Exercises 4: Better online learning Jared Fisher

Last Updated October 8, 2016

1 Improving SGD for logistic regression

The two techniques we learned on the last set of exercises—line search and Quasi-Newton methods—give us inspiration for two different ways of picking the step size in stochastic gradient descent.

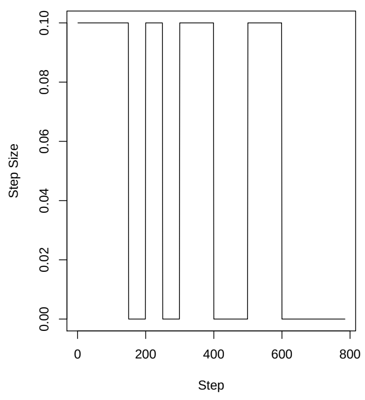
(A) First, return to your SGD code for logistic regression. Implement the following variant on line search to pick the step size.

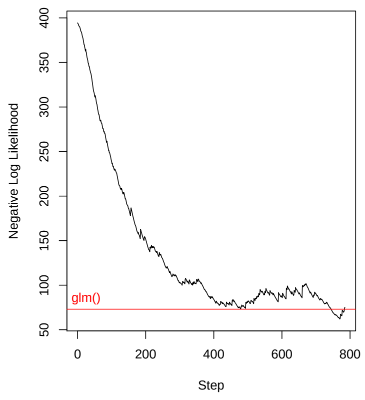
1. At some point during the learning, select a small subsample (a "minibatch") of the observations as a calibration set.
2. Try out various step sizes γ on this subsample (using the averaged gradient from this subsample as the search direction), and pick the γ that leads to the best fit on the calibration set.
3. Use this step size for awhile.

You then repeat this every so often to refresh the step size, which will end up decaying naturally over time.

Note: this way of choosing the step size based on a minibatch is *almost* exactly the same as line search for regular gradient descent, treating that minibatch as the full data set. The only difference is that the search direction vector is the *average* gradient contribution for all the observations in the minibatch, rather than the sum of all gradient contributions. These two vectors have the same direction but different length. You need the one whose length scales like that of a single g_t vector—and thus the average—to get the right step size for gradient-descent steps based on a single g_t.

Here, I run backtracking line search every 50 iterations, using 50 data points. There are a lot of "parameters" to calibrate. α_0, c, ρ for the line search, the likelihood smoothing factor α, the Polyak-Ruppert burn in size, the tolerance level, and of course the size and frequency of minibatch step sizes. With some tuning work though, the minimization converges fairly quickly.





(B) Now try the following approach, called AdaGrad (adaptive gradient descent). You'll probably need an entirely different function for this, so you can benchmark this method against the line-search-based method.

You'll remember that in the quasi-Newton method you implemented on the last exercises, your steps looked like this:

$$\beta^{(t+1)} = \beta^{(t)} - \gamma^{(t)} \Omega^{(t)} \nabla l(\beta^{(t)}) \, ,$$

where:

- $\nabla l(\beta^{(t)})$ was the gradient vector, evaluated at the curren estimate.
- $\Omega^{(t)}$ was an approximation to the inverse Hessian matrix, updated at each step according to the BFGS recursion.
- $\gamma^{(t)}$ was a scalar step-size parameter, chosen by line search.

This looks just like Newton's method, except that $\Omega^{(t)}$ is an approximate inverse Hessian rather than the true inverse Hessian (hence quasi-Newton). You will have noticed that, compared with gradient-descent, the quasi-Newton method converged in many, many fewer iterations. That's because the matrix $\Omega^{(t)}$ has the effect of re-scaling each component of the gradient vector by some amount that depends on the curvature of the objective at that point. Low curvature means a large step size is safer along that direction; high curvature means you need to be more conservative along that direction, because you might overshoot the minimum. Scaling by the *inverse* Hessian—or in the case of quasi-Newton, an estimate of this matrix—accomplishes this rescaling.

