Problem 1 Backtracking Line Search

Each iteration of line search is given by

$$x_{k+1} = x_k + \alpha_k p_k$$

where the positive scalar α_k is called step length. Most line search algorithms requires p_k to be a descent direction—one for which $p_k^T \nabla f_k < 0$ --because this property guarantees that the function *f* can be reduced along this direction. Moreover, the search direction often has the form

$$p_k = -B_k^{-1} \nabla f_k$$

where B_k is a symmetric and nonsingular matrix. In the steepest descent method, B_k simply the identity matrix I, while in Newton's method, is the exact Hessian $\nabla^2 f(x_k)$.

A simple condition we would impose on α_k is to require a reduction in *f*, that is,

$$f(x_k + \alpha_k p_k) < f(x_k).$$

This requirement is not enough to produce convergence because of insufficient reduction in *f* at each step. To avoid this behavior we need to enforce a *sufficient decrease* condition.

(A) The Wolfe Conditions

The Wolfe condition 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 sufficiently.

(a) The first condition is called *Armijo condition.* It stipulates α_k should first of all give *sufficient decrease* in the objective function *f*, as measured by the following inequality:

$$f(x_k + \alpha p_k) \le f(x_k) + c_1 \alpha \nabla f_k^T p_k$$

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

$$l(\beta_k - \alpha \nabla l(\beta_k)) \le l(\beta_k) - c_1 \alpha \| \nabla l(\beta_k) \|^2$$

c_1 is a constant with $c_1 \in (0,1)$. c_1 is usually small and 10^{-4} is given as an example. In other words, the reduction in *f* should be proportional to both the step length α_k and the directional derivative $\nabla f_k^T p_k$. 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 that are α's not too small, otherwise the algorithm will take a long time to converge. A second condition is required.

(b) The second condition *curvature condition*. It ensures that step sizes are not too short.

$$\nabla f(x_k + \alpha_k p_k)^T p_k \geq c_2 \nabla f_k^T p_k \text{, with } c_2 \in (0,1)$$

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

$$-\nabla f(x_k - \alpha_k \nabla l(\beta_k))^T \cdot \nabla l(\beta_k) \geq -c_2 \nabla f_k^T \cdot \nabla l(\beta_k)$$

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.

Note: about Goldstein conditions, see page 36 of textbook written by Nocedal and Wright.

Backtracking line search

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

(1) Choose $\bar{\alpha} > 0$. 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 $\rho \in (0,1)$, I chose $\rho = 0.5$. The ρ value represents what α is multiplied by at each iteration.

Choose a constant $c \in (0,1)$. I chose c = 0.001, it usually very small.

(2) Set $\alpha \leftarrow \bar{\alpha}$

Repeat until $f(x_k + \alpha p_k) \leq f(x_k) + c\alpha \nabla f_k^T p_k$, update $\alpha \leftarrow \rho\alpha$

In our case, $l(\beta_k - \alpha \nabla l(\beta_k)) \leq l(\beta_k) - c\alpha \| \nabla l(\beta_k) \|^2$

(3) End (repeat), terminate with $\alpha_k = \alpha$, as the step size for the gradient descent iteration.

(B) Now implement backtracking line search as part of batch gradient-descent code, and apply it to fit the logit model to the data sets (simulated or real). Compare its performance with some of earlier fixed choices of step size. Below are the results of the gradient descent with back tracking line search as follows:

coefficients	**X**	**XV3**	**XV4**	**XV5**	**XV6**	**XV7**	**XV8**	**XV9**	**XV10**	**XV11**	**XV12**
GLM	0.49	-7.22	1.65	-1.74	14.00	1.07	-0.08	0.68	2.59	0.45	-0.48
GD	0.47	-6.25	1.66	-2.63	13.86	1.07	-0.02	0.68	2.61	0.44	-0.49

The backtracking line search algorithm converged in 8476 iterations. It is much faster than Gradient Descent algorithm with fixed step size. The conditions on the step size give sufficient reduction in negative log likelihood function. Figure 1 shows that the negative log likelihood function decreases dramatically within 100 iterations.

