

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:
$\frac{\delta l(\beta)}{\delta\beta} = -\sum_{i=1}^N\{y_i\frac{1}{w_i}(\frac{\delta w_i}{\delta\beta}) + (m_i - y_i)\frac{1}{1-w_i}(-\frac{\delta w_i}{\delta\beta})\} = -\sum_{i=1}^N\{y_i\frac{1}{w_i}(\frac{x_i exp(-x_i'\beta)}{1+exp(-x_i'\beta)}) + (m_i - y_i)\frac{1}{1-w_i}(\frac{-x_i exp(-x_i'\beta)}{(1+exp(-x_i'\beta))^2})\}$

$= -\sum_{i=1}^N\{y_i\frac{1}{w_i}(w_i^2 x_i exp(-x_i'\beta)) + (m_i - y_i)\frac{1}{1-w_i}(w_i^2 x_i exp(-x_i'\beta))\}$

$= -\sum_{i=1}^N\{y_i w_i exp(-x_i'\beta) - (m_i - y_i)x_i w_i\} = -\sum_{i_1}^N\{(y_i w_i exp(-x_i'\beta) - (m_i - y_i)w_i)x_i\}$

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:

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

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.
> 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
 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
 2.59287426  0.44625631 -0.48248420
```

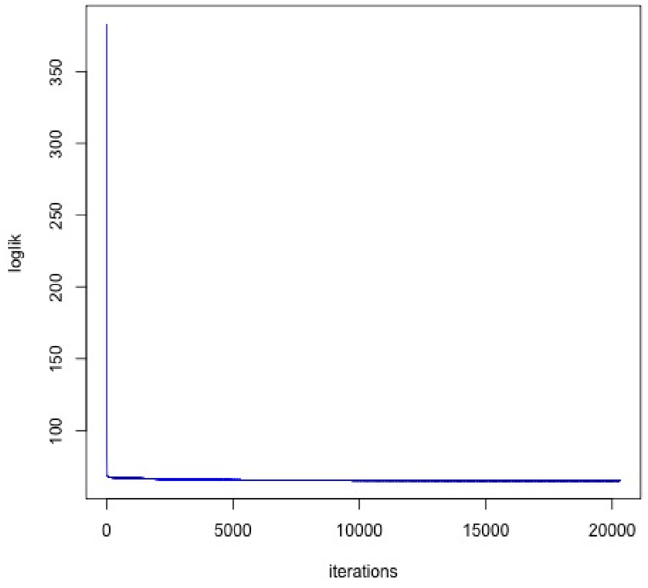


Figure 4: Log-likelihood funcion

The R code is as follows.

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

#Jennifer Starling
#26 August 2016

library(Matrix)
rm(list=ls())

#PART B:

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

#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.
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)
```

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

#———————————————————————————————————————————————————————————————————
#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.
logl <- function(X,Y,beta,m){
    w <- 1 / (1 + exp(-X %*% beta)) #Calculate probabilities vector w_i.
    logl <- - sum(Y*log(w+1E-4) + (m-Y)*log(1-w+1E-4)) #Calculate log—likelihood.
        #Adding constant to resolve issues with probabilities near 0 or 1.
    return(logl)
}

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

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

gradient <- function(X,Y,beta,m){
    w <- 1 / (1 + exp(-X %*% beta)) #Calculate probabilities vector w_i.

    gradient <- array(NA,dim=length(beta))  #Initialize the gradient.
    gradient <- -apply(X*as.numeric(Y-m*w),2,sum) #Calculate the gradient.

    return(gradient)
}

#———————————————————————————————————————————————————————————————————
#Gradient Descent Algorithm:
#Inputs:
#   X: n x p design matrix.
#   Y: response vector length n.
#   m: vector length n.
#   conv: Tolerance level for determining convergence, (length of gradient) < conv
    .
#   a: Step size.

#Outputs:
#   beta_hat: A vector of estimated beta coefficients.
#   iter: The number of iterations until convergence.
#   converged: 1/0, depending on whether algorithm converged.
#   loglik: Log—likelihood function.
```

```r
gradient_descent <- function(X,Y,m,maxiter=50000,conv=1E-10,a=.01){

    #1. Initialize values.
    loglik <- rep(0,maxiter)    #Initialize vector to hold loglikelihood function.

