SDS 385 – Big Data
Sept 13, 2016
Peer Review 1

Reviewer: Jennifer Starling
Reviewing: David Puelz

Topic: Exercise 01 (Gradient Descent & Newton's Method)

---

1. Performance Comparison for Two Newton Functions:

I compared the *newton()* and *Newtonapprox()* functions for performance, using the microbenchmark function. While these functions are doing the same thing from a theoretical perspective, I found that the *newton* function was much, much faster than the *newtonapprox* function.

All benchmarking times are in milliseconds, with ten iterations for the benchmark. I used the data from the class data set to benchmark, with betas set to a vector of zero (mimicking initializing the β vector.)

```
> microbenchmark(
+         newtonapprox(Y,X,B0,m=1,tol,iter,alpha),
+         newton(Y,X,B0,m=1,tol,iter,alpha),
+         unit='ms',times=10
+ )
```

Expression	Min	Lq	Mean	Median	Uq	Max	Neval	cld
Newtonapprox	127597.52805	128126.15686	128691.61843	128444.25522	128682.73582	131914.5958	10	b
newton	40.03894	41.30073	59.51399	43.33653	47.39467	126.6601	10	a

There are some areas where the *Newtonapprox()* function is adding a extra work, for example, beginning in row 136 of your functions file:

```
w = as.numeric(wts(Bmat[ii-1,],X))
 Wtil = diag((w+1e-6)*(1-w+1e-6))

    # "residuals"
    S = y - m*w

    # working weights
       A = M %*% Wtil
```

Here, you have a vector of diagonals w, and then you are re-expanding it to a matrix and multiplying, which requires a lot more work. Figuring out how to use w as a vector of diagonals instead of a matrix would be good in streamlining. (You can check out my Exercise 01 code on github for an example of one way to handle this.)

It also looks like you are setting up M as a diagonal matrix also, instead of just working with M as a vector:

In Line 131 of functions file, you have:

```
M = diag(rep(m,N))
```

Then in Line 143 of the functions file, you have:

```
A = M %*% Wtil
```

But both the 'm' values and the 'w' values are just two vectors – and you do an element-wise multiplication, which is much faster than this larger matrix operation. To illustrate, you could do something like this instead:

```
#Set up vector of sample sizes.  (All 1 for wdbc data.)
```

```
m <- rep(1,nrow(X))

w <- 1 / (1 + exp(-X %*% beta))
```

(The 'w' would appear anywhere you need up update weights based on the new betas.)

Then whenever you need to do something like calculate a gradient, you can do the following, where you must multiply m*w element-wise with no matrix operation.

```
gradient <- -apply(X*as.numeric(Y-m*w),2,sum)
```

---

2. General Code Review:

I looked over both of your files, and made a few general comments as follows.

1. I have not seen using a whole separate file for the functions before. There are definitely some pros and cons to this – easy to have the two files get separated, but also makes for a very clean-looking code. The other good thing is that this would make it easy to build a separate R package with your functions, if you were so inclined. And you handle the 'getting separated' thing by calling source(…) at the beginning of your homework file, so I think this approach is good.

2. In your 'functions' file, I would suggest a few coding-related additions for your functions.

   a. Add a visual delimiter in between each function, for easy readability. (Just something like #-------------------------, nothing fancy.)

   b. Add a header block for each function, so that you don't forget what the inputs/outputs are later. Include what each of the input variables is, and any requirements around it (such as, a matrix must be certain dimensions, etc.). Include what the function outputs. Doing this helps me when I have to go back and use old code that I have not looked at in ages, and will also help other people use your code easily.

3. For your 'invmethod' function, looks like you are using R's 'solve' function instead of our solver from HW 1. This is fine, I would just add a comment that the built-in solve function is using QR decomp (99.9% sure it is), so that you remember for future reference.

4. For your iterative functions (newton, gradient descent, etc), I would have the function return some information on convergence. I would return the number of iterations the function took to converge, as well as whether or not convergence was actually reached (just an indicator variable), versus the function running out to the maximum number of iterations allowed.

   a. This will help with a couple of things:
      i. Usability for others.
      ii. Helps you compare methodology in terms of the number of iterations required for convergence. (Such as analyzing the cost of Newton vs gradient descent, where we know each iteration takes longer for Newton. Is Newton worth it then? Depends how many iterations it takes to converge compared to gradient descent.) Now, you can microbenchmark for completion time, and also analyze number of iterations to convergence more easily.

5. For calculating the convergence criteria, it looks like the dist() function is calculating the absolute value, but it looks like you are calculating the absolute difference in the β matrix. Using change in log-likelihood is more widely accepted in the optimization community (per class discussion last week, neither are invalid) – but you might want to add log-likelihood in also.

```
distance[ii] = dist(Bmat[ii,]-Bmat[ii-1,])
    if(distance[ii] <= tol){ break }
    loglik[ii] = loglike(y,w,m)
```

3. R Appendix:

For reference, here is the R code that I used to benchmark the two Newton functions.

```
#Benchmarking for two Newton flavors:

#READ IN DATA:
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))

#INITIALIZE BETA VECTOR TO 0
B0 = rep(0,11)
alpha=1
tol=1E-10
iter=1000

#MICROBENCHMARKING:
microbenchmark(
 newtonapprox(Y,X,B0,m=1,tol,iter,alpha),
 newton(Y,X,B0,m=1,tol,iter,alpha),
 unit='ms'
)

# DAVID'S FUNCTIONS
# equivalent newton's method implementation using iterative least squares re-weighting
newtonapprox = function(y,X,B0,m=1,tol,iter,alpha)
{
  p = dim(X)[2]
  N = dim(X)[1]
  Bmat = matrix(0,iter,p)
  Bmat[1,] = B0
  M = diag(rep(m,N))
  distance = rep(0,iter)

  for(ii in 2:iter)
  {
    w = as.numeric(wts(Bmat[ii-1,],X))
    Wtil = diag((w+1e-6)*(1-w+1e-6))

    # "residuals"
    S = y - m*w

    # working weights
    A = M %*% Wtil

    # working responses
    z = X %*% Bmat[ii-1,] + diag(1/diag(A)) %*% S
    Bmat[ii,] = cholmethod(X,z,A)

  }
  return(list(Bmat=Bmat))
}
```

```r
# newton's method optimization implementation
newton = function(y,X,B0,m=1,tol,iter,alpha)
{
  # defining relevant variables and Bmat
  p = dim(X)[2]
  N = dim(X)[1]
  Bmat = matrix(0,iter,p)
  Bmat[1,] = B0
  mvec = rep(m,N)
  loglik = rep(0,iter)
  distance = rep(0,iter)

  # iteration loop
  for(ii in 2:iter)
  {
    w = as.numeric(wts(Bmat[ii-1,],X))

    # calculate hessian
    Hess = hessian(X,mvec,w)

    # calculation gradient
    Grad = -grad(y,X,w,mvec)

    # solve linear system for "beta step"
    delB = cholmethodgen(Hess,Grad)
    Bmat[ii,] = Bmat[ii-1,] + delB

    # calculate the step size for convergence, etc.
    distance[ii] = dist(Bmat[ii,]-Bmat[ii-1,])
    if(distance[ii] <= tol){ break }
    loglik[ii] = loglike(y,w,m)
  }
  return(list(Bmat=Bmat,loglik=loglik,dist=distance))
}
```