SDS 385: Exercise 03

September 8, 2016

Jennifer Starling

Exercises 3: Better online learning (preliminaries)

The goal of the next two sets of exercises is to make your SGD implementation better, faster, and able to exploit sparsity in the features. These exercises set the stage. On the next set, you'll then put everything together.

Once again, we'll return to the logistic-regression model, by now an old friend: $y_i \sim \text{Binomial}(m_i, w_i)$, 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:

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

As before, we'll use $l(\beta)$ to denote the loss function to be minimized: that is, the negative log likelihood for this model.

Before we get to work on improving stochastic gradient descent itself, we need to revisit our batch[1] optimizers from the first set exercises: ordinary gradient descent and Newton's method.

[1] Batch, in the sense that they work with the entire batch of data at once. The key distinction here is between batch learning and online learning, which processes the data points either one at a time, or in mini-batches.

Line search

Line search is a technique for getting a good step size in optimization. You have may already implemented line search on the first set of exercises, but if you haven't, now's the time.

Our iterative (batch) algorithms from the previous exercises involved updates that looked like this:

$$\beta^{(t+1)} = \beta^{(t)} + \gamma s^{(t)} ,$$

where $s^{(t)}$ is called the search direction. We tried two different search directions: the gradient-descent direction (i.e. in the opposite direction from the gradient at the current iterate), and the Newton direction.

In either case, we have the same question: how should we choose γ? It's clear that the best we can do, in a local sense, is to choose γ to minimize the one-dimensional function

$$\phi(\gamma) = l(\beta^{(t)} + \gamma s^{(t)}) ,$$

There's no other choice of γ that will lead to a bigger decrease in the loss function along the fixed search direction $s^{(t)}$. While in general it might be expensive to find the exact minimizing γ, we can do better than just guessing. Line search entails using some reasonably fast heuristic for getting a decent γ.

Here we'll focus on the gradient-descent direction. Read Nocedal and Wright, Section 3.1. Then:

(A) Summarize your understanding of the *backtracking line search* algorithm based on the Wolfe conditions (i.e. provide pseudo-code and any necessary explanations) for choosing the step size.

(B) Now implement backtracking line search as part of your batch gradient-descent code, and apply it to fit the logit model to one of your data sets (simulated or real). Compare its performance with some of your earlier fixed choices of step size. Does it converge more quickly?

Remember that modular code is reusable code. For example, one way of enhancing modularity in this context is to write a generic line-search function that accepts a search direction and a callable loss function as arguments, and returns the appropriate multiple of the search direction. If you do this, you'll have a line-search module that can be re-used for any method of getting the search direction (like the quasi-Newton method in the next section), and for any loss function.[2]

[2] This is not the only way to go, by any stretch. In particular, my comment assumes that you're adopting a more functional programming style, rather than a primarily object-oriented style with user-defined classes and methods.

Quasi-Newton

On a previous exercise, you implemented Newton's method for logistic regression. In this case, you (probably) used a step size of $\gamma = 1$, and your search direction solved the linear system

$$H^{(t)} s^{(t)} = -\nabla l(\beta^{(t)}) \, , \quad \text{or equivalently} \quad s = -H^{-1} \nabla l(\beta) \, ,$$

where $H^{(t)} = \nabla^2 l(\beta^{(t)})$ is the Hessian matrix evaluated at the current iterate. (The second version above just drops the t superscript to lighten the notation.) Newton's method converges very fast, because it uses the second-order (curvature) information from the Hessian. The problem, however, is that you have to form the Hessian matrix at every step, and solve a p-dimensional linear system involving that Hessian, whose computational complexity scales like $O(p^3)$.

Read about quasi-Newton methods in Nocedal and Wright, Chapter 2, starting on page 24. A quasi-Newton method uses an approximate Hessian, rather than the full Hessian, to compute the search direction. This is a very general idea, but in the most common versions, the approximate Hessian is updated at every step using that step's gradient vector. The intuition here is that, because the second derivative is the rate of change of the first derivative, the successive changes in the gradient should provide us with information about the curvature of the loss function. Then:

(A) Briefly summarize your understanding of the *secant condition* and the *BFGS* (Broyden-Fletcher-Goldfarb-Shanno) quasi-Newton method. Provide pseudo-code showing how BFGS can be used, in conjunction with backtracking line search, to fit the logistic regression model.

