STA6106 Statistic Computing Project2

Robert Norberg, Jung-Han Wang

Wednesday, October 22, 2014

## Problem1

The function g is given by

The goal is to find the minimum of g.

### a. Minimize g using Newton's method

The Newton's Algorithm is:

$$\pmb{x(n+1)=x(n)-H(x(n))^{-1}\bigtriangledown f(x(n))}$$

First step we need to verify that is non-singular.

hx<-D(D(expression(4\*x\*y+(x+y^2)^2), 'x'),'x');hx ## Check H(X(n)) is a non-singular matrix

[1] 2

Setting up stopping condition to be:

$$\pmb{\|\bigtriangledown f(x(n))\|} \le \epsilon $$

newton <- function(f3, x0, tol = 1e-9, n.max = 100) {  
# Newton's method starting at x0  
# f3 is a function that given x returns the list  
# f(x), f'(x), Hessian f''(x)  
x <- x0 ## Set initial value  
f3.x <- f3(x) ## Set Input Function  
n <- 0 ## Set first turn n<-0  
while ((max(abs(f3.x[[2]])) > tol) & (n < n.max)) { ##Set Convergence Criteria. If f'(x) greater than tol(tolerance) go to n+1.  
x <- x - solve(f3.x[[3]], f3.x[[2]]) ##Calculate f'(x)/f''(x)  
f3.x <- f3(x)   
n <- n + 1 ##Continue to the next n  
}  
if (n == n.max) { ##If n=maximum value, output "newton failed to converge"  
cat('newton failed to converge\n')  
} else {  
return(x)  
}  
}

Then Calculate first and second derivatives for function Using package (Ryacas).

library(Ryacas);  
k <- function(x) {  
4\*x[1]\*x[2]+(x[1]+x[2]^2)^2  
}  
##Calculate First Derivative of Function to X (f1)  
f1<-D(expression(4\*x\*y+(x+y^2)^2), 'x');f1

4 \* y + 2 \* (x + y^2)

##Calculate Second Derivative of Function to X (f11)  
f11<-D(D(expression(4\*x\*y+(x+y^2)^2), 'x'),'x');f11

[1] 2

##Calculate First Derivative of Function to y (f2)  
f2<-D(expression(4\*x\*y+(x+y^2)^2), 'y');f2

4 \* x + 2 \* (2 \* y \* (x + y^2))

##Calculate Second Derivative of Function to X (f22)  
f22<-D(D(expression(4\*x\*y+(x+y^2)^2), 'y'),'y');f22

2 \* (2 \* (x + y^2) + 2 \* y \* (2 \* y))

##Calculate Second Derivative of Function to X,y (f12)  
f12<-D(D(expression(4\*x\*y+(x+y^2)^2), 'x'),'y');f12

4 + 2 \* (2 \* y)

Set up function with return value using $\pmb{f}$(original function), $\pmb{\bigtriangledown f}$(first derivative function) and $\pmb{H(x(n))}$(second derivative function).

f3 <- function(x) {  
f <- 4\*x[1]\*x[2]+(x[1]+x[2]^2)^2  
##Calculate First Derivative of Function to x[1] (f1)  
f1<-4\*x[2]+2\*(x[1] + x[2]^2)  
##Calculate Second Derivative of Function to x[1] (f11)  
f11<-2  
##Calculate First Derivative of Function to x[2] (f2)  
f2<-4 \* x[1] + 2 \* (2 \* x[2] \* (x[1] + x[2]^2))  
##Calculate Second Derivative of Function to x[1] (f22)  
f22<-2 \* (2 \* (x[1] + x[2]^2) + 2 \* x[2] \* (2 \* x[2]))  
##Calculate Second Derivative of Function to x[1],x[2] (f12)  
f12<-4 + 2 \* (2 \* x[2])  
return(list(f, c(f1, f2), matrix(c(f11, f12, f12, f22), 2, 2))) ##Return 3 values, f, f'(x), f''(x)  
}

Run the program with start point (0,0) and try add 0.5 each run. Left side of the result represents the value of starting point. Right side of the result represent the value of the ending point.

for (x0 in seq(0,1, .5)) {  
for (y0 in seq(0,1, .5)) {  
cat(c(x0,y0), '--(Left=Start Point, Right=Extreme Value)--', newton(f3, c(x0,y0)), '\n')  
}}