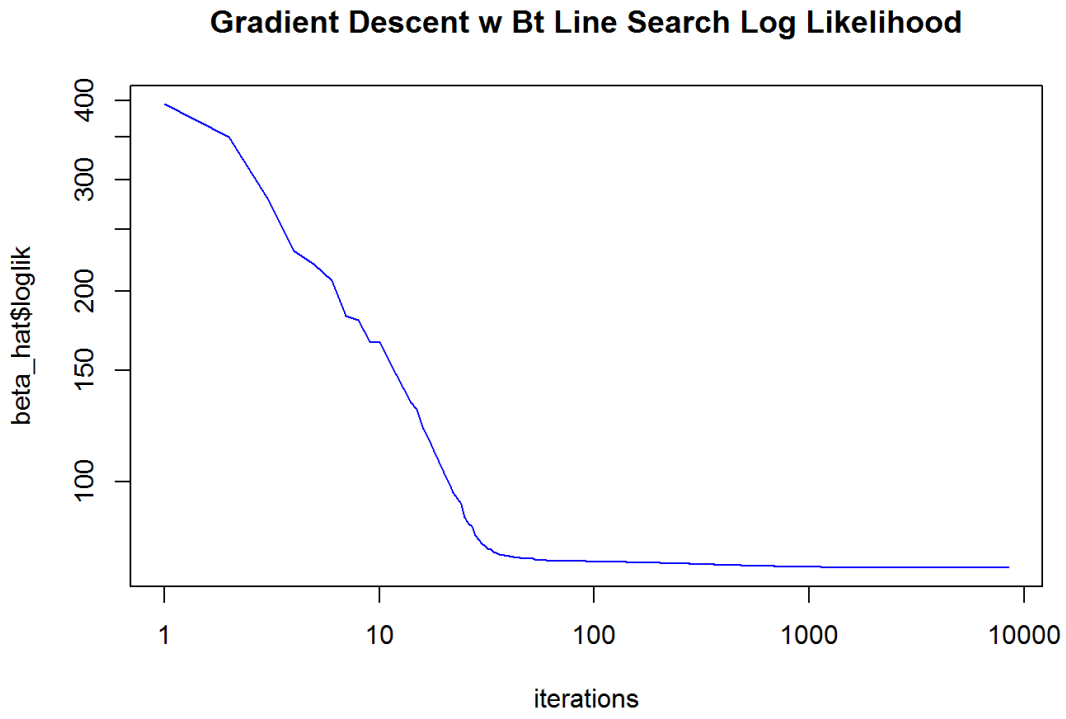


Figure 1: Gradient descent negative log likelihood function vs. number of iteration

Problem 2 Quasi-Newton Method

(A) *Secant Condition* and BFGS

A Quasi-Newton method uses an approximate Hessian, rather than the full Hessian, to compute the search direction and yet still attains a superlinear rate of convergence. Here the step size $\gamma = 1$. The BFGS provides a method to approximate the Hessian which ensures that the approximation B_k shares certain properties with the Hessian. The updates make use of the fact that changes in the gradient g provide information about the second derivative of f along the search direction.

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

$$B_{k+1} s_k = y_k$$

where $s_k = x_{k+1} - x_k$, $y_k = \nabla f_{k+1} - \nabla f_k$

This above *secant equation* can be written as

$$B_{k+1} = \frac{y_k}{s_k} = \frac{\nabla f_{k+1} - \nabla f_k}{x_{k+1} - x_k}$$

The intuition here is that the Hessian matrix *H* is the second derivative of the negative log-likelihood, i.e., 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 negative 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.

Typically, we impose additional conditions on B_{k+1}, such as symmetry (motivated by symmetry of the exact Hessian), and a requirement that the difference between successive approximations B_k and B_{k+1} have low rank. Two of the most popular formulae for updating the Hessian approximation B_k are the *symmetric-rank-one* (SR1) formula, defined by

$$B_{k+1} = B_k + \frac{(y_k - B_k s_k)(y_k - B_k s_k)^T}{(y_k - B_k s_k)^T s_k}$$

and the BFGS formula, named after its inventors, Broyden, Fletcher, Goldfarb, and Shanno, which is defined by

$$B_{k+1} = B_k - \frac{B_k s_k s_k^T B_k}{s_k^T B_k s_k} + \frac{y_k y_k^T}{y_k^T s_k}$$

Both updates satisfy the *secant equation* and both maintain symmetry. We can show that BFGS update generates positive definite approximations whenever the initial approximation B_0 is positive definite and $s_k^T y_k > 0$. Here we actually approximates 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.