Note: as discussed in Nocedal and Wright, your pseudo-code should use the BFGS formula to update the *inverse* of the approximate Hessian, rather than the approximate Hessian itself. An important question for you to answer here is: why?

(B) Now implement BFGS coupled with backtracking line search to fit the logit model to one of your data sets (simulated or real). Compare its performance both with Newton's method and with batch gradient descent, in terms of the number of overall steps required.

Gradient Descent with Backtracking Line Search

Part A

The Wolfe Conditions are a set of two requirements for the step size α_k. The purpose of these conditions is to ensure that each iteration progresses by decreasing the objective function.

(1) The first condition is called the Armijo condition. It ensures that the selected step size gives enough decrease in the objective function for each iteration.

$$f(x_k + \alpha p_k) \leq f(x_k) + c_1 \alpha \nabla {f_k}^t p_k$$

In our case, $x_k = \beta_k$ and $p_k = \nabla(\beta)$:

$$l(\beta_k) + \alpha\nabla(\beta_k) \leq l(\beta_k) + c_1\alpha\nabla(\beta_k)'\nabla(\beta_k) =$$
$$l(\beta_k) + \alpha\nabla(\beta_k) \leq l(\beta_k) + c_1\alpha||\nabla(\beta_k)||^2$$

• This is also sometimes called the 'sufficient decrease condition'.
• c_1 is a constant, with $c_1 \in (0, 1)$. c_1 is usually small; 10^{-4} is given as an example.
• The right side of the inequality is a linear function with negative slope. The condition is met for small positive α values, but we also want to make sure the αs are not too small, otherwise the algorithm will take a long time to converge. A second condition is required.

(2) The second condition is called the curvature condition. It ensures that step sizes are not too short.

$$\nabla f(x_k + \alpha_k p_k)' p_k \geq c_2 \nabla f_k' p_k$$

In our case, $x_k = \beta_k$ and $p_k = \nabla(\beta)$:

$$\nabla(\beta_k + \alpha_k\nabla(\beta_k))'\nabla(\beta_k) \geq c_2\nabla(\beta_k), \text{ with } c_2 \in (c_1, 1)$$

The left side of this inequality is the derivative of the left side of the previous inequality, so this condition is requiring that the slope of the left side of the inequality at α_k is at least c_2 times the initial slope.

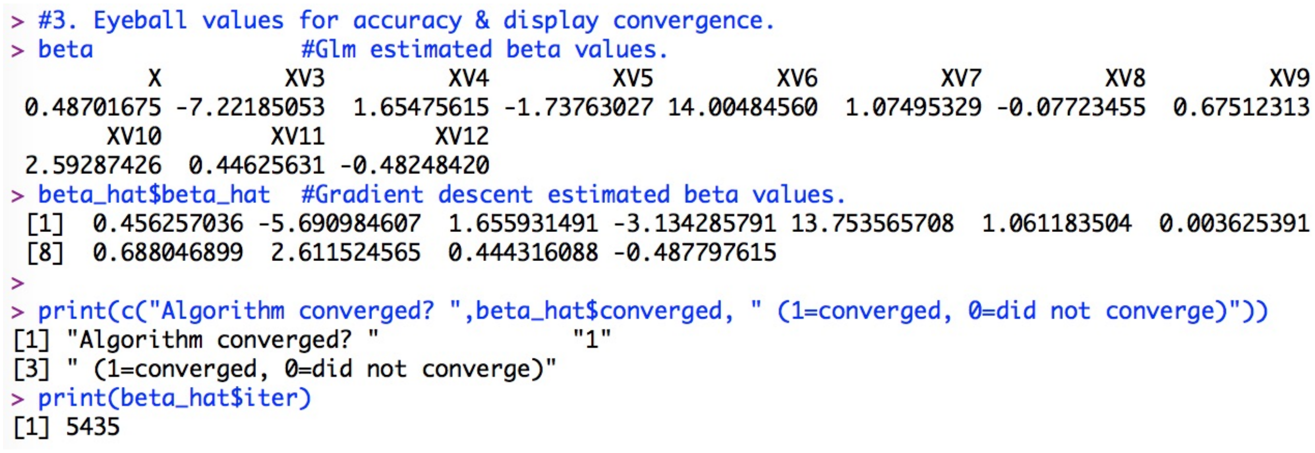
The intuition here is that if the slope is very negative, we should move further in this direction, and if the slope is only slightly negative or is positive, we should not continue the line search in this direction.