0 0 --(Left=Start Point, Right=Extreme Value)-- 0 0   
0 0.5 --(Left=Start Point, Right=Extreme Value)-- -7.438748e-11 6.286889e-11   
0 1 --(Left=Start Point, Right=Extreme Value)-- 0 0   
0.5 0 --(Left=Start Point, Right=Extreme Value)-- 0 0   
0.5 0.5 --(Left=Start Point, Right=Extreme Value)-- -7.240532e-16 6.847439e-16   
0.5 1 --(Left=Start Point, Right=Extreme Value)-- -4.225566e-14 4.196096e-14   
1 0 --(Left=Start Point, Right=Extreme Value)-- 0 0   
1 0.5 --(Left=Start Point, Right=Extreme Value)-- -1.3112e-14 1.293993e-14   
1 1 --(Left=Start Point, Right=Extreme Value)-- 0.8888889 -0.6666667

### b. Minimize g using the steepest descent method. Use (1,0) as starting point.

Define function using Golden-Section method to find

such that

gsection = function(ftn, x.l, x.r, x.m, tol = 1e-9) {  
 # applies the golden-section algorithm to minimize ftn  
 # we assume that ftn is a function of a single variable  
 # and that x.l < x.m < x.r and ftn(x.l), ftn(x.r) >= ftn(x.m)  
   
 # the algorithm iteratively refines x.l, x.r, and x.m and terminates  
 # when x.r - x.l <= tol, then returns x.m  
   
 # golden ratio plus one  
 gr1 = 1 + (1 + sqrt(5))/2  
 # successively refine x.l, x.r, and x.m  
 f.l = ftn(x.l)  
 f.r = ftn(x.r)  
 f.m = ftn(x.m)  
 while ((x.r - x.l) > tol) {   
 if ((x.r - x.m) > (x.m - x.l)) {  
 y = x.m + (x.r - x.m)/gr1  
 f.y = ftn(y)  
 if (f.y <= f.m) {  
 x.l = x.m  
 f.l = f.m  
 x.m = y  
 f.m = f.y  
 } else {  
 x.r = y  
 f.r = f.y  
 }  
 } else {  
 y = x.m - (x.m - x.l)/gr1  
 f.y = ftn(y)  
 if (f.y <= f.m) {  
 x.r = x.m  
 f.r = f.m  
 x.m = y  
 f.m = f.y  
 } else {  
 x.l = y  
 f.l = f.y  
 }  
 }  
 }  
 return(x.m)  
}

Create Function F=

##Set Function F  
f <- function(x) {  
f <- 4\*x[1]\*x[2]+(x[1]+x[2]^2)^2  
 return(f)  
}

Set up first order partial derivative Function F for x and y.

##Set Function Gradient F f'(x)  
gradf<- function (x)  
 {##Calculate First Derivative of Function to x[1] (f1)  
 f1<-4\*x[2]+2\*(x[1] + x[2]^2)  
 ##Calculate First Derivative of Function to x[2] (f2)  
 f2<-4 \* x[1] + 2 \* (2 \* x[2] \* (x[1] + x[2]^2))  
 return(c(f1, f2))  
 }

Set up algorithm for gradient line search. Trying to find Minimum point between

line.search <- function(f, x, gradf, tol = 1e-9, a.max = 100) {  
# x and gradf are vectors of length d  
# g(a) =f(x +a\*gradf) hasa local minumum at a,  
# within a tolerance  
# if no local minimum is found then we use 0 or a.max for a  
# the value returned is x + a\*y  
if (sum(abs(gradf)) == 0) return(x) # g(a) constant  
g <- function(a) return(f(x - a\*gradf))  
  
# find a.l < a.m < a.r such that  
# g(a.m) >=g(a.l) and g(a.m) >= g(a.r)  
# a.l  
a.l <- 0  
g.l <- g(a.l)  
# a.m  
a.m <- 1  
g.m <- g(a.m)  
while ((g.m > g.l) & (a.m > tol)) {  
a.m <- a.m/2  
g.m <- g(a.m)  
}  
# if a suitable a.m was not found then use 0 for a  
if ((a.m <= tol) & (g.m >= g.l)) return(x)  
# a.r  
a.r <- 2\*a.m  
g.r <- g(a.r)  
while ((g.m >= g.r) & (a.r < a.max)) {  
a.m <- a.r  
g.m <- g.r  
a.r <- 2\*a.m  
g.r <- g(a.r)  
}  
# if a suitable a.r was not found then use a.max for a  
if ((a.r >= a.max) & (g.m > g.r)) return(x - a.max\*gradf)  
# apply golden-section algorithm to g to find a  
a <- gsection(g, a.l, a.r, a.m)  
return(x - a\*gradf)  
}