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

(1) Initialize first iteration of values (*i*=1)
   i) Set first beta vector (β_1) to all zeroes;
   ii) Calculate negative log likelihood ($loglike_1$) function using β_1;
   iii) Calculate gradient using β_1;
   iv) Set identity matrix as the initial inverse Hessian approximation.

(2) Begin looping through, *i*=2 to *maxiter*
   i) Compute search direction and step size for updating beta vector (β_i), here *p* is the search direction, α is the step size;

   $p = -B_{i-1} g_{i-1}$ α = backtracking line search step length.

ii) Update beta vector as $\beta_i = \beta_{i-1} + \alpha * p$

Update log likelihood function using new betas: $\log like_i = \log like(y, X, \beta_i, m)$

Update gradient using new betas: $g_i = gradient(y, X, \beta_i, m)$;

iii) Update the inverse Hessian approximation *B*:

(a) Update values required for calculation:

$$s_k: \quad s = \alpha * p \qquad y_k: \quad z = g_i - g_{i-1}$$

$$\rho_k: \quad rho = \frac{1}{z^T s} \qquad tau = rho * (s\%*\%z^T) \qquad I = identity\ matrix$$

(b) Check if convergence is met. If no, increment *i* and repeat loop. If yes, end loop;

iv) Once loop has ended, return vector of β_i estimates.

(B) Implement BFGS coupled with backtracking line search to fit the logit model to the data sets. Compare its performance both with newton's method and with the batch gradient descent, in terms of the overall steps required. Below are the estimated β_i obtained from GLM and Quasi-Newton method. The coefficients comparison between GLM and Quasi-Newton algorithm is shown in Figure 3. Here I chose *c* = 0.1. The algorithm converged in 35 iterations.

coefficients	**X**	**XV3**	**XV4**	**XV5**	**XV6**	**XV7**	**XV8**	**XV9**	**XV10**	**XV11**	**XV12**
GLM	0.49	-7.22	1.65	-1.74	14.00	1.07	-0.08	0.68	2.59	0.45	-0.48
QN	0.49	-7.24	1.66	-1.73	14.02	1.08	-0.08	0.67	2.59	0.45	-0.48

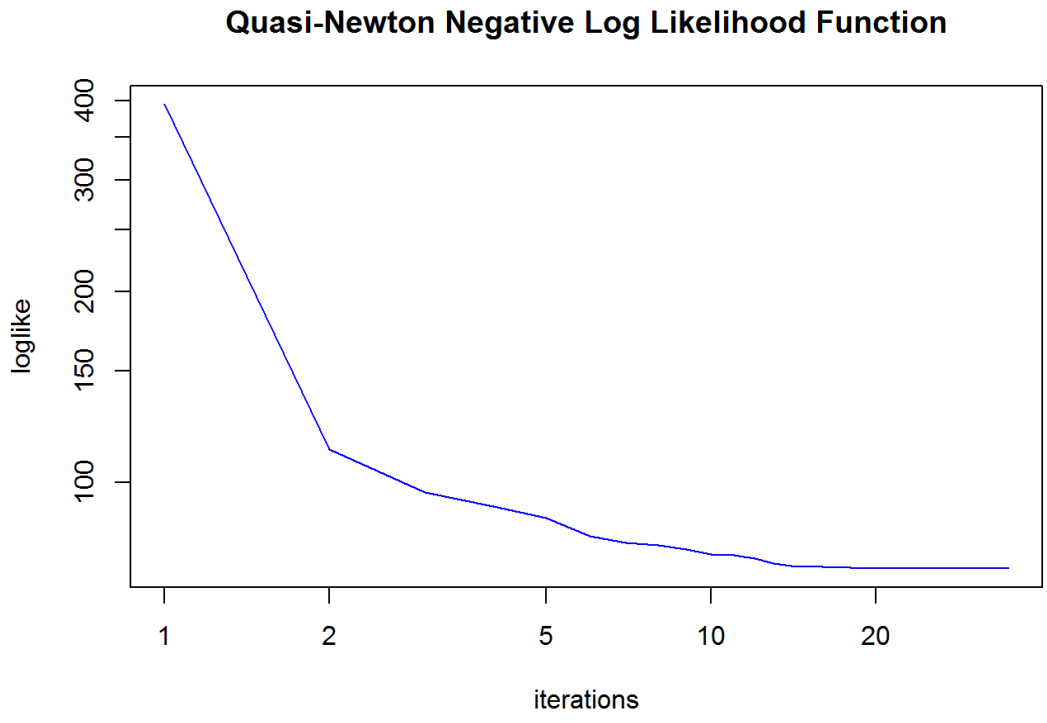


Figure 2: Quasi-Newton negative log likelihood function vs. number of iterations

Quasi-Newton method requires less steps (⩽35 steps) than batch gradient descent and a bit more steps than Newton methods (⩽10 steps). But it is more computationally efficient than Newton methods because it computes Hessian approximate instead of inverse Hessian matrix.