The **backtracking line search** is an algorithm that makes the second condition unnecessary. The algorithm is as follows:

```
Choose an initial step size, $\alpha_0$. Set $\alpha = \alpha_0$. This is often set to 1, as the initial step
size acts as the maximum step size; the algorithm will not take a longer step than this.

Choose a $\rho \in (0, 1)$. I chose $\rho = 0.5$. The $\rho$ value represents what $\alpha$ is
multiplied by at each iteration. (At each iteration, $\alpha$ is updated to equal $\alpha * \rho$.)

Choose a constant, $c \in (0, 1)$. This is usually very small; I set $c = 0.01$.

REPEAT UNTIL $l(\beta_k - \alpha\nabla(l(\beta_k)) \leq l(\beta_k) + c\alpha||\nabla(l(\beta))||^2$
    Update $\alpha \leftarrow \rho * \alpha$
(END REPEAT)
Return $\alpha_k = \alpha$ as the step size for this gradient descent iteration.
```

Part B

Results of the gradient descent with backtracking line search are as follows.



```
> #3. Eyeball values for accuracy & display convergence.
> beta               #Glm estimated beta values.
          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_hat$beta_hat  #Gradient descent estimated beta values.
 [1]  0.456257036 -5.690984607  1.655931491 -3.134285791 13.753565708  1.061183504  0.003625391
 [8]  0.688046899  2.611524565  0.444316088 -0.487797615
>
> print(c("Algorithm converged? ",beta_hat$converged, " (1=converged, 0=did not converge)"))
[1] "Algorithm converged? "              "1"
[3] " (1=converged, 0=did not converge)"
> print(beta_hat$iter)
[1] 5435
```