Set up iteration using Steepest Descent Algorithm.

descent <- function(f,gradf, x0, tol = 1e-9, n.max = 100) {  
# steepest descent algorithm  
# find a local minimum of f starting at x0  
# function gradf is the gradient of f  
x <- x0  
x.old <- x  
x <- line.search(f, x, gradf(x))  
n <- 1  
while (f(x.old)-(f(x)> tol) & (n < n.max)) {  
x.old <- x  
x <- line.search(f, x, gradf(x))  
n <- n + 1  
}  
return(x)  
}

The minimize result is

descent(f,gradf,c(1,0) )

[1] 0.8888889 -0.6666667

## Problem2

In 1986, the space shuttle Challenger exploded during takeoff, killing the seven astronauts aboard. The explosion was the result of an O-ring failure, a splitting of a ring of rubber that seals the parts of the ship together. The accident was believed to have been caused by the unusually cold weather ( or ) at the time of launch, as there is reason to believe that the O-ring failure probabilities increase as temperature decreases. Data on previous space shuttle launches and O-ring failures is given in the dataset challenger provided with the "mcsm" package of R. The first column corresponds to the failure indicators and the second column to the corresponding temperature , (1i 24).

# load challenger data  
library(mcsm)  
data(challenger)

### a) The goal is to obtain MLEs for and in the following logistic regression model

where is the probability that at least one O-ring is damaged and is the temperature. Create computer programs using Newton-Raphson algorithm to find MLEs of

The log-likelihood is given by

$$l(\beta) = \pmb{y}^T \pmb{Z \beta}-\pmb{b}^T \pmb{1}$$

where $\pmb{1}$ is a column vector of ones, $\pmb{y} = (y\_1...y\_n)^T$, $\pmb{b} = (b(\theta\_1)...b(\theta\_n))^T$, and $\pmb{Z}$ is the matrix whose th row is $\pmb{z}\_i^T$.

The score function is

$$\pmb{l}'(\pmb{\beta})=\pmb{Z}^T(\pmb{y}-\pmb{\pi})$$

where $\pmb{\pi}$ is a column vector of the Bernoulli probabilities . The Hessian is given by

$$\pmb{l}''(\pmb{\beta}) = \dfrac{d}{d \pmb{\beta}}(\pmb{Z}^T(\pmb{y}-\pmb{\pi})) = -\left( \dfrac{d \pmb{\pi}}{d \pmb{\beta}} \right)^T \pmb{Z} = -\pmb{Z}^T\pmb{WZ}$$

where $\pmb{W}$ is a diagonal matrix with th diagonal entry equal to .

Newton's update is therefore

$$\pmb{\beta}^{(t+1)} = \pmb{\beta}^{(t)}-\pmb{l}''(\pmb{\beta}^{(t)})^{-1}\pmb{l}'(\pmb{\beta}^{(t)}) = \pmb{\beta}^{(t)}+\left( \pmb{Z}^T\pmb{W}^{(t)}\pmb{Z}\right)^{-1}\left(\pmb{Z}^T(\pmb{y}-\pmb{\pi}^{(t)})\right)$$

where $\pmb{\pi}^{(t)}$ is the value of $\pmb{\pi}$ corresponding to $\pmb{\beta}^{(t)}$, and $\pmb{W}^{(t)}$ is the diagonal weight matrix evaluated at $\pmb{\pi}^{(t)}$.

To find MLEs of and we designate the vector $\pmb{y}$ to be the response vector in the data and $\pmb{Z}$ to be the matrix of ones and temperatures.

y <- challenger$oring  
x <- challenger$temp  
Z <- cbind(rep(1, nrow(challenger)), challenger$temp)

$\pmb{Z}$ will remain unchanged, but for each iteration we will need to compute the predicted probabilities using the current values of and . We will also need to compute for each iteration $\pmb{W}$, the diagonal matrix with diagonal entry equal to . Below, we define a function to do each of these computations.

