```
#### Thanks to Jared Fisher and Quan Zhang.
I learned a lot from their code, including how to structure R functions and how to make

gradient_descent=function(convex.function, gradient.function, stepsize.function, X,y, st
{
#Optimizes over bet vector, input to convex.function
# Dimensions
pp = length(starting.values)
nn = length(y)

# Starting value. Defaults as 0's
bet = starting.values
Bet = matrix(nrow = pp, ncol = max.iter )
Bet[,1]=bet
object_value=numeric()
object_value[1]=exp(700)
stepsize=numeric()

gradient_mat=matrix(nrow=pp,ncol=max.iter)
minibatch=1:nn
iterating=T
s=1
while(iterating)
{
s=s+1
#cat(s)
if(stochastic) minibatch = sample(1:nn,size=minibatch_size)
### # need change
gradient = gradient.function(bet = bet, X=X, y=y, minibatch=minibatch, ...)#lam=lam, alp
gradient_mat[,s]=gradient
```

```
#stepsize[s]=1/s
#if(F)
stepsize[s]=stepsize.function(iteration = s,
                                  convex.function = convex.function,
                                  direction.vector = -gradient,
                                  current.gradient.vector = gradient,
                                  current.beta.vector = bet,
                                  X=X,y=y,...
                                  )

#cat(stepsize[s],"\n")
bet = bet - stepsize[s] * gradient
# Polyak-Ruppert averaging of betas.
if(F)
{
if(missing(burnin)|| s<= burnin)
{
  Bet[,s] = bet
}else
{
# smoothing bet
   Bet[,s] <- 1/(s-burnin) * bet  +  (s-burnin-1)/(s-burnin) * Bet[,s-1]
}
}
Bet[,s]=bet
# use AR(1) to smooth the objective function value
object_value[s]=convex.function(bet=Bet[,s], X=X,y=y, minibatch=minibatch,...)*function_
object_value[s-1]*(1-function_smoothing_factor)
if( abs(object_value[s]-object_value[s-1]) < tol || (s >=max.iter))
  iterating=FALSE
```

```
}
if(s==max.iter)
print('Failed to converge')
return(list(bet=bet, Bet=Bet, total_iter=s, stepsize=stepsize, object_value=object_value
}

ilogit=function(u) return( 1/(1+exp(-u)));

Lalpha_logistic_nlg=function(X,y,bet,lam,alpha,...)
{
# X=data$X
# y=data$y
# n.list=data$n.list
n=length(y)
p=ncol(X)
minibatch=1:n
m=length(minibatch)
## create useful variables
lam.alpha.m1=lam *(alpha-1)
pi.t=ilogit(as.numeric(X[minibatch,]%*%bet))
logpi.t=ifelse(pi.t==0,-exp(700), log(pi.t))
log1mpi.t=ifelse(pi.t==1, -exp(700), log(1-pi.t))
    #W=diag(n.list[minibatch]*pi.t*(1-pi.t))
    res=-sum(n.list[minibatch]*y[minibatch]*logpi.t)+-sum(n.list[minibatch]*(1-y[minibat
return(res)
}

Grad=function(X,y,bet,lam,alpha,minibatch,...)
{
# X=data$X
```

```
# y=data$y
# n.list=data$n.list
n=length(y)
p=ncol(X)
m=length(minibatch)

X.t.n.list.y=crossprod(X[minibatch,], n.list[minibatch]*y[minibatch])
pi.t=ilogit(as.numeric(X[minibatch,]%*%bet))
return(
-c(X.t.n.list.y-crossprod(X[minibatch,],
   n.list[minibatch]*pi.t)-lam*abs(bet)^(alpha-1)*sign(bet))*m/n
)
}

backtracking.line.search = function(iteration, convex.function, direction.vector, curren
{
  # Returns step size (scalar) for optimization that fulfills sufficient decrease condit
  # Requires as input: convex.function, gradient.vector, direction.vector,
  # current.beta, gam0 > 0, rho.contraction.factor in (0,1), c.constant in (0,1)
  # input "iteration" is just there to match other step size functions ...

  # Check for correct ranges of function parameters
  if(gam0 <= 0){stop('gam0 is not positive')}
  if(rho.contraction.factor <= 0 | rho.contraction.factor >= 1){stop('rho.contraction.fa
  if(c.constant <= 0 | c.constant >= 1){stop('c.constant is not in (0,1)')}

  # Set initial gam value
  gam = gam0

  # Important values that need only be calculated once
```

```
  current.function.value = convex.function(X=X, y=y, bet=current.beta.vector,  ...)
  dot.product = c.constant*sum(current.gradient.vector*direction.vector)
  # Adjust to correct appropriate step size (gam)
  while( convex.function(bet = (current.beta.vector + gam*direction.vector), X=X, y=y,..
  {
   #cat("here", "\n")

    gam = gam * rho.contraction.factor
  }

  # Return better step size!
  return(gam)
}


############
res = gradient_descent(
  convex.function = Lalpha_logistic_nlg,
  gradient.function =Grad,
  # 1. set fixed stepsize
  #stepsize.function = function(...){10^-5},
  # 2. set stepsize\propto c/iteration, c is a constant
  #stepsize.function=function(iteration,...){2/iteration},
  # 3. find stepsize using backtracking line search, but we need further tuning of c and
  stepsize.function=backtracking.line.search,
  X=X,
  y=y,
  starting.values = rnorm(5,0,2),
  function.smoothing.factor=1,
  max.iter=10000,
```

```
  min.iter=1,
  tol = 0.00001,
  stochastic = T,
  minibatch_size=20,
  burnin=max.iter/2,
  function_smoothing_factor=1,
  lam=2,
  alpha=1.2
)
```