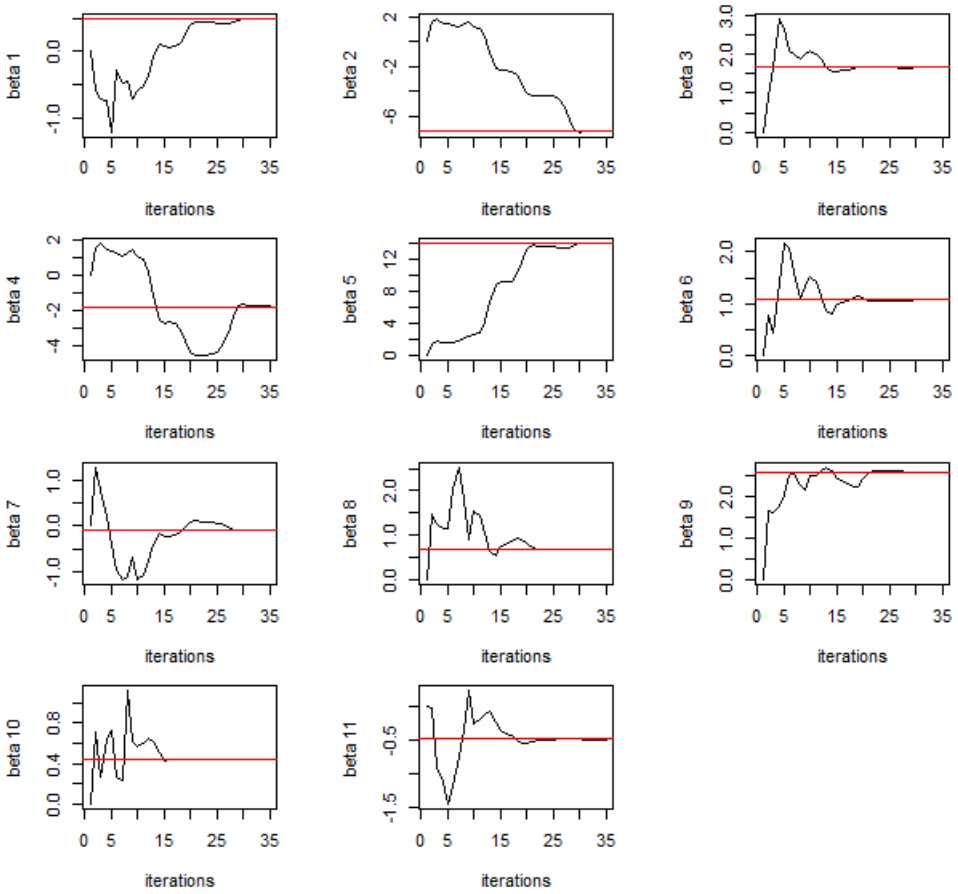


Figure 3: coefficients comparison between GLM and Quasi-Newton algorithm

exercise03_Backtracking_line_search.R

Shuchen

Fri Oct 20 14:02:04 2017

```
### SDS 385 - Exercises 03 - Backtracking Line Search
# Some of the code is based on the R code of Jennifer Starling
# Jennifer Starling: https://github.com/jestarling/bigdata/tree/master/Exercise-03

# This code implements gradient descent to estimate the
# beta coefficients for binomial logistic regression.
# It uses backtracking line search to calculate the step size.

# Yanxin Li   Oct 2 2017

rm(list=ls())
library(Matrix)
set.seed(1) # Reproducibility
```

```
# Read in data
wdbc <- read.csv("https://raw.githubusercontent.com/jgscott/SDS385/master/data/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 <- rep(0,length(Y))
y[Y=='M'] <- 1 # Convert y values to 1/0's, response vector

beta <- as.matrix(rep (0, ncol(X))) # Initial guess of beta
m <- 1 # Number of trials is 1
```

```
# Sigmoid function
sigmoid <- function(beta, X){
  w <- 1/(1+exp(-X %*% beta))
  w <- pmax(w, 1E-5); w <- pmin(w, 1-1E-5) # Avoid probability of 0 or 1
  return(w)
}
```

```
# Negative log likelihood function
loglike <- function(beta, y, X, m){
  # Input: beta: regression parameter P x 1
  #        y: vector of response N x 1
  #        X: matrix of features N x P
  #        m: number of trial for the ith case
  # Output:Negative log likelihood of binomial distribution

  loglik <- -sum(dbinom(y, m, sigmoid(beta, X), log = TRUE))
  return(loglik)
}
```

```r
# Gradient of the negative log likelihood
gradient <- function(beta, y, X, m){
  grad <- as.numeric(y - m * sigmoid(beta, X)) * X
  return(-colSums(grad))
}
```

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

```r
# Line Search Function
# Input:  X: design matrix
#             y: vector of 1/0 response values
#             beta: vector of betas
#           gr: gradient for beta vector
#         m: sample size vector m
#         maxalpha: the maximum allowed step size
# Output: alpha: the multiple of the search direction
linesearch <- function(beta, y, X, gr, m, maxalpha=1){