You could certainly imagine applying a similar idea to stochastic gradient descent. That is, you could make your updates look like this:

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

where just as before, g_t is the contribution to the gradient from the single data point processed at step t. Here we've absorbed the step size into the matrix $\Omega^{(t)}$, which you should think of as exactly like the approximate inverse Hessian $\Omega^{(t)}$ that arises in quasi-Newton.

However, this scheme suffers from two drawbacks. First, if we use something like the BFGS update rule to update Ω^t using the g_t's at each stage, we'll end up with a super noisy Hessian. That's because our single-data-point observations are noisy, their gradient contributions are even noisier, and their Hessian contributions are noisier still. The following R snippet will convince you of the general principle that noise accumulates very fast as you take higher-order finite differences of something noisy.

```
x = seq(0,4*pi, length=1000)

# Slightly noisy cosine curve
sigma = 0.025
y = cos(x) + rnorm(1000, 0, sigma)

# Function is easy to see despite the noise
plot(x, y, pch=19, col='grey')
curve(cos(x), add=TRUE, col='red', lwd=2)

# First differences are super noisy despite small sigma
plot(tail(x,-1), diff(y)/diff(x), pch=19, col='grey')
curve(-sin(x), add=TRUE, col='red', lwd=2)  # true first derivative

# Second differences? Yikes!
plot(tail(x,-2), diff(diff(y))/diff(diff(x)))
```

In Equation 1 above, naïvely applying the BFGS update rule to calculate Ω using the noisy gradients is like trusting the second and third pictures generated by this code snippet as good approximations to the first and second derivatives. We're effectively asking to get a reasonable approximation to the Hessian using finite differences of noisy observations. You should be wary of this.

Equally worrying, however, is the issue of scalability. Calculating the matrix-vector product $\Omega^{(t)} g_t(\beta^{(t)})$ requires $O(p^2)$ multiplications, where p is the number of regression coefficients.[1] This isn't a big deal if p is small. But if p is large, then we've somewhat defeated the main advantage of stochastic gradient

[1]This ignores the cost of updating Ω, which adds even more flops.

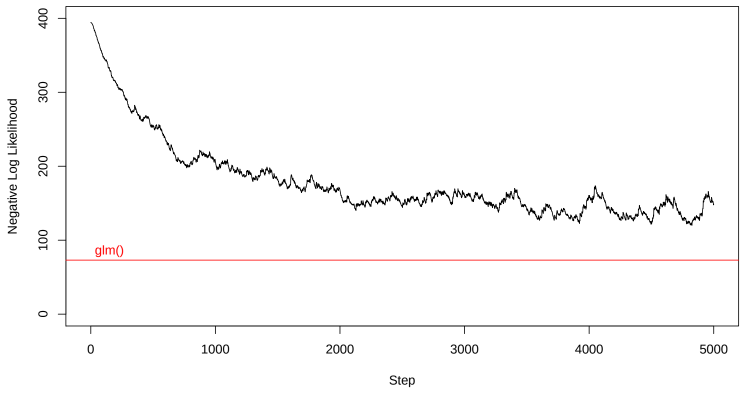
descent, which is that processing a single data is super fast: only a dot product of two length-p vectors, which is linear in p rather than quadratic.

The AdaGrad update rule addresses both problems. It says: let's use a *diagonal* approximation to the inverse Hessian matrix, updating the matrix at each step using g_t. Because we're only estimating p numbers on that diagonal (rather than $p(p+1)/2$ entries in the full inverse Hessian), there are fewer quantities to estimate, and the noise in g_t hurts us less. And if Ω is diagonal, then the cost of calculating $\Omega^{(t)} g_t(\beta^{(t)})$ scales linearly, not quadratically, in p.