    #Initialize matrix to hold gradients for each iteration.
    grad <- matrix(0,nrow=maxiter,ncol=ncol(X))

    #Initialize matrix to hold beta vector for each iteration.
    betas <- matrix(0,nrow=maxiter+1,ncol=ncol(X))

    converged <- 0      #Indicator for whether convergence met.

    #Initialize values for first iteration.
    betas[1,] <- rep(0,ncol(X)) #Initialize beta vector to 0 to start.
    loglik[1] <- logl(X,Y,betas[1,],m)
    grad[1,] <- gradient(X,Y,betas[1,],m)

    #2. Perform gradient descent.
    for (i in 2:maxiter){

        #Set new beta equal to beta - a*gradient(beta).
        betas[i,] <- betas[i-1,] - a * grad[i-1,]

        #Calculate loglikelihood for each iteration.
        loglik[i] <- logl(X,Y,betas[i,],m)

        #Calculate gradient for beta.
        grad[i,] <- gradient(X,Y,betas[i,],m)

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

    } #End gradient descent iterations.

    return(list(beta_hat=betas[i,], iter=i, converged=converged, loglik=loglik[1:i
       ]))
}

#-------------------------------------------------------------------
#Run gradient descent and view results.

#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

#2. Call gradient descent function to estimate.
beta_hat <- gradient_descent(X,Y,m,maxiter=100000,conv=1E-10,a=.01)

#3. Eyeball values for accuracy & display convergence.
```

```
beta                  #Glm estimated beta values.
beta_hat$beta_hat     #Gradient descent estimated beta values.

print(c("Algorithm converged? ",beta_hat$converged, " (1=converged, 0=did not
    converge)"))
print(beta_hat$iter)

#4. Plot log-likelihood function for convergence.
plot(1:length(beta_hat$loglik),beta_hat$loglik,type='l',xlab='iterations',col='
    blue',log='xy')

