Exact linesearch for LASSO

Stephen Becker, University of Colorado Boulder*

Original version 2011, last updated May 19, 2016

Abstract

A method for computing the exact linesearch for LASSO problems is described. For vectors of size n, the method requires sorting n numbers and a few $\mathcal{O}(n)$ operations. The algorithm is similar in spirit to fast projections onto the ℓ_1 ball, and falls into a broader class of algorithms which have efficient solutions (cf. P. Brucker (1984)[1]). As such, the algorithm is not novel and variants have likely been derived, but it is not easy for a non-specialist to find a description or code, which motivates the present note. As a companion to this note, MATLAB code is released at https://github.com/stephenbeckr/exactLASSOlinesearch

Consider the LASSO problem

$$f(\mathbf{x}) = \frac{1}{2}\|\mathbf{A}\mathbf{x} - \widetilde{\mathbf{b}}\|_2^2 + \lambda\|\mathbf{x}\|_1 \tag{1}$$

for $\mathbf{x} = (x_i)_{i=1}^n \in \mathbb{R}^n$, which we will rewrite in a slightly more amenable form

$$\begin{aligned} &= \frac{1}{2}\langle \mathbf{x}, \underbrace{\mathbf{A}^T\mathbf{A}}_{\mathcal{A}}\mathbf{x}\rangle - \langle \mathbf{x}, \underbrace{\mathbf{A}^T\widetilde{\mathbf{b}}}_{\mathbf{b}}\rangle + \frac{1}{2}\|\widetilde{\mathbf{b}}\|_2^2 + \lambda\|\mathbf{x}\|_1 \\ &= \frac{1}{2}\langle \mathbf{x}, \mathcal{A}\mathbf{x}\rangle - \langle \mathbf{x}, \mathbf{b}\rangle + \lambda\|\mathbf{x}\|_1 + \text{constant}. \end{aligned} \tag{2}$$

In the process of minimizing f using gradient methods, we have a given reference point $\mathbf{x}$, and a search direction $\mathbf{p}$, where $\mathbf{p} = -\nabla f(\mathbf{x})$ if we use standard gradient descent. We can then form the 1D function φ and minimize φ to find the optimal stepsize $t^\star = \operatorname{argmin}_t \varphi(t)$ ("exact linesearch") where

$$\begin{aligned} \varphi(t) &\stackrel{\text{def}}{=} f(\mathbf{x} + t\mathbf{p}) \\ &= \frac{1}{2}\langle \mathbf{p}, \mathcal{A}\mathbf{p}\rangle t^2 + \big(\langle \mathbf{x}, \mathcal{A}\mathbf{p}\rangle - \langle \mathbf{b}, \mathbf{p}\rangle\big)t + \lambda\|\mathbf{x} + t\mathbf{p}\|_1 + \text{constant} \\ &= \frac{1}{2}c_1 t^2 + c_2 t + \lambda\|\mathbf{x} + t\mathbf{p}\|_1 + \text{constant} \end{aligned} \tag{3}$$

for constants c_1 and c_2. The optimal solution $t^\star$ will satisfy

$$\begin{aligned} 0 &\in \partial\varphi(t^\star) \\ &= c_1 t^\star + c_2 + \langle \mathbf{p}, \partial \underbrace{\|\mathbf{x} + t^\star\mathbf{p}\|_1}_{\mathbf{s}}\rangle \end{aligned} \tag{4}$$

where $\partial\varphi$ is the subdifferential of φ. Hence we need to solve the 1D equation

$$t^\star = -c_2/c_1 - \lambda/c_1\langle \mathbf{p}, \mathbf{s}\rangle \tag{5}$$

where $\mathbf{s} = \mathbf{s}(t)$. In order for f to be convex, we need $\lambda \geq 0$, and furthermore since $\mathcal{A} \succeq 0$ we have $c_1 \geq 0$, so $\lambda/c_1 \geq 0$. For convenience, we will absorb a factor of $1/c_1$ into c_2 and λ, so our optimality equation is now

$$t^\star = -c_2 - \lambda\langle \mathbf{p}, \mathbf{s}\rangle = g(t^\star) \tag{6}$$

*stephen.becker@colorado.edu

[1] "An O(n) algorithm for quadratic knapsack problems", Oper. Res. Let., **3**(3) pp. 163–166