The AdaGrad rule for tuning the SGD step size was originally proposed in a paper from 2011 by Duchi et. al.[2] You are obviously welcome to read it, but unless you are already an expert at convex optimization and online learning, you will almost surely find it incomprehensible. (I certainly did the first time I read it—although try revisiting it after the semester is over!) Luckily there are many sources out there that explain the basic AdaGrad rule in plainer language. The `https://en.wikipedia.org/wiki/Stochastic_gradient_descent#AdaGrad`Wikipedia page is just fine here. Here are some other references:

- `http://sebastianruder.com/optimizing-gradient-descent/index.html#adagrad`A nice blog post with a lot of other stuff about stochastic gradient descent.
- `https://xcorr.net/2014/01/23/adagrad-eliminating-learning-rates-in-stochastic-gradient-descent/`
  blog post with some nice pseudo-code.
- `https://courses.cs.washington.edu/courses/cse547/15sp/slides/adagrad.pdf`Some slides from a friend of mine, Emily Fox, at U. Washington. (These are a bit more technical.)

And of course, Google is your friend. Read up on AdaGrad and implement it. How does it work for when used on one of your earlier, familiar data sets?
Below is the convergence using the Adagrad update. It seems reasonable, but for this WDBC dataset, is not exceptional.



2 Putting it all together on some biggish data

By now you've got the ice cream for your SGD sundae. Here are some sprinkles you might consider adding:

[2]Duchi, John; Hazan, Elad; Singer, Yoram (2011). "Adaptive subgradient methods for online learning and stochastic optimization." Journal of Machine Learning Research 12: 2121–59. Duchi was a Ph.D student at the time, and the technique described in this paper more or less went straight into deployment at every major tech firm you've heard of. They all use SGD, and the AdaGrad rule is darn useful.

- Support for sparse matrices (i.e. situations where your X matrix has a lot of zeros in it). Remember the speed-ups and reduced memory requirements that are possible by exploiting the sparsity of the X matrix.[3]

- Adding an ℓ^2 regularizer, i.e. the addition of a penalty term to the original objective that stabilizes your estimates (which will modify the gradients in a straightforward way).[4] This corresponds to optimizing the objective
$$\tilde{l}(\beta) = l(\beta) + \lambda \|\beta\|_2^2 \, ,$$
where λ is a scalar penalty and $l(\beta)$ is the negative log likelihood.

- Implementation in a compiled language (e.g. using Rcpp and either RcppArmadillo or RcppEigen if you're working in R).

- Streaming data off disk in chunks, so that you don't have to load everything into memory at once. There are several R packages appropriate for this, e.g. `LaF` or `stream` (there are probably others, maybe even better ones).

Aim for speed. SGD is supposed to be fast :-)

Once you've got your code up and running (using either AdaGrad or minibatch line search), apply SGD to fit a logistic regression model to the `http://archive.ics.uci.edu/ml/datasets/URL+Reputation`data here, on detecting malicious URLs in web traffic. It has about 2.4 million observations and 3.2 million features (so $p > n$). Forget about random sampling here—just stream through the data one observation at a time, in the order they sit in the file.[5] As a final step, try splitting the data into training and test sets. Fit the model on the training set, and then make predictions on the testing set. How well does your model perform?

The course web page gives you some pointers and code for pre-processing this data. And remember, you can probably learn a lot from your classmates on this one.

First I perform a comparison using my dense sgd function in Rcpp. The Rcpp function performs 20,000 SGD iterations 2600 times faster than standard R code on the dense wdbc cancer dataset. I continue with some modifications to previous SGD implementations. First, the likelihood can be simplified to reduce computations, as shown below.

$$\begin{aligned}
\ell(\beta) &= ylog(w) + (m-y)log(1-w) \\
&= ylog\left(\frac{exp\{X\beta\}}{1+exp\{X\beta\}}\right) + (m-y)log\left(\frac{1}{1+exp\{X\beta\}}\right) \\
&= ylog\left(exp\{X\beta\}\right) - ylog\left(1+exp\{X\beta\}\right) + (y-m)log\left(1+exp\{X\beta\}\right) \\
&= yX\beta - ylog\left(1+exp\{X\beta\}\right) + ylog\left(1+exp\{X\beta\}\right) - mlog\left(1+exp\{X\beta\}\right) \\
&= yX\beta - mlog\left(1+exp\{X\beta\}\right)
\end{aligned}$$

Next, I'm not yet comfortable with the notions of lazy updating put forward in class. So here's my take, for step size γ and likelihood penalty λ, with t indicating the global iteration, where β_j was updated at t. For simplification, let this be the first pass through the data, so that $t = i$.