  c <- 0.001            # a constant, in (0,1)
  alpha <- maxalpha # the max step size, ie the starting step size
  p <- 0.5              # the multiplier for the step size at each iteration

  # Update alpha while line search condition holds
  while((loglike(beta - alpha*gr,y,X,m))>loglike(beta,y,X,m)-c*alpha*norm_vec(gr)^2){
    alpha <- p*alpha
  }
  return(alpha)
}
```

```r
# Gradient Descent Algorithm:
# Input:  X: n x p design matrix
#           y: response vector length n
#         conv: Tolerance level for determining convergence
#           a: Step size
# Output: 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=1E-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] <- loglike(betas[1,], y, X, m)

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

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

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

    # Backtracking line search to calculate step size
    step <- linesearch(beta=betas[i-1,],y,X,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] <- loglike(betas[i,],y,X,m)

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

    iter <- i   # Track iterations

    # check if convergence met: If yes, exit loop
    if (abs(loglik[i]-loglik[i-1])/(abs(loglik[i-1])+conv) < 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
```

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

```
beta <- glm1$coefficients

# 2. Call gradient descent function to estimate beta_hat
beta_hat <- gradient_descent(X,y,m,maxiter=50000,conv=1E-10)

# 3. Eyeball values for accuracy & display convergence.
round(beta,2)                    # glm estimated beta values
```

```
##     X   XV3   XV4   XV5   XV6   XV7   XV8   XV9  XV10  XV11  XV12
##  0.49 -7.22  1.65 -1.74 14.00  1.07 -0.08  0.68  2.59  0.45 -0.48
```

```
round(beta_hat$beta_hat,2)  # Gradient descent estimated beta values
```

```
##  [1]  0.47 -6.25  1.66 -2.63 13.86  1.07 -0.02  0.68  2.61  0.44 -0.49
```

