SDS 385: Exercise 02

September 8, 2016

Jennifer Starling

Exercises 2: Online learning

Stochastic gradient descent: background

In the previous set of exercises, we learned about gradient descent. To review: suppose we have a loss function $l(\beta)$ that we want to minimize, like a negative log likelihood or negative log posterior density. In gradient descent, we start with an initial guess $\beta^{(0)}$ and then repeatedly take steps in a downhill direction:

$$\beta^{(t+1)} = \beta^{(t)} - \gamma^{(t)} g^{(t)} ,$$

where $g^{(t+1)} = \nabla l(\beta^{(t)})$ is the gradient of the loss function evaluated at the previous iterate, which points locally uphill, and where $\gamma^{(t)}$ is a scalar step size (which might or might not actually depend on the iteration number t).

To calculate the gradient, we need to sum over the contributions from all n data points. But what if instead $g^{(t)}$ were not the actual gradient of the loss function, but merely an approximation to that gradient? In this context, by an approximation, we mean that the (negative) step direction $g^{(t)}$ is a random variable that satisfies

$$E(g^{(t)}) = \nabla l(\beta^{(t)}) .$$

Thus $g^{(t)}$ is an unbiased estimate of the gradient, but has some error. If you used such a random $g^{(t)}$ in your update instead of the actual gradient, some individual steps would lead you astray, but each step would take you in the right direction, on average. This is called *stochastic gradient descent*, or SGD.

Does SGD actually converge to the minimum of $l(x)$? It's easy to convince yourself that, if your step sizes $\gamma^{(t)}$ were constant, then you'd never get to the minimum. You might get close, but the randomness in the search directions would have you perpetually bouncing around the minimum like a moth around a flame. It follows that, if you ever hope to get to the minimum, a necessary condition is that your step sizes $\gamma^{(t)}$ get smaller, at least on average.

So assuming you handle the step sizes properly—a big if, as we'll see—here are two follow-up questions.

1. How random can these $g^{(t)}$'s be and still end up getting us to minimum of $l(\beta)$?

2. Why on earth would we want to inject randomness into the gradient-descent direction in the first place?

The answers are: 1) pretty darn random, and 2) so that we only ever have to touch one data point at a time! Let's explore.

SGD for logistic regression

(A) Let $l(\beta)$ be the negative log likelihood associated with the logistic regression model, which was a sum over n terms (one term for each data point). Earlier you derived the gradient of $l(\beta)$. If you haven't already, show that this gradient can be written in the form

$$\begin{aligned}\nabla l(\beta) &= \sum_{i=1}^{n} g_i(\beta) \\ g_i(\beta) &= (\hat{y}_i - y_i)x_i \\ \hat{y}_i &= E(y_i \mid \beta) = m_i \cdot w_i(\beta) = m_i \cdot \frac{1}{1+\exp(-x_i^T\beta)} .\end{aligned}$$

(B) Optional but interesting. Suppose that you draw a single data point at random from your sample, giving you the pair $\{y_i, x_i\}$. If you can, show that the random vector $ng_i(\beta)$ is an unbiased estimate of $\nabla l(\beta)$:

$$E\{ng_i(\beta)\} = \nabla l(\beta) ,$$

where the expectation is under random sampling from the set of all $\{y_i, x_i\}$ pairs. Alternatively, you can also show that these two vectors have the same expectation under the assumption that all the $\{y_i, x_i\}$ pairs are drawn *i.i.d.* from some wider population distribution Π.

Note: when we apply SGD using this fact, we typically drop the leading term of n in front of $g_i(\beta)$ and absorb it implicitly into the step size $\gamma^{(t)}$.

(C) For the rest of the exercises, you can take the claim in (B) as given. The idea here is that, instead of using the gradient calculated from all n data points to choose our step direction in gradient descent, we use the gradient $g_i(\beta)$ calculated from a single data point, sampled randomly from the whole data set. Because this single-data-point gradient is an unbiased estimate of the full-data gradient, we move in the right direction toward the minimum, on average.

Try this out! That is, code up stochastic gradient descent for logistic regression, in which each step takes the form

$$\beta^{(t+1)} = \beta^{(t)} - \gamma^{(t)} g_t(\beta^{(t)}) ,$$

where $g_t(\beta)$ is the gradient contribution from single randomly sampled data point, evaluated at the current guess for β.