find\_pis <- function(beta, x){  
 # beta a vector of parameter estimates, x a vector of independent variable observations  
 pis <- exp(beta%\*%t(Z))/(1+exp(beta%\*%t(Z)))  
 return(pis)  
}  
  
find\_W <- function(pis){  
 W <- diag(c(pis\*(1-pis)))  
 return(W)  
}

Next we difine a function to calculate Newton's update using $\pmb{Z}$, $\pmb{y}$, $\pmb{\beta}$, and the and $\pmb{W}$ calculated using the functions above.

find\_update <- function(beta, Z, W, y, pis){  
 numerator <- t(Z)%\*%(y-t(pis))  
 denominator <- t(Z)%\*%W%\*%Z  
 update <- solve(denominator)%\*%numerator  
 return(beta+c(update))  
}

Finally, we must define a stopping criterion so that the algorithm ceases to update at some point and returns to us values for and . We will stop updating our estimates when an iteration takes place that does not change them. We are not concerned with accuracy at more than seven decimal places, but we will write the program so that this may be changed.

We choose startting values for and to be , as is reccomended in the textbook. Then we begin a while loop to update these estimates until convergence occurs.

beta <- c(0, 0)  
old\_beta <- c(NA, NA)  
counter <- 0  
decimal\_places\_of\_precision <- 7  
verbose <- TRUE  
  
while(!identical(beta, old\_beta)){  
 pis <- find\_pis(beta, x)  
 W <- find\_W(pis)  
 # set old beta equal to current beta before updating  
 old\_beta <- beta  
 # replace beta with updated estimates  
 beta <- find\_update(beta, Z, W, y, pis)  
 # round to the number of decimal places desired  
 beta <- round(beta, decimal\_places\_of\_precision)  
 counter <- counter+1  
 # show us the current value of beta if "verbose" is TRUE  
 if(verbose==T){  
 msg <- paste0('Iteration ', counter, ': Beta = (', beta[1], ', ', beta[2], ')')  
 cat(msg)  
 cat('\n')  
 }  
}

Iteration 1: Beta = (9.6190476, -0.1495238)  
Iteration 2: Beta = (13.6557372, -0.211247)  
Iteration 3: Beta = (14.9382896, -0.2306001)  
Iteration 4: Beta = (15.0422911, -0.2321537)  
Iteration 5: Beta = (15.0429016, -0.2321627)  
Iteration 6: Beta = (15.0429016, -0.2321627)

Now that the while loop has terminated, the current values of beta should be the MLEs of and . We check these against the estimates given using the glm() function in R.

# see our estimates  
print(beta)

[1] 15.0429016 -0.2321627

# see the glm estimates  
glm(y~x, family=binomial)$coefficients

(Intercept) x   
 15.0429016 -0.2321627

Our estimates match those produced by the glm() function, so we are confident that we have correctly found the MLEs of and .

### b) Solve the same problem using the "Iterative Reweighted Least Squares" algorithm and the Newton-Raphson algorithm to find MLEs of

If we let

$$\pmb{e}^{(t)} = \pmb{y}-\pmb{\pi}^{(t)}$$

and

$$\pmb{x}^{(t)} = \pmb{Z\beta}^{(t)} + (\pmb{W}^{(t)})^{-1}\pmb{e}^{(t)}$$

then the Fisher scoring update can be written as

$$\pmb{\beta}^{(t+1)} = \pmb{\beta}+\left( \pmb{Z}^T\pmb{W}^{(t)}\pmb{Z} \right)^{-1} \pmb{Z}^T\pmb{e}^{(t)}$$

$$ = \left( \pmb{Z}^T\pmb{W}^{(t)}\pmb{Z}\right)^{-1}\pmb{Z}^T\pmb{W}^{(t)}\pmb{x}^{(t)}$$

where $\pmb{x}^{(t)}$ is the *working response* that gets updated with eac iteration.

If we reuse the functions defined earlier to calculate $\pmb{\pi}$ and $\pmb{W}$, we just need to define a function for calculating $\pmb{x}$ and define a new function for updating using the iteratively reweighted least squares update before we begin to iterate. We define a function for calculating $\pmb{x}$ as follows:

find\_x <- function(y, beta, pis, Z, W){  
 e\_t <- y-c(pis)  
 x <- Z%\*%beta+solve(W)%\*%e\_t  
 return(x)  
}

And a new updating function:

find\_update\_irls <- function(Z, W, working\_response){  
 numerator <- t(Z)%\*%W%\*%working\_response  
 denominator <- t(Z)%\*%W%\*%Z  
 update <- solve(denominator)%\*%numerator  
}