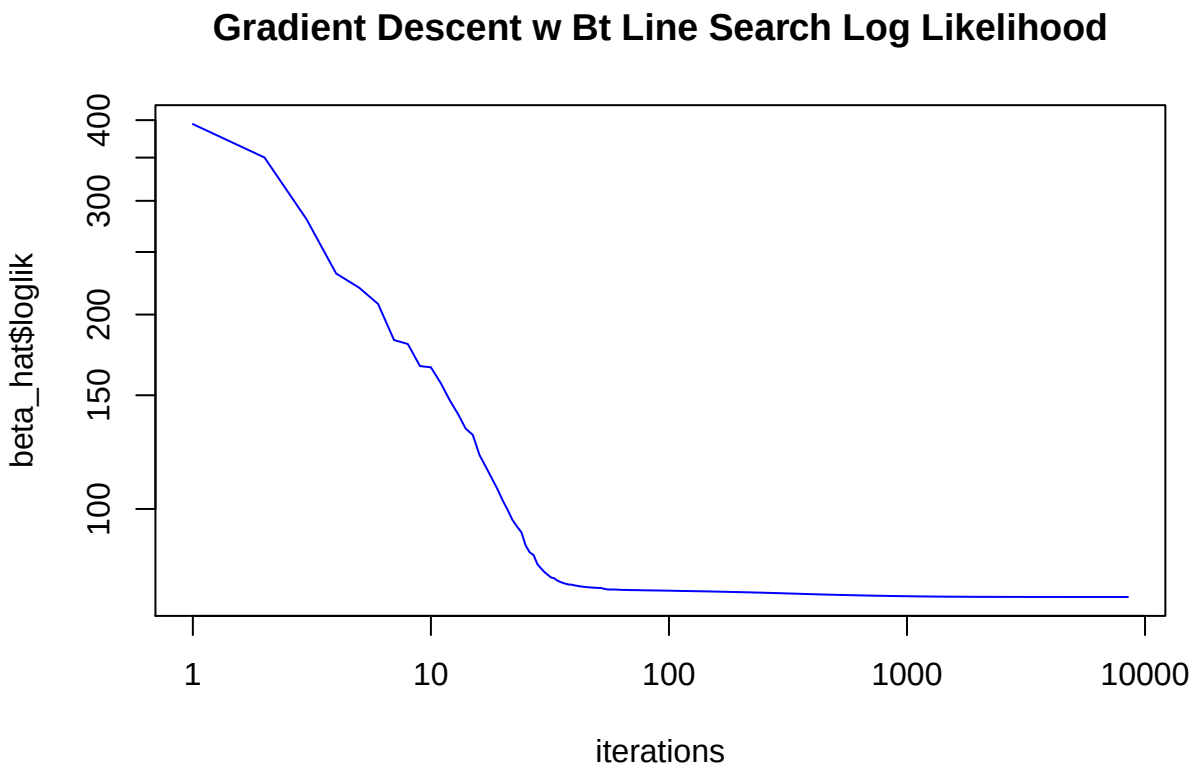
```
print(c("Algorithm converged?",beta_hat$converged,"(1=converged,0=did not converge)"))
```

```
## [1] "Algorithm converged?"             "1"
## [3] "(1=converged,0=did not converge)"
```

```
print(beta_hat$iter)
```

```
## [1] 8476
```

```r
# 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 Log Likelihood')
```



exercise03_Quasi-Newton.R

Shuchen

Fri Oct 20 14:03:38 2017

```
### SDS 385 - Exercises 03 - Quasi-Newton
# Some of the code is based on the R code of Jennifer Starling
# Jennifer Starling: https://github.com/jestarling/bigdata/tree/master/Exercise-03

# This code implements Quasi-Newton algorithm to estimate the
# beta coefficients for binomial logistic regression.

# Yanxin Li   Oct 2 2017

rm(list=ls())
library(Matrix)
set.seed(1) # Reproducibility
```

```
# Read in data
wdbc <- read.csv("https://raw.githubusercontent.com/jgscott/SDS385/master/data/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 <- rep(0,length(Y))
y[Y=='M'] <- 1 # Convert y values to 1/0's, response vector

beta <- as.matrix(rep (0, ncol(X))) # Initial guess of beta
m <- 1 # Number of trials is 1
```

```
# Sigmoid function
sigmoid <- function(beta, X){
  w <- 1/(1+exp(-X %*% beta))
  w <- pmax(w, 1E-5); w <- pmin(w, 1-1E-5) # Avoid probability of 0 or 1
  return(w)
}
```

```
# Negative log likelihood function
loglike <- function(beta, y, X, m){
  # Input: beta: regression parameter P x 1
  #        y: vector of response N x 1
  #        X: matrix of features N x P
  #        m: number of trial for the ith case
  # Output:Negative log likelihood of binomial distribution

  loglik <- -sum(dbinom(y, m, sigmoid(beta, X), log = TRUE))
  return(loglik)
}
```

```r
# Gradient of the negative log likelihood
gradient <- function(beta, y, X, m){
  grad <- as.numeric(y - m * sigmoid(beta, X)) * X
  return(-colSums(grad))
}
```

```r
# Line Search Function: p is the direction vector
line.search <- function(beta,y,X,gr,p,m,maxalpha=1){
  c <- 0.1          # a constant, in (0,1)
  alpha <- maxalpha # max step size, ie the starting step size
  rho <- 0.5                # the multiplier for the step size at each iteration

  while((loglike(beta+alpha*p,y,X,m)) > loglike(beta,y,X,m) + c*alpha*t(gr)%*%p){
    alpha <- rho*alpha
  }
  return(alpha)
}
```

```r
# Quasi-Newton with Backtracking Line Search Algorithm
# Inputs:  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: negative log-likelihood

quasi_newton <- function(X,y,m,maxiter=50000,conv=1E-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

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

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

  # 4. Initialize list of approximations of Hessian inverse B
  B <- list()
  B[[1]] <- diag(ncol(betas)) # use identity matrix as initial value

  # 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 <- line.search(betas[i-1,],y,X,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] <- loglike(betas[i,],y,X,m)

    # Calculate gradient for new betas
    grad[i,] <- gradient(betas[i,],y,X,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)
    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])+conv) < 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.)
glm <- glm(y~X-1, family='binomial') # fits model, obtains beta values
```