[3]The example below probably won't run if you don't represent the data as a sparse matrix.

[4]Consult the course page for some reading about this idea, if it is unfamiliar.

[5]As a benchmark, my code (which has all the "sprinkles" mentioned above except for the last one) takes about 13 seconds on a MacBook Pro to pass through this data set once, once it's been loaded into memory. It is reasonably efficient, but not super optimized or anything like that. I'll be very pleased if you can beat this! I'll have to think of a prize here.

$$\begin{aligned}
\beta_j^{t+1} &= \beta_j^t - \gamma[\nabla \ell(\beta)]_j \\
&= \beta_j^t - \gamma \frac{(\hat{y}_{i+1}^t - y_{i+1})x_{i+1,j} + 2\lambda\beta_j^t}{\sqrt{G_{jj}^t + \epsilon}} \\
&= \beta_j^t - \gamma \frac{2\lambda\beta_j^t}{\sqrt{G_{jj}^t + \epsilon}} && \text{let } x_{i+1,j} = 0 \\
&= \left(1 - \gamma \frac{2\lambda}{\sqrt{G_{jj}^t + \epsilon}}\right)\beta_j^t \\
\Rightarrow \beta_j^{t+2} &= \left(1 - \gamma \frac{2\lambda}{\sqrt{G_{jj}^{t+1} + \epsilon}}\right)\beta_j^{t+1} && \text{similarly, let } x_{i+2,j} = 0 \\
&= \left(1 - \gamma \frac{2\lambda}{\sqrt{G_{jj}^{t+1} + \epsilon}}\right)\left(1 - \gamma \frac{2\lambda}{\sqrt{G_{jj}^t + \epsilon}}\right)\beta_j^t \\
\Rightarrow \beta_j^{t+3} &= \beta_j^{t+2} - \gamma \frac{(\hat{y}_{i+3}^{t+2} - y_{i+3})x_{i+3,j} + 2\lambda\beta_j^{t+2}}{\sqrt{G_{jj}^{t+2} + \epsilon}} && \text{for } x_{i+3,j} \neq 0
\end{aligned}$$

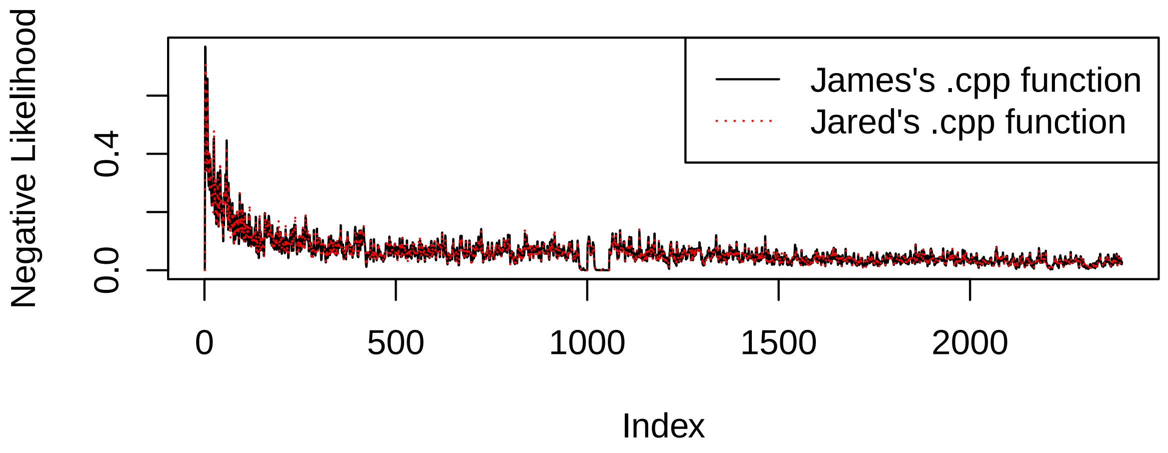
Should the diagonal of the approximated Hessian G for the Adagrad update be the sum of the likelihood evaluations, or that of the penalized likelihood? If not including the penalty, then G_{jj} doesn't update when $x_{ij} = 0$, so the lazy update here clearly would be

$$\beta_j^{t+2} = \left(1 - \gamma \frac{2\lambda}{\sqrt{G_{jj}^t + \epsilon}}\right)^2 \beta_j^t$$