#Save plot.
jpeg(file='/Users/jennstarling/UTAustin/2016_Fall_SDS 385_Stats Models for Big
    Data/385_Exercise_R_Code/R_Output/Ex01_B_loglik.jpeg')
plot(1:length(beta_hat$loglik),beta_hat$loglik,type='l',xlab='iterations',col='
    blue')
dev.off()
```

Part C

Hessian:
First, find the Hessian of $l(\beta)$, as this will be a key part of the Taylor series expansion.

$\nabla^2 l(\beta) = \frac{\delta}{\delta\beta}[-\sum_{i=1}^N (y_i x_i - m_i w_i x_i)] = \sum_{i=1}^N m_i x_i (\frac{\delta w_i}{\delta\beta})$.

From previous parts, $\frac{\delta w_i}{\delta\beta} = x_i {w_i}^2 exp(-x_i'\beta)$, and $exp(-x_i'\beta) = (\frac{1}{w_i} - 1)$, so plug in to get
$\frac{\delta w_i}{\delta\beta} = x_i {w_i}^2 (\frac{1}{w_i} - 1) = x_i w_i (1 - w_i)$.

Then $\nabla^2 l(\beta) = \frac{\delta^2 \beta}{\delta\beta\delta\beta'} = \sum_{i=1}^N m_i x_i x_i w_i (1 - w_i)$.

In matrix form, $\nabla^2 l(\beta) = X'AX$, with A = diagonal matrix of $m_i w_i (1 - w_i)$ elements.

Taylor Series Second-Order Expansion:
General multivariate 2nd order Taylor series form: $q(x; a) = f(a) + g(a)'(x - a) + \frac{1}{2}(x - a)'H(a)(x - a)$ where $g(a)$ indicates the gradient evaluated at a, and $H(a)$ indicates the Hessian evaluated at a.

$q(\beta; \beta_0) = l(\beta_0) + g'(a)(\beta - \beta_0) + \frac{1}{2}(\beta - \beta_0)'H(\beta - \beta_0)$

Plug in $g(a) = X'(mw - y)$ and $H = X'AX$: $q(\beta; \beta_0) = l(\beta_0) + [X'(mw - y)]'(\beta - \beta_0) + \frac{1}{2}(\beta - \beta_0)'X'AX(\beta - \beta_0)$

Distribute the transpose in the middle term: $q(\beta; \beta_0) = l(\beta_0) + (Y - mw)'X(\beta - \beta_0) + \frac{1}{2}(\beta - \beta_0)'X'AX(\beta - \beta_0)$

Expand terms: $q(\beta; \beta_0) = l(\beta_0) + (Y - mw)'X\beta + (Y - mw)'X\beta_0 + \frac{1}{2}\beta'X'AX\beta - 2(\frac{1}{2})\beta_0'X'AX\beta + \frac{1}{2}\beta_0'X'AX$

The following terms are constant, so let $C = l(\beta_0) + (Y - mw)'X\beta_0 + \frac{1}{2}\beta_0'X'AX\beta_0$.

Then rewrite as: $q(\beta; \beta_0) = C + (Y - mw)'X\beta - \beta_0'X'AX\beta + \frac{1}{2}\beta'X'AX\beta$

Group first-order terms, treaing $X\beta$ as the variable in the quadratic form:
$q(\beta; \beta_0) = C + [(Y - mw)' - AX\beta_0]'X\beta + \frac{1}{2}(X\beta)'AX\beta$

(a) In second term, brought $\beta_0'X'A$ into transpose, so $(\beta_0'X'A)' = A'X\beta_0 = AX/beta_0$ since $A' = A$.

(b) In third term, rearranged so $\beta'X' = (X\beta)'$

Now complete the square, using trick:
$a + b'X + X'CX = \frac{1}{2}(X - m)'M(X - m)$ with $M = C, m = -C^{-1}, v = a - \frac{1}{2}b'C^{-1}b$

For our equation:

(a) In the expanded form, we have $a =$ constant, $b = [(y - mw)' - AX\beta_0], c = A$.

(b) Then our $M = A$, $v =$ constant that does not depend on beta, $m = -A^{-1}[(y - mw)' - AX\beta_0] = [A^{-1}(y - mw) + X\beta_0] = Z$

Therefore, $q(\beta; \beta_0) = \frac{1}{2}(X\beta - Z)'A(X\beta - Z) + C = \frac{1}{2}(Z - X\beta)'A(Z - X\beta) + C$, where:

(a) $A =$ diagonal matrix, with diagonal elements $m_i w_i (1 - w_i)$

(b) $Z = [A^{-1}(y - mw) + X\beta_0]$

(c) $C =$ a constant that does not depend on β_0

Part D

My implementation of the Newton method converged in 10 iterations. Results are as follows.

```
> 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
 2.59287426  0.44625631 -0.48248420
> beta_newt
 [1]  0.48701675 -7.22185053  1.65475615 -1.73763027 14.00484560  1.07495329 -0.07723455
0.67512313
 [9]  2.59287426  0.44625631 -0.48248420
```

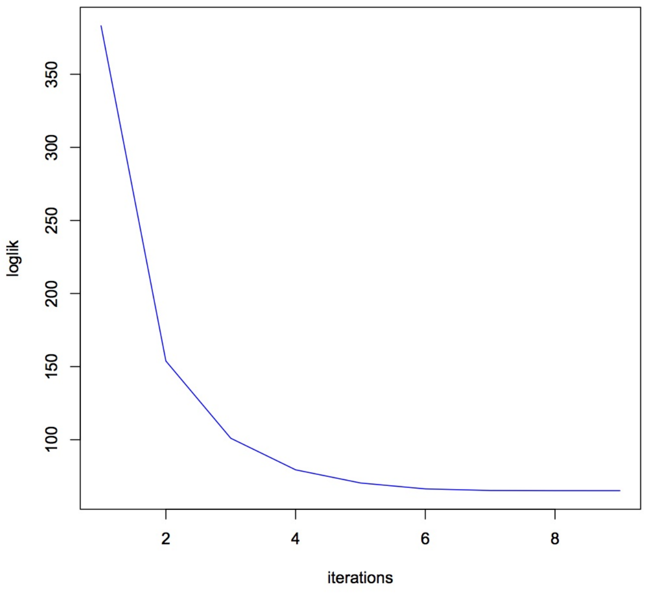


Figure 5: Log-likelihood for Newton algorithm

The R code for implementing the Newton algorithm is as follows.