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

```
beta <- glm$coefficients

# 2. Call Quasi-Newton function to estimate
output <- quasi_newton(X,y,m,maxiter=10000,conv=1E-10)

# 3. Eyeball values for accuracy & display convergence
round(beta,2)               # glm estimated beta values
```

```
##     X   XV3   XV4   XV5   XV6   XV7   XV8   XV9  XV10  XV11  XV12
##  0.49 -7.22  1.65 -1.74 14.00  1.07 -0.08  0.68  2.59  0.45 -0.48
```

```
round(output$beta_hat,2)    # gradient descent estimated beta values
```

```
##  [1]  0.49 -7.24  1.66 -1.73 14.02  1.08 -0.08  0.67  2.59  0.45 -0.48
```

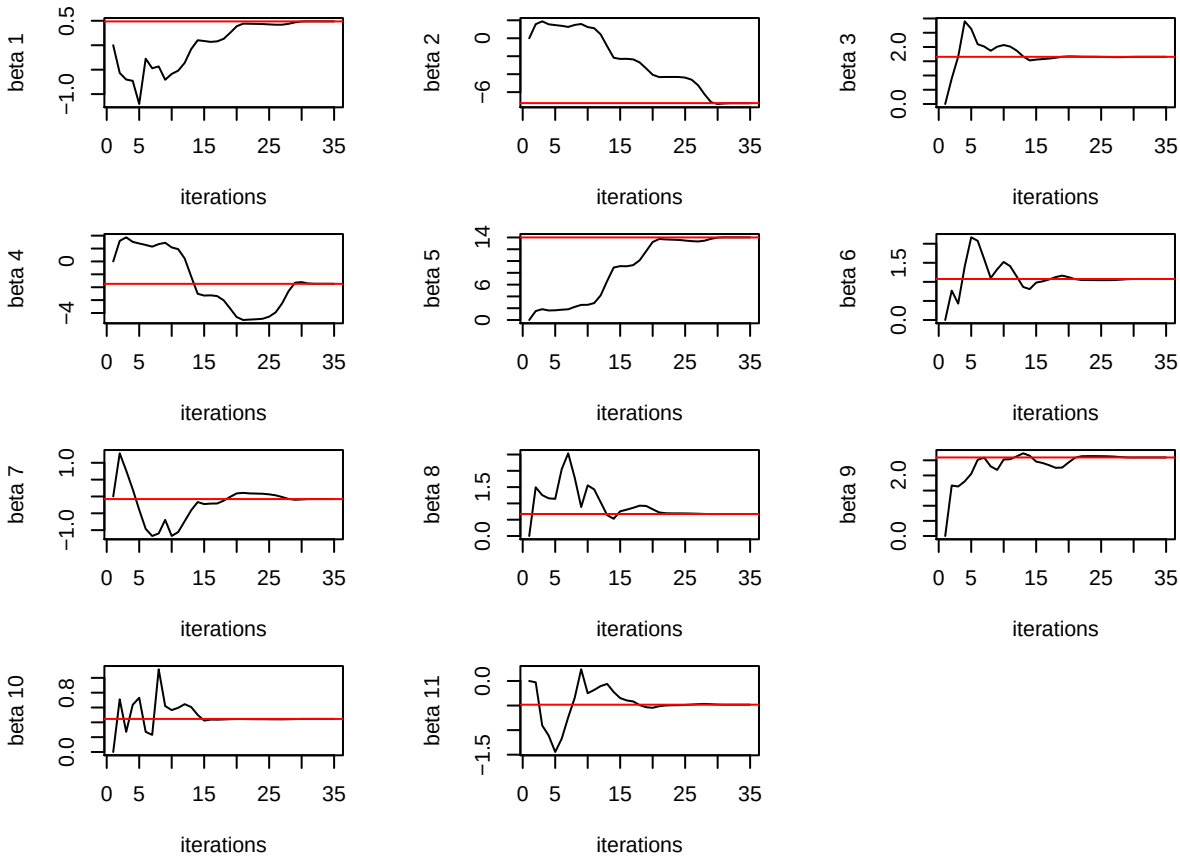
```
# Check whether the algorithm has converged, and the number of iterations
if(output$converged>0){cat('Algorithm converged in',output$iter, 'iterations.')}
```

```
## Algorithm converged in 35 iterations.
```

