Exercises 3: Better online learning (preliminaries)

Jared Fisher
Last Updated September 26, 2016

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 ith 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.

1 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.

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.

We want to choose a step size to minimize the function

$$\phi(\alpha) = \ell(\beta_k + \alpha p_k),\ \alpha > 0$$

where p_k is the step direction and $\ell()$ is the function we're minimizing (e.g. the negative log likelihood). Note on the intuition: this function $\phi(\alpha) = \ell(\beta_k + \alpha p_k) = \ell(\beta_{k+1})$ is the function of interest's value at the next step. The best α value is hard to find, so instead we'll find good ranges for α. Specifically, we'll establish rules for the boundaries of these ranges, then find an α that works in the range!

The first of the Wolfe conditions is called the Armijo condition:

$$\ell(\beta_k + \alpha_k p_k) \leq \ell(\beta_k) + c_1 \alpha \nabla \ell_k^T p_k$$

[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.

which ensures that the step yields a sufficient decrease in the function of interest. However, this allows for really small, unhelpful steps. To ensure the steps are of reasonable size, we impose the second of the Wolfe conditions called the curvature condition:

$$\nabla \ell(\beta_k + \alpha_k p_k)^T p_k \geq c_2 \nabla \ell_k^T p_k.$$

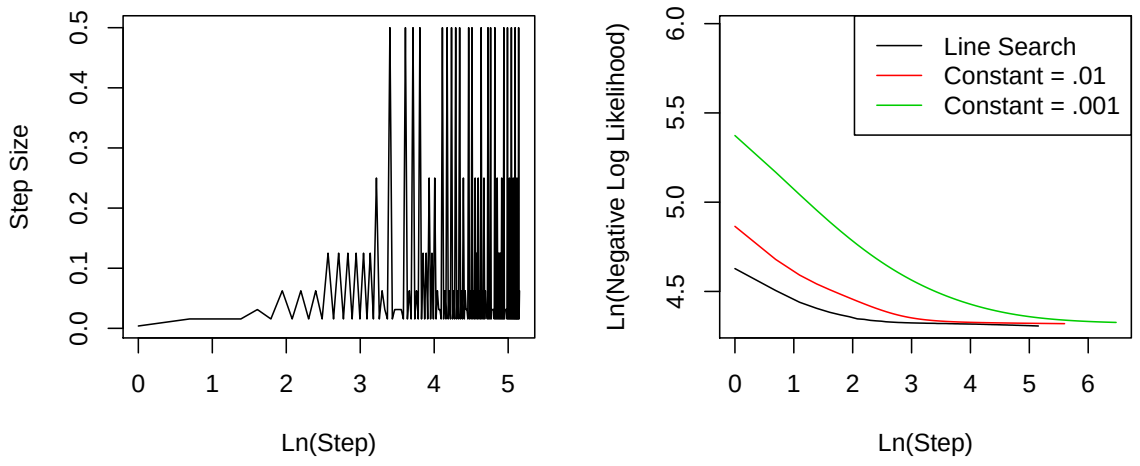
This uses the current slope to inform the size of the step, namely if the slope is largely negative, we still have more hill to descend as opposed to when the slope is almost flat and we've almost hit the minimum.

However, if we use a backtracking approach, we can forget about the curvature condition (WHY??? the book doesn't elaborate). So, just using the Armijo or "sufficient decrease condition" we get the backtracking line search:

- Choose $\alpha_0 > 0$; $\rho, c \in (0, 1)$
- Set $\alpha = \alpha_0$
- while $\ell(\beta_k + \alpha p_k) > \ell(\beta_k) + c\alpha \nabla \ell_k^T p_k$,
  - set $\alpha = \rho\alpha$
- Choose $\alpha_k = \alpha$

(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?

The parameters α_0, ρ and c require some tuning to enhance performance. Here, $\alpha_0 = 1$, $\rho = 0.5$, and $c = 0.2$. I think the book's suggestion to allow ρ to vary on different iterations will help performance. (Instead of taking time to implement non-constant ρ, I was debugging my line search. But after looking at David and Jennifer's scripts, I realized the direction vector is not just the gradient, but is negative. Thanks y'all!)



Remember that modular code is reusable code. Yep! My code is modular, thanks to practice in James' classes:) 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]

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 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.

The *secant condition* is

$$B_{k+1} s_k = d_k$$

where $s_k = \beta_{k+1} - \beta_k$ is the step, $d_k = \nabla \ell_{k+1} - \nabla \ell_k$, and B is an approximated Hessian. This is obtained by approximating results from Taylor's theorem. The Broyden, Fletcher, Goldfarb, Shanno (BFGS) formula yields a symmetric update to the Hessian estimate:

$$B_{k+1} = B_k - \frac{B_k s_k s_k^T B_k}{s_k^T B_k s_k} + \frac{d_k d_k^T}{d_k^T s_k}.$$

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?

Let the approximate inverse Hessian be $H_k = B_k^{-1}$. The equivalent update of this is

$$H_{k+1} = \left(I - \frac{d_k s_k^T}{d_k^T s_k}\right) H_k \left(I - \frac{d_k s_k^T}{d_k^T s_k}\right)^T + \frac{s_k s_k^T}{d_k^T s_k}$$

which can be used advantageously along side line search. This is demonstrated in the following psuedocode, where it's faster to update and use the inverse of the approximate Hessian to have a single matrix multiplication instead of solving a linear system. (right?) For step $k+1$,

- Use inverse Hessian approximation H_k to find new search direction $p_k = -H_k \nabla \ell_k$
- Line search for step size α_k
- Update $\beta_{k+1} = \beta_k + \alpha_k p_k$

[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.

- Check if tolerance condition is met. If so, terminate loop.
- BFGS update to of H_k to H_{k+1}.

(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.

Here I compare this new quasi Newton's method with the other methods we look at. It converges at about the same speed as Newton's method and gradient descent with line search, but each terminates at different points. Newton's Method is the fastest and most precise.