Figure 1: Gradient Descent Output

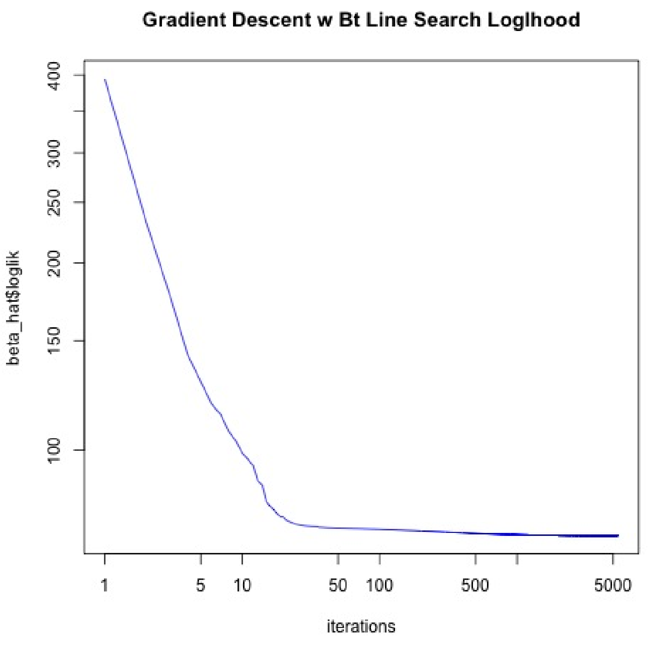


Figure 2: Gradient Descent Negative Log-Likelihood Plot

R Code: Gradient Descent with Backtracking Line Search

```r
### SDS 385 - Exercises 03 - Backtracking Line Search
#This code implements gradient descent to estimate the
#beta coefficients for binomial logistic regression.
#It uses backtracking line search to calculate the step size.

#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)
}

#------------------------------------------------------------------
#Line Search Function
    #Inputs:  X = design matrix
    #         Y = vector of 1/0 response values
    #         b = vector of betas
    #         g = gradient for beta vector
    #         m = sample size vector m
    #         maxalpha = The maximum allowed step size.
    #Outputs: alpha = The multiple of the search direction.

linesearch <- function(X,Y,b,gr,m,maxalpha=1){
    c <- .01             #A constant, in (0,1)
    alpha <- maxalpha    #The max step size, ie the starting step size.
    p <- .5              #The multiplier for the step size at each iteration.

    #Update alpha while condition holds.
    while( (logl(X,Y,b - alpha*gr,m)) > logl(X,Y,b,m) - c*alpha*norm_vec(gr)^2 ) {
        alpha <- p*alpha
    }

    return(alpha)
}

#------------------------------------------------------------------
#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.

gradient_descent <- function(X,Y,m,maxiter=50000,conv=1*10^-10){

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

    #2. Initialize values for log-likelihood.
    loglik <- rep(0,maxiter)    #Initialize vector to hold loglikelihood function.
    loglik[1] <- logl(X,Y,betas[1,],m)

    #Initialize matrix to hold gradients for each iteration.
    grad <- matrix(0,nrow=maxiter,ncol=ncol(X))
    grad[1,] <- gradient(X,Y,betas[1,],m)

    converged <- 0      #Indicator for whether convergence met.
    iter <- 1           #Counter to track iterations for function output.

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

        #Backtracking line search to calculate step size.
        step <- linesearch(X,Y,b=betas[i-1,],gr=grad[i-1,],m,maxalpha=1)

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

        iter <- i   #Track iterations.

        print(iter)

        #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=iter, 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=10000,conv=1*10^-10)

#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',
    main='Gradient Descent w Bt Line Search Loglhood')

```

```
#Save plot.
jpeg(file='/Users/jennstarling/UTAustin/2016_Fall_SDS 385_Stats Models for Big
    Data/Exercise 03 LaTeX Files/Ex03_gradwlinesearch_loglhood.jpeg')
plot(1:length(beta_hat$loglik),beta_hat$loglik,type='l',xlab='iterations',col='
    blue',log='xy',
     main='Gradient Descent w Bt Line Search Loglhood')
dev.off()
```

Problem 2: Quasi-Newton Method

Part A

Secant Condition Intuition

The crux of Quasi Newton's Method is to use an approximation of the Hessian matrix, to avoid the calculation overhead involved in actually computing the Hessian for each iteration. The BFGS method provides a way to approximate the Hessian which ensures that the approximation shares certain properties with the Hessian.

Specifically, we expect the Hessian to be symmetric, and to satisfy a condition called the *Secant Condition*.

The *Secant Condition* is as follows:

$B_{k+1} s_k = z_k$, with

$s_k = x_{k+1} - x_k$,

$z_k = \nabla(f_{k+1}) - \nabla(f_k)$,

B = the Hessian approximation.

This equation rearranges as $B_{k+1} = \frac{z_k}{s_k} = \frac{\nabla(f_{k+1}) - \nabla(f_k)}{x_{k+1} - x_k}$.

The intuition here is that the Hessian matrix H is the second derivative of the log-likelihood, ie the first derivative of the gradient. If we use the finite differences method to approximate the first derivative of the gradient, we get:

$$H_{k+1} \approx \frac{\nabla(l(\beta_{k+1})) - \nabla(l(\beta_k))}{\beta_{k+1} - \beta_k}$$

This is the secant condition, since our $f(x)$ is the log-likelihood function $l(\beta)$, and our x is β. So it is logical to require a Hessian approximation to meet this same condition as the true Hessian.

The BFGS method is an equation to approximate the Hessian which ensures that the approximation is symmetric, and also meets the Secant Condition.

Also note that we are actually approximating the inverse of the Hessian instead of the Hessian itself. This technique is more efficient, so that we do not have to perform the Hessian approximation and then invert the result separately.