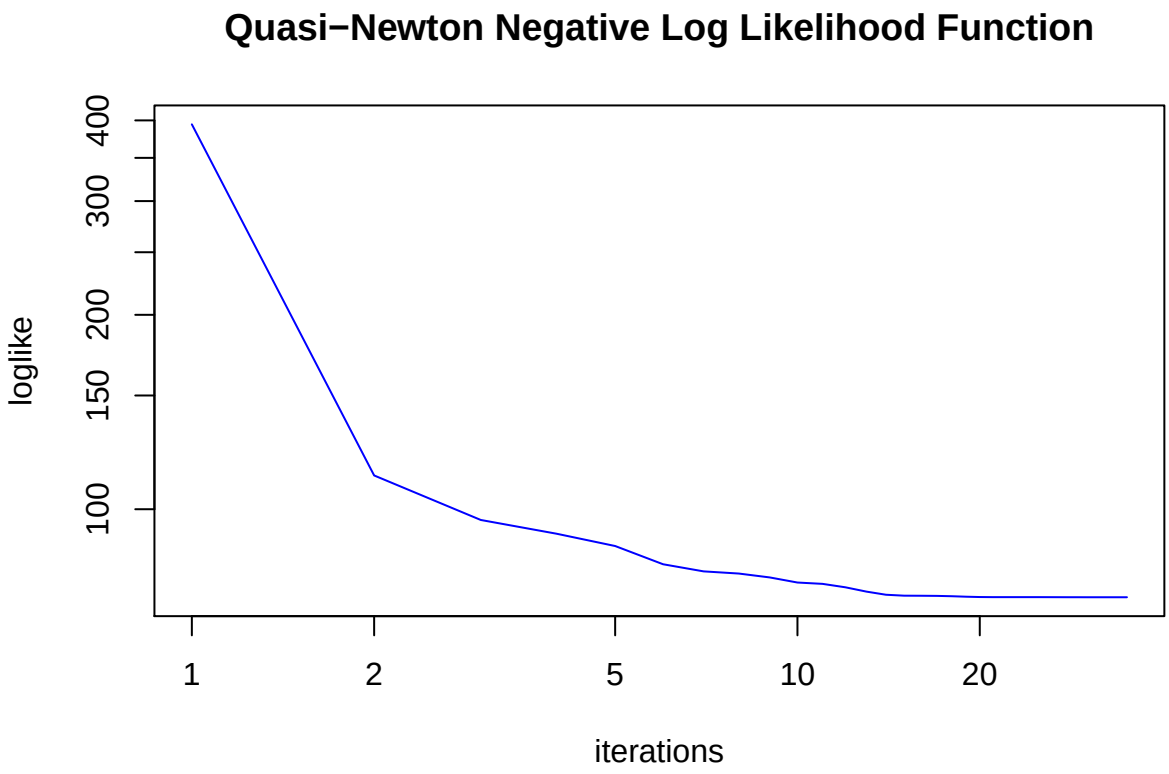
```
if(output$converged<1){cat('Algorithm did not converge. Ran for max iterations.')}

# 4. Plot the convergence of the beta variables compared to glm
par(mfrow=c(4,3))
par(mar=c(4,4,1,3))
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
par(mar=c(5.1,4.1,4.1,2.1))
par(mfrow=c(1,1))
```




```
plot(1:length(output$loglik),output$loglik,type='l',xlab='iterations',ylab='loglike',
     col='blue',log='xy',main='Quasi-Newton Negative Log Likelihood Function')
```


Quasi−Newton Negative Log Likelihood Function
loglike
100
150
200
300
400
1
2
5
10
20
iterations