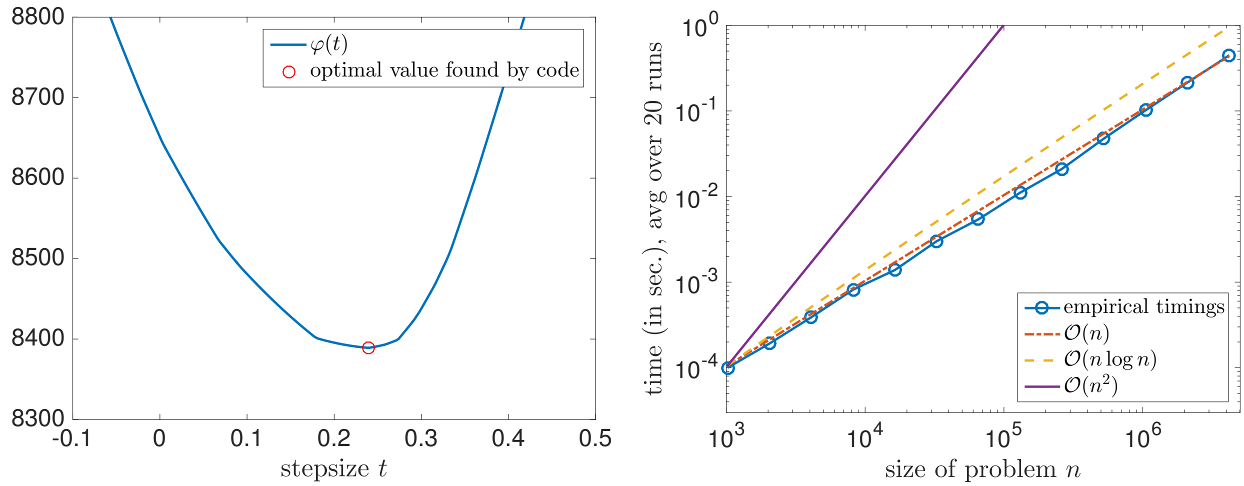

Figure 1: Left: for a sample $n = 80$ problem, the function φ and the optimal value found by the proposed algorithn. Right: for a range of n, showing the average time to solve, on problems with random data, which suggests average complexity is no more than $\mathcal{O}(n \log n)$.

for $\lambda \geq 0$ (c_2 may be any sign). We now look for a root of $t \mapsto t - g(t)$.

Since $\mathbf{s}$ is the subdifferential, it depends only on the sign of $\mathbf{x}+t\mathbf{p}$, and this sign changes only at n "turning points" given by $t_i = -x_i/p_i$ for $i = 1, \ldots, n$. For convenience, we will assume that $t_1 \leq t_2 \leq \ldots \leq t_n$; in practice, we will need to sort n numbers, then use the new sorting index to re-order other relevant quantities.[2]

For $t < t_1$, we can calculate $\mathbf{s}$ and hence $g(t)$. Denote this value of t as t_0, and $\mathbf{s}_0 \stackrel{\text{def}}{=} \mathbf{s}(t_0)$.

If we increase t to $t_1 < t < t_2$, exactly one term in $\mathbf{s}$ changes sign (from -1 to $+1$ or from $+1$ to -1), and $g(t)$ changes by $\pm 2\lambda s_1$. Moving to $t_2 < t < t_3$, exactly one more term in $\mathbf{s}$ changes sign, and $g(t)$ changes by $\pm 2\lambda s_2$. This process can be efficiently computed by pre-computing the cumulative sum of $\mathbf{p}\mathbf{s}_0$ ($\mathbf{p}\mathbf{s}_0$ being the element-wise product of $\mathbf{p}$ and $\mathbf{s}_0$), and as t moves past the next turning point, g is increased by 2λ times the next term in the cumulative sum. The cumulative sum takes $\mathcal{O}(n)$ operations.

Examining each i^{th} term of $\mathbf{p}\mathbf{s}_0$ we have

$$p_i \text{sign}\,(x_i + t_0 p_i) \tag{7}$$

and by construction, $t_0 < t_i \stackrel{\text{def}}{=} -x_i/p_i$ for all i. If $p_i \geq 0$ this means

$$x_i + t_0 p_i < x_i + (-x_i/p_i)p_i = 0 \tag{8}$$

hence $p_i \text{sign}\,(x_i + t_0 p_i) \leq 0$. If $p_i \leq 0$ then the sign is positive and we still have $p_i \text{sign}\,(x_i + t_0 p_i) < 0$. Overall, this means that the cumulative sum is monotonically decreasing in value, so as we move from one break point to the next, $g(t)$ decreases while t increases, and this enables us to quickly find the right break-point region for t. There is a chance that $t^\star$ falls exactly on a break-point, which can be checked for.

In an actual code, there are some boundary cases and concerns about underflow (since near convergence of an algorithm, $\|\mathbf{p}\|$ may be very small), which we do not describe in this note but are handled in the companion code.

[2] It may be possible, as in the case of projecting onto the ℓ_1 ball, that one can avoid the sort using median finding algorithms, since finding the median of n numbers can be done on $\mathcal{O}(n)$ time. However, this seems to have little practical use because such optimal-in-the- worst-case algorithms are seldom used, and typical efficient median finding algorithms (i.e., those with small constants and optimized implementations) are not $\mathcal{O}(n)$ worst-case, hence we see little benefit over using a sorting algorithm especially since sorting algorithms are highly optimized.