UNIVERSITY OF TEXAS

Peer review: Mingzhang Yin

SDS 385

David Puelz

September 19, 2016

CONTENTS

1 Comments on code

In general, your code is very easy to read. I had a good time looking through it. I have a couple suggestions regarding individual functions – please see below.

1.1 Exercise 1 code

In the sparseX function, it may make sense to pass the truebeta vector as a parameter. Then, you can compare your solver's results with the true coefficient vector. I do like your use of crossprod() and the Matrix library!

```r
sparseX=function(N,P,prop=0.05) #generate N*P sparse matrix X return A,
    B s.t. A*beta=B
{
  X=matrix(rnorm(N*P),nrow=N)
  mask = matrix(rbinom(N*P,1,prop), nrow=N)
  X = mask*X
  W=diag(sample(c(1,2),N,replace=T))
  #generate Y
  truebeta<-rnorm(P,sd=10)
  error=rnorm(N)
  Y=X%*%truebeta+error
  #generate matrix A and B
  A=Matrix(crossprod(X,W)%*%X,sparse=T) #use crossprod to accelerate
  B=Matrix(crossprod(X,W)%*%Y,sparse=F)
  return(list(A=A,B=B))
}
```

1.2 Exercise 2 code

For the two functions below, it could make sense to have a separate function that constructs the weights. This could save potential debugging time and could make the code more modular.

```r
#Calculate gradient for the negative loglikelihood of binomial logistic
grad=function(X,y,m,beta)
{
  w=1/(1+exp(-X%*%beta))
  delta=m*w-y
  gradient<-t(X)%*%delta
  return(gradient)
}

#Calculate hessian for the negative loglikelihood of binomial logistic
hess=function(X,y,m,beta)
{
  w=1/(1+exp(-X%*%beta))
  D=diag(as.vector(m*w*(1-w)))  #Here diag should use a vector!
  return(crossprod(X,D)%*%X)
}
```

Just a general comment on the gradient descent code below... I really like your use of betarecord to dynamically construct the sequence of betas. This probably helps with speed and upfront memory allocation. I will use this technique for future code I write! I also appreciate your use of comments here for each of the individual steps. It makes the code very readable.

```r
#gradient descent to get MLE
#X is covariates
#y is observation
```

```
#m is parameter of binomial, 1 here
#beta0 is initial value
#return betarecord is beta value along steps, logrecord is -loglik along steps, step is total steps
#beta is final beta, obj_value is final objective function value
gradientdescent=function(X,y,m,beta0)
{
  maxstep=5000
  stepsize=10e-4 #some try and error. If set as 0.1, objective function value will oscillate in the end
  accu_beta=10e-3
  accu_log=10e-3

  step=0
  beta=beta0
  obj_value=negloglike(X,y,m,beta)
  diff_beta=1+accu_beta
  diff_log=1+accu_log

  betarecord=c()
  logrecord=c()
  while(step<maxstep && diff_beta>accu_beta && diff_log>accu_log)
  {
    gradient=grad(X,y,m,beta)
    #new \beta and -log(likelihood)
    betanew<-beta-stepsize*gradient
    obj_value_new=negloglike(X,y,m,betanew)

    #calculate one step change
    diff_beta=norm(betanew-beta,type="f")
    diff_log=abs(obj_value_new-obj_value)

    #record beta and -log(likelihood)
    betarecord=c(betarecord,betanew)
    logrecord=c(logrecord,obj_value_new)

    #update beta and -log(likelihood)
    beta=betanew
    obj_value=obj_value_new

    step=step+1
  }
  return(list(betarecord=betarecord,logrecord=logrecord,step=step,beta=beta,obj_value=obj_value))
}
```

2 General comments

I enjoyed looking through your code and running it. It would be interesting to see plots for the estimate coefficients and their convergence. Also, additional plots and analysis involving tweaking the step size and using different (simulated?) data sets would be cool to look at.