Some notes:

1. Focus on functionality first. Forget about speed for now. It's said in CS that premature optimization is the root of all evil. Don't worry: soon enough, you'll have an SGD implementation that will blaze.
2. You'll almost certainly want to try this on simulated data first, so you can compare the answer you get both with the ground truth and with the answer you get from something like Newton's method. Simply plotting truth versus SGD estimate, or Newton estimate versus SGD estimate, will be informative.
3. For this part, keep the step size constant over time: that is, $\gamma^{(t)} \equiv \gamma$. You'll want to start your debugging process with γ pretty small. (For perspective, $\gamma = 1$ is considered a big step size). A good step size will depend very much on your data, but if things are going crazy, go an order of magnitude smaller, i.e. from 0.1 to 0.01 or even smaller. This doesn't have to be elegantly handled in your code (i.e. you can hard-code it), because you won't be using a constant step size going forward—it's just for intuition-building here.
4. Once you feel like your code is doing something vaguely reasonable, start experimenting with the step size γ. Note any general lessons you learn.
5. Sampling can be with or without replacement. However, it can easily be the case that you need more SGD steps than there are samples n in the data set, implying that if you sample without replacement, you'll have to start over (but not reset β!) after n samples. That's OK! Sometimes a batch of n SGD steps, sampled without replacement from the full data set—which is just a permutation of the original data points—is called a *training epoch*. (Prompting the question: epic training or training epoch?)

You'll also want to track the convergence of the algorithm. Some notes on this point as you think through your options:

1. The surefire way to do this is to track the value of the full objective function $l(\beta)$ at every step, so you can plot it over time and check whether it's actually going down.

2. Of course, if you do this, you'll defeat the purpose of touching only one data point at every iteration. You'll probably want to do it anyway for now, as you build intuition and debug code. But you should also consider tracking something that doesn't suffer from this problem: the running average of $l_t(\beta)$, the individual log likelihood contribution from the data point you sample at step t. Although. . . .

3. Really, if you want to be even more clever than this, you can track the exponentially weighted moving average of the $l_t(\beta)$'s, so that the influence of the early (bad) log-likelihood evaluations decays over time.

(D) Now try a decaying step size. Specifically, use the Robbins–Monro rule for step sizes:

$$\gamma^{(t)} = C(t + t_0)^{-\alpha} \, ,$$

where $C > 0$, $\alpha \in [0.5, 1]$, and t_0 (the "prior number of steps") are constants. The exponent α is usually called the learning rate. Clearly the closer α is to 1, the more rapidly the step sizes decay.

Implement the Robbins-Monro rule in your SGD code. Pick a smallish t_0 (1 or 2) and run with it. Fiddle around with C and α to see if you can get good performance.

(E) Finally, try the following averaging approach. Keep the updating scheme exactly as before. But now, instead of reporting the iterates $\beta^{(t)}$, report the time-averaged iterates

$$\bar{\beta}^{(t)} = \frac{1}{t} \sum_{k=0}^{t-1} \beta^{(t)} \, .$$

To be clear: you don't use this formula to actually compute the updates, which are exactly the same as before. You merely use it to report the final answer, and to track the log-likelihood of the intermediate answers as the algorithm proceeds. Moreover, you might want to start this averaging after an initial burn-in period.

This scheme is called *Polyak–Ruppert* averaging. Does it seem to improve matters?

Linear Regression

Part A

In Exercise 01, Generalized Linear Regression, Part A, the derivation of the gradient of the negative log-likelihood in matrix form is:

$\nabla l(\beta) = -\sum_{i=1}^{N} (y_i - m_i w_i) x_i$

In matrix form: $\nabla l(\beta) = -X'(y - mw) = X'(mw - y) = X'(\hat{y} - y)$

with $w_i = w_i(\beta) = \frac{1}{1+exp(-x_i{}'\beta)}$

Part B

Suppose you draw a single point at random from the sample of data, giving the pair $\{y_i, x_i\}$, where y_i is a single response from row i and x_i is the i^{th} row of design matrix X.

Then $E(ng_i(\beta)) = nE(g_i(\beta))$.

i is the only random value here, with $P(i = j) = 1/n$ for $j \in \{i = 1, ..., n\}; 0$ otherwise.

So $nE(g_i(\beta)) = n(\frac{1}{n}) \sum_{i=1}^{n} g_i(\beta) = \sum_{i=1}^{n} g_i(\beta) = \nabla(l(\beta))$.

Therefore $E(ng_i(\beta)) = \nabla(l(\beta))$.

Part C

I implemented the running average of $l(\beta)$ to track convergence.

Part D

For a decaying step size, $C = 40, \alpha = 0.5, t_0 = 2$ gave good approximations of the betas within a million iterations.

Part E

The Polyak-Ruppert Averaging did not improve my results. My results without the averaging were actually closer to the GLM-generated βs.