Next we choose for the initial values of $\pmb{\beta}$ and begin another while loop, this time updating using the iteratively reweighted least squares update. (We are still satisfied with 7 decimal places of accuracy.)

beta <- c(0, 0)  
old\_beta <- c(NA, NA)  
counter <- 0  
decimal\_places\_of\_precision <- 7  
verbose <- TRUE  
  
while(!identical(beta, old\_beta)){  
 pis <- find\_pis(beta, x)  
 W <- find\_W(pis)  
 working\_response <- find\_x(y, beta, pis, Z, W)  
 # set old beta equal to current beta before updating  
 old\_beta <- beta  
 # replace beta with updated estimates  
 beta <- find\_update\_irls(Z, W, working\_response)  
 # round to the number of decimal places desired  
 beta <- round(c(beta), decimal\_places\_of\_precision)  
 counter <- counter+1  
 # show us the current value of beta if "verbose" is TRUE  
 if(verbose==T){  
 msg <- paste0('Iteration ', counter, ': Beta = (', beta[1], ', ', beta[2], ')')  
 cat(msg)  
 cat('\n')  
 }  
}

Iteration 1: Beta = (9.6190476, -0.1495238)  
Iteration 2: Beta = (13.6557372, -0.211247)  
Iteration 3: Beta = (14.9382896, -0.2306001)  
Iteration 4: Beta = (15.0422911, -0.2321537)  
Iteration 5: Beta = (15.0429016, -0.2321627)  
Iteration 6: Beta = (15.0429016, -0.2321627)

We arrive at the same MLEs as before, in the same number of iterations as before.

### c) We are also interested in predicting O-ring failure. Challenger was launched at . What is the predicted probability of O-ring damage at ? How many O-ring failures should be expected at ? What can you conclude?

At we predict the probability of at least one O-ring failure to be

This prediction means that we are almost certain that at least one O-ring will fail at .

## Problem 3

The elastic net (Zou and Hastie, 2006) is considered to be a compromise between the ridge and lasso penalties. The elastic net can be formulated using the Lagrangian as follows:

where and . The "credit" data set is discussed in the textbook of James et al., p83. We will fit the elastic net model to the "credit" data set using only the quantitative predictors. Our challenge is to select the appropriate and before fitting the final model.

### a) Write a function in R using the cross-validation approach to find the optimum values of and

### b) Repeat the same question as in (a) but using now the *one-standard-error* (1–SE) rule cross validation

## Appendix with R code

# Clear working environment  
rm(list=ls())  
  
# Options for document compilation  
knitr::opts\_chunk$set(warning=FALSE, message=FALSE, comment=NA, fig.width=4, fig.height=3)  
hx<-D(D(expression(4\*x\*y+(x+y^2)^2), 'x'),'x');hx ## Check H(X(n)) is a non-singular matrix  
newton <- function(f3, x0, tol = 1e-9, n.max = 100) {  
# Newton's method starting at x0  
# f3 is a function that given x returns the list  
# f(x), f'(x), Hessian f''(x)  
x <- x0 ## Set initial value  
f3.x <- f3(x) ## Set Input Function  
n <- 0 ## Set first turn n<-0  
while ((max(abs(f3.x[[2]])) > tol) & (n < n.max)) { ##Set Convergence Criteria. If f'(x) greater than tol(tolerance) go to n+1.  
x <- x - solve(f3.x[[3]], f3.x[[2]]) ##Calculate f'(x)/f''(x)  
f3.x <- f3(x)   
n <- n + 1 ##Continue to the next n  
}  
if (n == n.max) { ##If n=maximum value, output "newton failed to converge"  
cat('newton failed to converge\n')  
} else {  
return(x)  
}  
}  
  