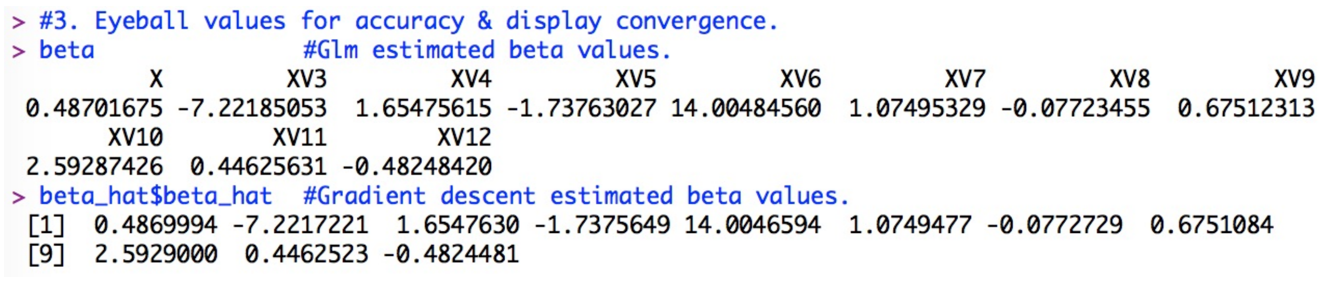
Pseudo-Code for BFGS Quasi-Newton with Backtracking Line Search

The pseudo-code for Quasi-Newton with BFGS and backtracking line search:

```
1. Initialize first iteration of values (i=1).
    (a) Set first Beta vector ($\beta_1$) to all zeros.
    (b) Calculate log-likelihood ($logl_1$) function using first Beta vector.
    (c) Calculate gradient using first Beta vector ($g_1$).
    (d) Set the $pxp$ identity matrix as the initial inverse Hessian approximation ($B_1$).

2. Begin looping, i=2 to maxiter.
    (a) Compute direction and step size for updating the Beta vector ($\beta_i$). ($p$ is direction, $\alpha$ is step size.)
        $p = -B_{i-1}\ g_{i-1}$ (Uses previous iteration's inverse Hessian approximation)
        $\alpha$ = linesearch(f(x)=loglikelihood, x=$\beta_{i-1}$, gradient = $g_{i-1}$,p,maxalpha=1,c=.01,rho=.5)
            (Perform line search according to algorithm described previously.)

    (b) Update Beta vector as $\beta_i = \beta_{i-1} + \alpha * p$
    (c) Update loglikelihood function using new betas: $logl_i = logl(X, Y, \beta_i, m)$
    (d) Update gradient using new betas: $g_i = gradient(X, Y, \beta_i, m)$

    (e) Update inverse Hessian approximation B:

        (i) First, update values required for calculation:
            $s = \alpha * p$ (This is $s_k$ in article.)
            $z = g_i - g_{i-1}$ (This is $y_k$ in article.)
            $rho = \frac{1}{z's}$ (This is $\rho_k$ in article.)
            $tau = rho * (s\% * \%z')$ (Saving a frequently used term as its own var, for simplicity.)
            $I = pxp$ identity matrix.

        (ii) Second, update inverse Hessian approximation using BFGS formula:
            $B_i = (I - tau)\% * \%B_{i-1}\% * \%(I - tau') + rho * ss'$

    (f) Check if convergence is met. If no, increment $i$ and repeat loop. If yes, end loop.

3. Once loop has ended, return vector of $\beta_i$ estimates.
```

Part B

Results of the Quasi-Newton Method with backtracking line search are as follows. The algorithm converged in 40 iterations.



```
> #3. Eyeball values for accuracy & display convergence.
> beta              #Glm estimated beta values.
         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_hat$beta_hat  #Gradient descent estimated beta values.
 [1]  0.4869994 -7.2217221  1.6547630 -1.7375649 14.0046594  1.0749477 -0.0772729  0.6751084
 [9]  2.5929000  0.4462523 -0.4824481
```