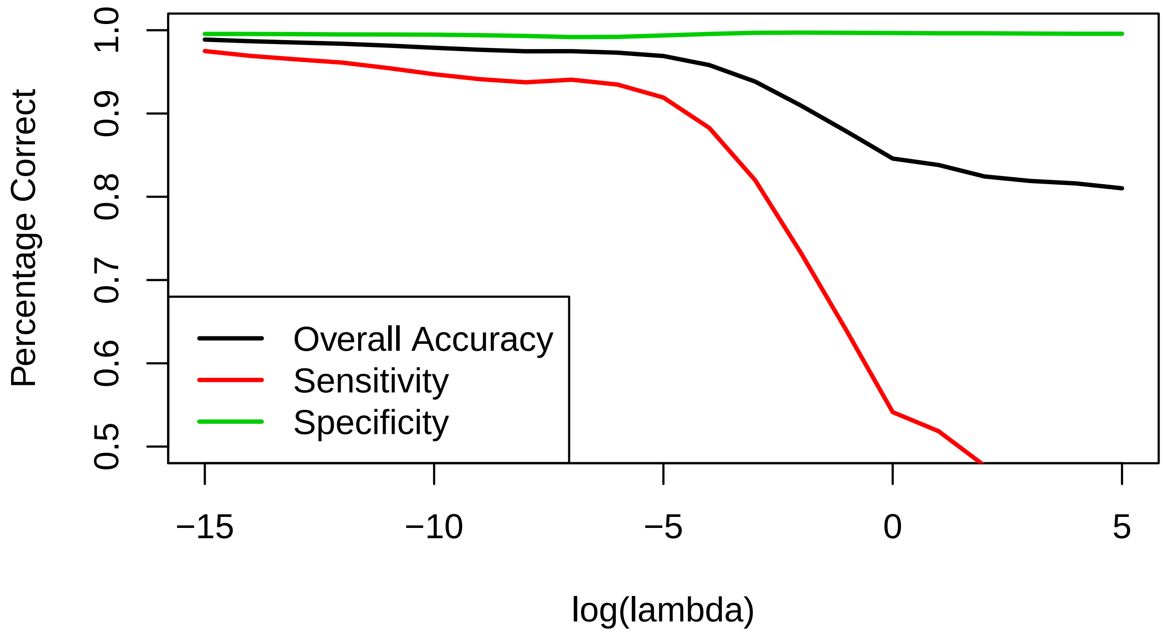
and this holds if we let G be a "stale" approximation even if we include the penalty in the updates to G. So, when updating β_j^{t+k} after last updating at t, there were $k-1$ iterations where $x_{ij} = 0$, and the lazy update should be to the power of $k-1$.

HOWEVER, this update struggles with numerical instability. So, for the sake of time, I use the "hack" variation of my derivation above.

$$\beta_j^{t+k} = \left(1 - k\gamma \frac{2\lambda}{\sqrt{G_{jj}^t + \epsilon}}\right)\beta_j^t$$



My SGD runs in 11.393 seconds, which is comparable to the 12.743 seconds required for the benchmark (James' SGD). Including invSqrt() function does not increase speed. This seems odd to me. When it comes to choosing the appropriate constant λ for the $l2$ likelihood penalty, we perform cross validation. 70% of the observations are randomly chosen to be in the training set, and the remainder are in the testing dataset. At each different λ, I take 2 passes through the data to improve the parameter estimates. There's a jump in the negative log likelihood around the break between datasets, but it happens over thousands of iterations, not the single transition iteration between passes.


Percentage Correct
1.0
0.9
0.8
0.7
0.6
0.5
−15
−10
−5
0
5
log(lambda)
Overall Accuracy
Sensitivity
Specificity