library(Ryacas);  
k <- function(x) {  
4\*x[1]\*x[2]+(x[1]+x[2]^2)^2  
}  
##Calculate First Derivative of Function to X (f1)  
f1<-D(expression(4\*x\*y+(x+y^2)^2), 'x');f1  
##Calculate Second Derivative of Function to X (f11)  
f11<-D(D(expression(4\*x\*y+(x+y^2)^2), 'x'),'x');f11  
##Calculate First Derivative of Function to y (f2)  
f2<-D(expression(4\*x\*y+(x+y^2)^2), 'y');f2  
##Calculate Second Derivative of Function to X (f22)  
f22<-D(D(expression(4\*x\*y+(x+y^2)^2), 'y'),'y');f22  
##Calculate Second Derivative of Function to X,y (f12)  
f12<-D(D(expression(4\*x\*y+(x+y^2)^2), 'x'),'y');f12  
f3 <- function(x) {  
f <- 4\*x[1]\*x[2]+(x[1]+x[2]^2)^2  
##Calculate First Derivative of Function to x[1] (f1)  
f1<-4\*x[2]+2\*(x[1] + x[2]^2)  
##Calculate Second Derivative of Function to x[1] (f11)  
f11<-2  
##Calculate First Derivative of Function to x[2] (f2)  
f2<-4 \* x[1] + 2 \* (2 \* x[2] \* (x[1] + x[2]^2))  
##Calculate Second Derivative of Function to x[1] (f22)  
f22<-2 \* (2 \* (x[1] + x[2]^2) + 2 \* x[2] \* (2 \* x[2]))  
##Calculate Second Derivative of Function to x[1],x[2] (f12)  
f12<-4 + 2 \* (2 \* x[2])  
return(list(f, c(f1, f2), matrix(c(f11, f12, f12, f22), 2, 2))) ##Return 3 values, f, f'(x), f''(x)  
}  
for (x0 in seq(0,1, .5)) {  
for (y0 in seq(0,1, .5)) {  
cat(c(x0,y0), '--(Left=Start Point, Right=Extreme Value)--', newton(f3, c(x0,y0)), '\n')  
}}  
  
gsection = function(ftn, x.l, x.r, x.m, tol = 1e-9) {  
 # applies the golden-section algorithm to minimize ftn  
 # we assume that ftn is a function of a single variable  
 # and that x.l < x.m < x.r and ftn(x.l), ftn(x.r) >= ftn(x.m)  
   
 # the algorithm iteratively refines x.l, x.r, and x.m and terminates  
 # when x.r - x.l <= tol, then returns x.m  
   
 # golden ratio plus one  
 gr1 = 1 + (1 + sqrt(5))/2  
 # successively refine x.l, x.r, and x.m  
 f.l = ftn(x.l)  
 f.r = ftn(x.r)  
 f.m = ftn(x.m)  
 while ((x.r - x.l) > tol) {   
 if ((x.r - x.m) > (x.m - x.l)) {  
 y = x.m + (x.r - x.m)/gr1  
 f.y = ftn(y)  
 if (f.y <= f.m) {  
 x.l = x.m  
 f.l = f.m  
 x.m = y  
 f.m = f.y  
 } else {  
 x.r = y  
 f.r = f.y  
 }  
 } else {  
 y = x.m - (x.m - x.l)/gr1  
 f.y = ftn(y)  
 if (f.y <= f.m) {  
 x.r = x.m  
 f.r = f.m  
 x.m = y  
 f.m = f.y  
 } else {  
 x.l = y  
 f.l = f.y  
 }  
 }  
 }  
 return(x.m)  
}  
##Set Function F  
f <- function(x) {  
f <- 4\*x[1]\*x[2]+(x[1]+x[2]^2)^2  
 return(f)  
}  
##Set Function Gradient F f'(x)  
gradf<- function (x)  
 {##Calculate First Derivative of Function to x[1] (f1)  
 f1<-4\*x[2]+2\*(x[1] + x[2]^2)  
 ##Calculate First Derivative of Function to x[2] (f2)  
 f2<-4 \* x[1] + 2 \* (2 \* x[2] \* (x[1] + x[2]^2))  
 return(c(f1, f2))  
 }  
line.search <- function(f, x, gradf, tol = 1e-9, a.max = 100) {  
# x and gradf are vectors of length d  
# g(a) =f(x +a\*gradf) hasa local minumum at a,  
# within a tolerance  
# if no local minimum is found then we use 0 or a.max for a  
# the value returned is x + a\*y  
if (sum(abs(gradf)) == 0) return(x) # g(a) constant  
g <- function(a) return(f(x - a\*gradf))  
  