Results

My results for the stochastic gradient descent were as follows, using the following:

- $C = 40, \alpha = 0.5, t_0 = 2$
- One million iterations
- No specific convergence criteria; just ran for full iterations.

```
> beta #GLM estimates
          X         XV3         XV4         XV5         XV6         XV7
 0.48701675 -7.22185053  1.65475615 -1.73763027 14.00484560  1.07495329
        XV8         XV9        XV10        XV11        XV12
-0.07723455  0.67512313  2.59287426  0.44625631 -0.48248420
> betas[i,] #Stochastic estimates.
 [1]  0.4931931 -3.9841799  1.6160841 -4.0661091 12.7862893  1.0945374
 [7]  0.1505122  0.8018911  2.6743230  0.5257934 -0.4182426
> beta_pr  #Estimates with Polyak—Ruppert Averaging
 [1]  0.30690366 -3.63914437  1.66252165 -4.01274222 12.33884238  1.05591877
 [7]  0.05485383  0.70729371  2.61896729  0.45418036 -0.50787328
```

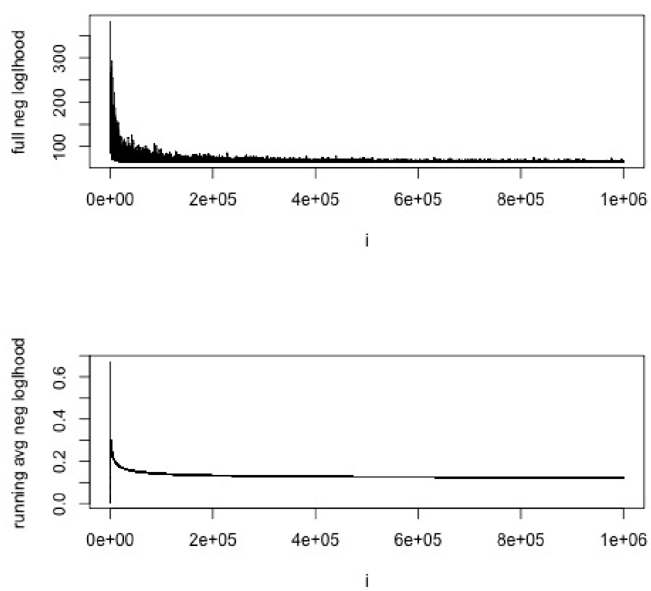

Figure 1: Negative Log-Likelihood Plots

Appendix: R Code

```
### SDS 385 - Exercises 02 - Part C
#This code implements stochastic gradient descent to estimate the
#beta coefficients for binomial logistic regression.

#Jennifer Starling
#30 August 2016

rm(list=ls())   #Cleans workspace.
library(microbenchmark)
library(permute)
library(zoo)    #For rolla pply

#PART C:

#Read in code.
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))

#------------------------------------------------------------------
#Binomial Negative Loglikelihood function.
    #Inputs: Design matrix X, vector of 1/0 vals Y,
    #   coefficient matrix beta, sample size vector m.
    #Output: Returns value of negative log-likelihood
    #   function for binomial logistic regression.
logl <- function(X,Y,beta,m){
    w <- 1 / (1 + exp(-X %*% beta)) #Calculate probabilities vector w_i.
    logl <- - sum(Y*log(w+1E-4) + (m-Y)*log(1-w+1E-4)) #Calculate log-likelihood.
        #Adding constant to resolve issues with probabilities near 0 or 1.
    return(logl)
}

#------------------------------------------------------------------
#Stochastic Gradient Function:
    #Inputs: Vector X (One row of design matrix), vector of 1/0 vals Y,
    #   coefficient matrix beta, sample size vector m.
    #Output: Returns value of gradient function for binomial
    #   logistic regression.

gradient <- function(X,Y,beta,m){
    w <- 1 / (1 + exp(-X %*% beta)) #Calculate probabilities vector w_i.