```
### SDS 385 - Exercises 01 - Part B - Problem D
#This code implements Newton's Method to estimate the
#beta coefficients for binomial logistic regression.

#Jennifer Starling
#26 August 2016

rm(list=ls())
library(Matrix)

#PART C:
```

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

#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.
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)

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

#------------------------------------------------------------------
#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.
logl <- function(X,Y,beta,m){
    w <- 1 / (1 + exp(-X %*% beta)) #Calculate probabilities vector w_i.
    logl <- - sum(Y*log(w+1E-4) + (m-Y)*log(1-w+1E-4)) #Calculate log-likelihood.
        #Adding constant to resolve issues with probabilities near 0 or 1.
    return(logl)
}

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

#------------------------------------------------------------------
#Gradient Function:
    #Inputs: Design matrix X, vector of 1/0 vals Y,
    #   coefficient matrix beta, sample size vector m.
    #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.

    gradient <- array(NA,dim=length(beta))  #Initialize the gradient.
    gradient <- -apply(X*as.numeric(Y-m*w),2,sum) #Calculate the gradient.

    return(gradient)
}

#------------------------------------------------------------------
```

```
#Gradient Function:

hessian <- function(X,Y,beta,m){
    w <- 1 / (1 + exp(-X %*% beta)) #Calculate probabilities vector w_i.

    #Create diag matrix of weights with ith element equal to m_i*w_i*(1-w_i)
    A <- Diagonal(length(m),m*w*(1-w))

    #Calculate Hessian as X'AX.
    H <- t(X) %*% A %*% X
    return(H)
}

#-------------------------------------------------------------------
#QR Solver Function:
qr_decomp <- function(A,b){
    #Solves linear system Ax=b.

    #Obtain QR decomposition of matrix A.  Extract components.
    QR <- qr(A)
    Q <- qr.Q(QR)
    R <- qr.R(QR)

    #Backsolve for x.
    x <- qr.solve(A,b)
    return(x)
}
#-------------------------------------------------------------------

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 = inv(R')b.
    z = solve(t(R)) %*% b

    #2. Solve Rx=z for x.  This is x = inv(R)z. (x = beta_hat)
    B_hat_chol <- solve(R) %*% z

    return(B_hat_chol)
}

#-------------------------------------------------------------------
#Newton's Method algorithm:
```

```
#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.
grad <- list()       #Initialize list to hold gradients for each iteration.
hess <- list()       #Initialize list to hold hessians for each iteration.
maxiter <- 100000    #Specify max iterations allowed.
betas <- list()      #Initialize list to hold beta vector for each iteration.

conv <- 1E-6         #Set convergence level.

#Initialize first iteration of values.
```

Part E

Newton is a second-order optimization method, while Gradient Descent is a first-order. So Newton uses the second derivatives in determining the direction to take each step. This means that Newton takes fewer iterations to find the local min of the cost function than gradient descent.

However, iteration-to-iteration, Newton is a more expensive function to calculate. It requires evaluation of the Hessian matrix, and solving the linear system $Hessian * dir = Gradient$ for the dir (direction) gradient, which is used to update the betas for the next step:

$\beta^{(i+1)} = \beta^{(i)} - H^{-1}(\beta^{(n)}\nabla(\beta^{(n)})$

Instead of inverting the Hessian directly, we can solve the following equation with a QR solver (or other matrix decomposition method).

$H \cdot dir = \nabla$, which yields $dir = H^{-1}\nabla$, and so $\beta^{(i+1)} = \beta^{(i)} - dir$.

This does improve the speed of the Hessian inversion, but is still an added cost compared to the calculation of each Gradient Descent iteration.

The addition of the Hessian matrix may also be problematic for the Newton method; if the Hessian is singular, the method does not work.