# find a.l < a.m < a.r such that  
# g(a.m) >=g(a.l) and g(a.m) >= g(a.r)  
# a.l  
a.l <- 0  
g.l <- g(a.l)  
# a.m  
a.m <- 1  
g.m <- g(a.m)  
while ((g.m > g.l) & (a.m > tol)) {  
a.m <- a.m/2  
g.m <- g(a.m)  
}  
# if a suitable a.m was not found then use 0 for a  
if ((a.m <= tol) & (g.m >= g.l)) return(x)  
# a.r  
a.r <- 2\*a.m  
g.r <- g(a.r)  
while ((g.m >= g.r) & (a.r < a.max)) {  
a.m <- a.r  
g.m <- g.r  
a.r <- 2\*a.m  
g.r <- g(a.r)  
}  
# if a suitable a.r was not found then use a.max for a  
if ((a.r >= a.max) & (g.m > g.r)) return(x - a.max\*gradf)  
# apply golden-section algorithm to g to find a  
a <- gsection(g, a.l, a.r, a.m)  
return(x - a\*gradf)  
}  
descent <- function(f,gradf, x0, tol = 1e-9, n.max = 100) {  
# steepest descent algorithm  
# find a local minimum of f starting at x0  
# function gradf is the gradient of f  
x <- x0  
x.old <- x  
x <- line.search(f, x, gradf(x))  
n <- 1  
while (f(x.old)-(f(x)> tol) & (n < n.max)) {  
x.old <- x  
x <- line.search(f, x, gradf(x))  
n <- n + 1  
}  
return(x)  
}  
  
descent(f,gradf,c(1,0) )  
  
# load challenger data  
library(mcsm)  
data(challenger)  
y <- challenger$oring  
x <- challenger$temp  
Z <- cbind(rep(1, nrow(challenger)), challenger$temp)  
find\_pis <- function(beta, x){  
 # beta a vector of parameter estimates, x a vector of independent variable observations  
 pis <- exp(beta%\*%t(Z))/(1+exp(beta%\*%t(Z)))  
 return(pis)  
}  
  
find\_W <- function(pis){  
 W <- diag(c(pis\*(1-pis)))  
 return(W)  
}  
find\_update <- function(beta, Z, W, y, pis){  
 numerator <- t(Z)%\*%(y-t(pis))  
 denominator <- t(Z)%\*%W%\*%Z  
 update <- solve(denominator)%\*%numerator  
 return(beta+c(update))  
}  
beta <- c(0, 0)  
old\_beta <- c(NA, NA)  
counter <- 0  
decimal\_places\_of\_precision <- 7  
verbose <- TRUE  
  
while(!identical(beta, old\_beta)){  
 pis <- find\_pis(beta, x)  
 W <- find\_W(pis)  
 # set old beta equal to current beta before updating  
 old\_beta <- beta  
 # replace beta with updated estimates  
 beta <- find\_update(beta, Z, W, y, pis)  
 # round to the number of decimal places desired  
 beta <- round(beta, decimal\_places\_of\_precision)  
 counter <- counter+1  
 # show us the current value of beta if "verbose" is TRUE  
 if(verbose==T){  
 msg <- paste0('Iteration ', counter, ': Beta = (', beta[1], ', ', beta[2], ')')  
 cat(msg)  
 cat('\n')  
 }  
}  
# see our estimates  
print(beta)  
# see the glm estimates  
glm(y~x, family=binomial)$coefficients  
find\_x <- function(y, beta, pis, Z, W){  
 e\_t <- y-c(pis)  
 x <- Z%\*%beta+solve(W)%\*%e\_t  
 return(x)  
}  
find\_update\_irls <- function(Z, W, working\_response){  
 numerator <- t(Z)%\*%W%\*%working\_response  
 denominator <- t(Z)%\*%W%\*%Z  
 update <- solve(denominator)%\*%numerator  
}  
beta <- c(0, 0)  
old\_beta <- c(NA, NA)  
counter <- 0  
decimal\_places\_of\_precision <- 7  
verbose <- TRUE  
  
while(!identical(beta, old\_beta)){  
 pis <- find\_pis(beta, x)  
 W <- find\_W(pis)  
 working\_response <- find\_x(y, beta, pis, Z, W)  
 # set old beta equal to current beta before updating  
 old\_beta <- beta  
 # replace beta with updated estimates  
 beta <- find\_update\_irls(Z, W, working\_response)  
 # round to the number of decimal places desired  
 beta <- round(c(beta), decimal\_places\_of\_precision)  
 counter <- counter+1  
 # show us the current value of beta if "verbose" is TRUE  
 if(verbose==T){  
 msg <- paste0('Iteration ', counter, ': Beta = (', beta[1], ', ', beta[2], ')')  
 cat(msg)  
 cat('\n')  
 }  
}