Figure 3: Quasi-Newton Negative Log-Likelihood Plots

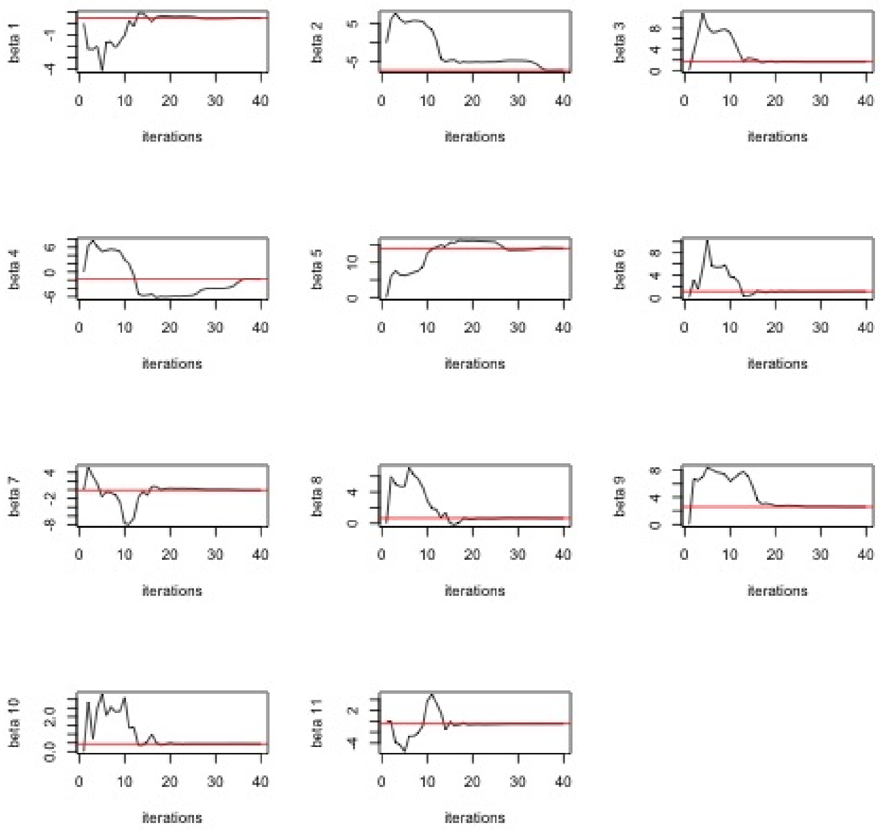


Figure 4: Quasi-Newton Betas Convergence

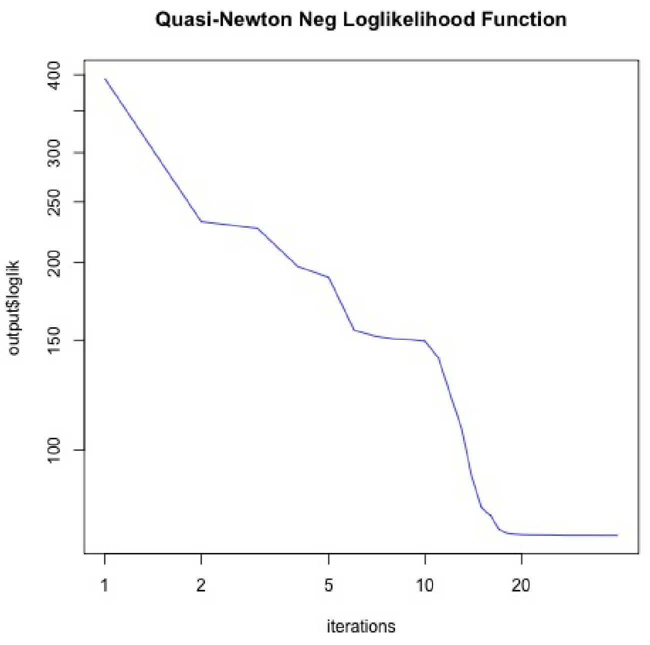


Figure 5: Quasi-Newton Negative Log-Likelihood Plot

R Code: Quasi-Newton with Backtracking Line Search

```r
### SDS 385 - Exercises 03 - Backtracking Line Search
#This code implements gradient descent to estimate the
#beta coefficients for binomial logistic regression.
#It uses backtracking line search to calculate the step size.

#Jennifer Starling
#26 August 2016

library(Matrix)
rm(list=ls())   #Clean workspace.

#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)
}

#-------------------------------------------------------------------
#Line Search Function
    #Inputs:  X = design matrix
    #         Y = vector of 1/0 response values
    #         b = vector of betas
    #         g = gradient for beta vector
    #         p = direction vector
    #         m = sample size vector m
    #         maxalpha = The maximum allowed step size.
    #Outputs: alpha = The multiple of the search direction.

linesearch <- function(X,Y,b,gr,p,m,maxalpha=1){
    c <- .01            #A constant, in (0,1)
    alpha <- maxalpha   #The max step size, ie the starting step size.
    rho <- .5               #The multiplier for the step size at each iteration.

    while( (logl(X,Y,b + alpha*p,m)) > logl(X,Y,b,m) + c*alpha*t(gr) %*% p ) {
        alpha <- rho*alpha
    }

    return(alpha)
}

#-------------------------------------------------------------------
#Quasi Newton with Backtracking Line Search Algorithm:
#Inputs:
#   X: n x p design matrix.
#   Y: response vector length n.
#   m: vector length n.
#   conv: Tolerance level for evaluating convergence.
#   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.

quasi_newton <- function(X,Y,m,maxiter=50000,conv=1*10^-10){

    converged <- 0      #Indicator for whether convergence met.

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

    #2. Initialize values for log-likelihood.
    loglik <- rep(0,maxiter)    #Initialize vector to hold loglikelihood fctn.
    loglik[1] <- logl(X,Y,betas[1,],m)

    #3. Initialize matrix to hold gradients for each iteration.
    grad <- matrix(0,nrow=maxiter,ncol=ncol(X))
    grad[1,] <- gradient(X,Y,betas[1,],m)
    #4. Initialize list of approximations of Hessian inverse, B.
    #   (Use identity matrix as initial value.)
    B <- list()
    B[[1]] <- diag(ncol(betas))

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