```

```
    gradient <- array(NA,dim=length(beta))  #Initialize the gradient.
    gradient <- apply(X*as.numeric(m*w-Y),2,sum) #Calculate the gradient.

    return(gradient)
}

#------------------------------------------------------------------------
#Robbins-Monro Step Size Function:
#   Inputs: C>0, a constant.  a in [.5,1], a constant.
#       t, the current iteration number.  t0, the prior number of steps.
#       (Try smallish t0, 1 to 2.)
#   Outputs: step, the step size.

rm_step <- function(C,a,t,t0){
    step <- C*(t+t0)^(-a)
    return(step)
}

#Playing with step sizes:
t <- 1:50
#sp <- rm_step(C=5,a=.75,t=t,t0=2)
p#lot(t,sp)

#Varying C:
cl <- rainbow(5)
#plot(t,rm_step(C,a[1],t,t0[2]),col=cl,lwd=1,pch=20,cex=.5)

#Varying a:
#plot(t,rm_step(C[1],a,t,t0[2]),col=cl,lwd=1,pch=20,cex=.5)

#Varying t:
cl2 <- rainbow(2)
#plot(t,rm_step(C[2],a[5],t,t0),col=cl2,lwd=1,pch=20,cex=.5)

#Play with ideal step size curve shape:
C=10;  t0=1;  a=.75;
plot(t,rm_step(C,a,t,t0),type='l',col='blue')

#------------------------------------------------------------------------
#Stochastic Gradient Descent Algorithm:

#1. Fit glm model for comparison. (No intercept: already added to X.)
glm1 = glm(y~X-1, family='binomial') #Fits model, obtains beta values.
beta <- glm1$coefficients

maxiter <- 1000000  #Specify max iterations allowed.

#Initialize matrix to hold gradients for each iteration.
grad <- matrix(0,nrow=maxiter,ncol=ncol(X))

#Initialize matrix to hold beta vector for each iteration.
betas <- matrix(0,nrow=maxiter+1,ncol=ncol(X))

#Initialize vector to hold full loglikelihood fctn for each iter.
loglik <- rep(0,maxiter)
#Initialize vector to hold loglikelihood for each indiv t obs.
loglik_t <- rep(0,maxiter)
#Initialize vector to hold running avg for logl for t's.
loglik_ra <- rep(0,maxiter)
```

```

conv <- 1E-14   #Set convergence level.

#Set up random iterations through data, up to maxiter.
npermutes <- ceiling(maxiter/nrow(X))
obs_order <- as.vector(t(shuffleSet(1:nrow(X),nset=npermutes)))

#Initialize values:
i=1
t  <- obs_order[i]
Xnew <- matrix(X[t,,drop=F],nrow=1,byrow=T)
loglik_t[i] <- logl(Xnew,Y[t],betas[i,],m[t])
loglik_ra[i] <- loglik_t[i]
grad[1,] <- gradient(Xnew,Y[t],betas[i,],m[t])
betas[1,] <- 0

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

        #Select one random obs per iter.
        t  <- obs_order[i]
        Xnew <- matrix(X[t,,drop=F],nrow=1,byrow=T)

        #Calculate Robbins-Monro step size.
        step <- rm_step(C=40,a=.5,t=i,t0=2)

        #Set new beta equal to beta - a*gradient(beta).
        betas[i,] <- betas[i-1,] - step * grad[i-1,]

        #Calculate fullloglikelihood for each iteration.
        loglik[i] <- logl(X,Y,betas[i,],m)

        #Calculate loglikelihood of individual observation t.
        loglik_t[i] <- logl(Xnew,Y[t],betas[i,],m[t])

        #Calculate running average of loglikelihood for individual t's.
        loglik_ra[i] <- (loglik_ra[i-1]*(i-1) + loglik_t[i])/i

        #Calculate stochastic gradient for beta, using only obs t.
        grad[i,] <- gradient(Xnew,Y[t],betas[i,],m[t])

        print(i)

        #Check if convergence met:  If yes, exit loop.
        #Note: Not using norm(gradient) like with regular gradient descent.
        #Gradient is too variable in stochastic case.
        #Can run for set iterations, but here, checking for convergence based
        #on iter over iter change in running avg of log-likelihoods.

        #Check if convergence met: If yes, exit loop.
        if (abs(loglik_ra[i]-loglik_ra[i-1])/abs(loglik_ra[i-1]+1E-3) < conv ){
            converged=1;
            break;
        }

} #End gradient descent iterations.

#------------------------------------------------------------------
#Perform Polyak-Ruppert averaging to obtain final beta result:
```

```r

#Calculate burn-in period to discard: 1/2 of the total iterations.
t <- floor(i*.5):i
beta_pr <- colMeans(betas[t,])

#-------------------------------------------------------------------
#OUTPUT DATA RESULTS:

beta #GLM estimates
betas[i,] #Stochastic estimates.
beta_pr #Estimates with Polyak-Ruppert Averaging

#Plot full log-likelihood function for convergence, and running average for log-
    likelihoods.
par(mfrow=c(2,1))
plot(2:i,loglik[2:i],type='l',xlab='i',ylab='full neg loglhood')
plot(2:i,loglik_ra[2:i],type='l',xlab='i',ylab='running avg neg loglhood')

#Save plots:
jpeg(file='/Users/jennstarling/UTAustin/2016_Fall_SDS 385_Stats Models for Big
    Data/Exercise 02 LaTeX Files/Ex02_loglik_and_ravg.jpeg')
par(mfrow=c(2,1))
plot(2:i,loglik[2:i],type='l',xlab='i',ylab='full neg loglhood')
plot(2:i,loglik_ra[2:i],type='l',xlab='i',ylab='running avg neg loglhood')
dev.off()
```