Exercises 3: Better online learning (preliminaries)

The goal of the next two sets of exercises is to make your SGD implementation better, faster, and able to exploit sparsity in the features. These exercises set the stage. On the next set, you'll then put everything together.

Once again, we'll return to the logistic-regression model, by now an old friend: $y_i \sim \text{Binomial}(m_i, w_i)$, where y_i in an integer number of "successes," m_i is the number of trials for the *i*th case, and the success probability w_i is a regression on a feature vector x_i given by the inverse logit transform:

$$w_i = \frac{1}{1 + \exp\{-x_i^T \beta\}} \, .$$

As before, we'll use $l(\beta)$ to denote the loss function to be minimized: that is, the negative log likelihood for this model.

Before we get to work on improving stochastic gradient descent itself, we need to revisit our batch[1] optimizers from the first set exercises: ordinary gradient descent and Newton's method.

Line search

Line search is a technique for getting a good step size in optimization. You have may already implemented line search on the first set of exercises, but if you haven't, now's the time.

Our iterative (batch) algorithms from the previous exercises involved updates that looked like this:

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

where $s^{(t)}$ is called the search direction. We tried two different search directions: the gradient-descent direction (i.e. in the opposite direction from the gradient at the current iterate), and the Newton direction.

In either case, we have the same question: how should we choose γ? It's clear that the best we can do, in a local sense, is to choose γ to minimize the one-dimensional function

$$\phi(\gamma) = l(\beta^{(t)} + \gamma s^{(t)}) \, ,$$

There's no other choice of γ that will lead to a bigger decrease in the loss function along the fixed search direction $s^{(t)}$. While in general it might be expensive to find the exact minimizing γ, we can do better than just guessing. Line search entails using some reasonably fast heuristic for getting a decent γ.

[1] Batch, in the sense that they work with the entire batch of data at once. The key distinction here is between batch learning and online learning, which processes the data points either one at a time, or in mini-batches.

Here we'll focus on the gradient-descent direction. Read Nocedal and Wright, Section 3.1. Then:

(A) Summarize your understanding of the *backtracking line search* algorithm based on the Wolfe conditions (i.e. provide pseudo-code and any necessary explanations) for choosing the step size.

(B) Now implement backtracking line search as part of your batch gradient-descent code, and apply it to fit the logit model to one of your data sets (simulated or real). Compare its performance with some of your earlier fixed choices of step size. Does it converge more quickly?

Remember that modular code is reusable code. For example, one way of enhancing modularity in this context is to write a generic line-search function that accepts a search direction and a callable loss function as arguments, and returns the appropriate multiple of the search direction. If you do this, you'll have a line-search module that can be re-used for any method of getting the search direction (like the quasi-Newton method in the next section), and for any loss function.[2]

Quasi-Newton

On a previous exercise, you implemented Newton's method for logistic regression. In this case, you (probably) used a step size of $\gamma = 1$, and your search direction solved the linear system

$$H^{(t)} s^{(t)} = -\nabla l(\beta^{(t)}) \, , \quad \text{or equivalently} \quad s = -H^{-1} \nabla l(\beta) \, ,$$

where $H^{(t)} = \nabla^2 l(\beta^{(t)})$ is the Hessian matrix evaluated at the current iterate. (The second version above just drops the t superscript to lighten the notation.) Newton's method converges very fast, because it uses the second-order (curvature) information from the Hessian. The problem, however, is that you have to form the Hessian matrix at every step, and solve a p-dimensional linear system involving that Hessian, whose computational complexity scales like $O(p^3)$.

Read about quasi-Newton methods in Nocedal and Wright, Chapter 2, starting on page 24. A quasi-Newton method uses an approximate Hessian, rather than the full Hessian, to compute the search direction. This is a very general idea, but in the most common versions, the approximate Hessian is updated at every step using that step's gradient vector. The intuition here is that, because the second derivative is the

[2] This is not the only way to go, by any stretch. In particular, my comment assumes that you're adopting a more functional programming style, rather than a primarily object-oriented style with user-defined classes and methods.

rate of change of the first derivative, the successive changes in the gradient should provide us with information about the curvature of the loss function. Then:

(A) Briefly summarize your understanding of the *secant condition* and the *BFGS* (Broyden-Fletcher-Goldfarb-Shanno) quasi-Newton method. Provide pseudo-code showing how BFGS can be used, in conjunction with backtracking line search, to fit the logistic regression model.

Note: as discussed in Nocedal and Wright, your pseudo-code should use the BFGS formula to update the *inverse* of the approximate Hessian, rather than the approximate Hessian itself. An important question for you to answer here is: why?

(B) Now implement BFGS coupled with backtracking line search to fit the logit model to one of your data sets (simulated or real). Compare its performance both with Newton's method and with batch gradient descent, in terms of the number of overall steps required.