        #Compute direction and step size for beta update.
        p <- -B[[i-1]] %*% grad[i-1,]
        alpha <- linesearch(X,Y,b=betas[i-1,],gr=grad[i-1,],p,m,maxalpha=1)

        #Update beta values based on step/direction.
        betas[i,] <- betas[i-1,] + alpha*p

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

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

        #Update values needed for BFGS Hessian inverse approximation.
        s <- alpha*p
        z <- grad[i,] - grad[i-1,]
        rho <- as.vector(1/(t(z) %*% s))      #as.vector to make rho a scalar.
        tau <- rho * s %*% t(z) #Just breaking up the formula a bit for ease.
        I <- diag(ncol(grad))

        #BFGS formula for updating approx of H inverse.
        B[[i]] <- (I-tau) %*% B[[i-1]] %*% (I-t(tau)) + rho * s %*% t(s)

        print(i)

        #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(betas=betas[1:i,],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.
output <- quasi_newton(X,Y,m,maxiter=10000,conv=1*10^-10)

#3. Eyeball values for accuracy & display convergence.
beta                    #Glm estimated beta values.
output$beta_hat #Gradient descent estimated beta values.
```

```
#Print whether the algorithm has converged, and the number of iterations.
if(output$converged>0){cat('Algorithm converged in',output$iter, 'iterations.')}
if(output$converged<1){cat('Algorithm did not converge. Ran for max iterations.')}

#4. Plot the convergence of the beta variables compared to glm.
par(mfrow=c(4,3))
for (j in 1:length(output$beta_hat)){
    plot(1:nrow(output$betas),output$betas[,j],type='l',xlab='iterations',ylab=
        paste('beta',j))
    abline(h=beta[j],col='red')
}

#5. Plot log-likelihood function for convergence.
plot(1:length(output$loglik),output$loglik,type='l',xlab='iterations',col='blue',
   log='xy',
    main='Quasi-Newton Neg Loglikelihood Function')

#6. Save plots.
jpeg(file='/Users/jennstarling/UTAustin/2016_Fall_SDS 385_Stats Models for Big
   Data/Exercise 03 LaTeX Files/Ex03_quasinewtonloglhood.jpg')
plot(1:length(output$loglik),output$loglik,type='l',xlab='iterations',col='blue',
   log='xy',
    main='Quasi-Newton Neg Loglikelihood Function')
dev.off()

jpeg(file='/Users/jennstarling/UTAustin/2016_Fall_SDS 385_Stats Models for Big
   Data/Exercise 03 LaTeX Files/Ex03_quasinewtonbetas.jpg')
par(mfrow=c(4,3))
for (j in 1:length(output$beta_hat)){
    plot(1:nrow(output$betas),output$betas[,j],type='l',xlab='iterations',ylab=
        paste('beta',j))
    abline(h=beta[j],col='red')
}

dev.off()
```