Writing a Faster, Cleaner Elastic Net Solver

Mike Chrzanowski

June 4, 2014

1 Introduction

An elastic net is an optimization problem of the form:

$$\text{minimize } \frac{1}{2} \| Ax - b \|_2^2 + \lambda \left( \eta \| x \|_1 + (1 - \eta)(1/2)\| x \|_2^2 \right),$$

where $x \in \mathbb{R}^n$ is the variable, and $A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^m$, $\lambda > 0$, and $0 \leq \eta \leq 1$ are the problem data. This problem is convex in $x$ and ubiquitous in fields such as machine learning and statistics [HZ05, HTZ06]. Instances of this problem are efficiently solved by the library glmnet.

Unfortunately, while the library is very fast, it’s written in Fortran, and the codebase consists of one 8,000 line file [FHT10]. The style of programming is very low-level, and the code is poorly commented. This combination of factors makes it nearly impossible for anyone who is not the author to understand the code, let alone contribute improvements. The goal is to implement a replacement to glmnet that incorporates its most important features, is written in a clean way with heavily commented code, and is performance-competitive.

In this paper, we present a reformulation of (1) that is easier to minimize. We then present the algorithm that solves the reformulated problem, and we discuss the features of glmnet that our library (tentatively called improved glmnet) supports. We’ll discuss various optimizations we use to increase our code’s runtime performance, and we’ll finish with benchmarks between the two libraries.

2 Problem Reformulation

At its core, the algorithm uses Newton’s method to carry out minimization. However, the cost function of (1) is not differentiable in $x$ when $\eta > 0$. So, we first add the variables $u$, $l \in \mathbb{R}_+^n$ and also add the constraint:

$$x = u - l.$$  

We thus wish to solve the following optimization problem:

$$\text{minimize } \frac{1}{2} \| A(u - l) - b \|_2^2 + \lambda \left( \eta^T(u + l) + (1 - \eta)(1/2)(\| u \|_2^2 + \| l \|_2^2) \right)$$

subject to $u, l \succeq 0,$

(2)
which is convex and differentiable in $u$ and $l$. This problem is equivalent to (1) as $u^*$ and $l^*$ are orthogonal, and hence we have $u_i^*l_i^* = 0$ for all $i = 1, \ldots, n$.

We will ignore the constraints for now. We now wish to find the gradient as well as the Hessian of the cost function. We differentiate with respect to these two variables to get:

$$
g = \begin{bmatrix} -A^T b + A^T A (u - l) + \lambda \eta 1 + \lambda (1 - \eta) u \\ A^T b + A^T A (l - u) + \lambda \eta 1 + \lambda (1 - \eta) l \end{bmatrix}.
$$

We can rewrite this as:

$$
g = \begin{bmatrix} -A^T b + \lambda \eta 1 \\ A^T b + \lambda \eta 1 \end{bmatrix} + \begin{bmatrix} A^T A + \lambda (1 - \eta) I & -A^T A \\ -A^T A & A^T A + \lambda (1 - \eta) I \end{bmatrix} \begin{bmatrix} u \\ l \end{bmatrix}.
$$

Let $K$ be the matrix in equation 3, and let $z = (u, l)$. We note that $K$ is the Hessian of the function we wish to minimize. We thus have all the components required to solve for $\Delta z^{(k)}$:

$$
K \Delta z^{(k)} = -g^{(k)}
$$

during iteration $k$ of the algorithm. We also note that it’s always the case that $K \succ 0$ when $\lambda (1 - \eta) > 0$. For the rest of the paper, we assume that $K$ has this property.

## 3 Algorithm

Bertsekas notes that there’s little reason to explicitly solve (4) [Ber82]. Instead, he observes that the first order conditions of the optimal point of (2) are:

$$
g_i \geq 0, \quad i = 1, \ldots, 2n
$$

$$
g_i = 0 \quad \text{if } z_i^* > 0, \quad i = 1, \ldots, 2n.
$$

We can solve (4) for only those variables that violate these conditions at iteration $k$, and in a finite number of iterations, we will solve (2).

Our first step is then, for iteration $k$, to find all elements for which progress might be made in terms of solving (2). This corresponds to finding elements of $z^{(k)}$ that are strictly positive or for which the gradient term is non-positive. The indices of these elements are stored in the following sorted list:

$$
\mathcal{A}^{(k)} = \left\{ i \in \{1, \ldots, 2n\} \mid g_i^{(k)} \leq 0 \lor z_i^{(k)} > 0 \right\}.
$$

Let $\hat{z}^{(k)}$ be the subvector of $z^{(k)}$ consisting only of those elements whose indices are in $\mathcal{A}^{(k)}$. We also define the subvector $\hat{g}^{(k)}$ similarly.

We turn our attention to computing the submatrix of $K$ corresponding to the indices in $\mathcal{A}^{(k)}$. In doing so, it is important for the rest of our algorithm that the submatrix we construct remain positive definite. Therefore, let $\hat{K}^{(k)}$ be the submatrix formed by taking the indices of $K$ that are in the Cartesian product $\mathcal{A}^{(k)} \times \mathcal{A}^{(k)}$. 

We then wish to solve the following subproblem:

\[-\hat{g}^{(k)} = \hat{K}^{(k)} \Delta \hat{z}^{(k)}\]  

(6)

using conjugate gradient descent [NW06]. Once we have found \(\Delta \hat{z}^{(k)}\), we then apply it to \(z^{(k)}\) using an appropriate step size \(\alpha^{(k)}\) and then project the subvector onto the non-negative orthant. This gives us the elements of \(z^{(k+1)}\) whose indices are in \(\mathcal{A}^{(k)}\). The remaining elements of \(z^{(k+1)}\) are simply copied over from \(z^{(k)}\).

That is, for all \(i = 1, \ldots, 2n\), we have:

\[z_i^{(k+1)} = \begin{cases} 
\hat{z}_j^{(k)} + \alpha^{(k)} \Delta \hat{z}_j^{(k)} & \text{if } i = \mathcal{A}_j^{(k)} \text{ for some } j \\
\hat{z}_i^{(k)} & \text{otherwise}
\end{cases}\]

Selecting \(\alpha^{(k)}\) can be done with an Armijo-like step rule [Ber82].

We now describe the pseudocode for the algorithm.

**Algorithm Elastic net solver.**

given \(z^{(0)}, \lambda, \eta\).

\(k = 0\).

repeat

1. \(g^{(k)} = \begin{bmatrix} -A^T b + \lambda \eta 1 \\ A^T b + \lambda \eta 1 \end{bmatrix} + K z^{(k)}\).

2. \(\mathcal{A}^{(k)} = \left\{ i \in \{1, \ldots, 2n\} \mid g_i^{(k)} \leq 0 \lor z_i^{(k)} > 0 \right\}\).

3. Compute subvector \(\hat{g}^{(k)}\).

4. break if \(\|\hat{g}^{(k)}\|_2 < \epsilon\).

5. Compute submatrix \(\hat{K}^{(k)}\).

6. Solve \(\hat{K}^{(k)} \Delta \hat{z}^{(k)} = -\hat{g}^{(k)}\) using conjugate gradient descent.

7. Select step size \(\alpha^{(k)}\).

8. for \(i = 1, \ldots, 2n:\)

\[z_i^{(k+1)} = \begin{cases} 
\hat{z}_j^{(k)} + \alpha^{(k)} \Delta \hat{z}_j^{(k)} & \text{if } i = \mathcal{A}_j^{(k)} \text{ for some } j \\
\hat{z}_i^{(k)} & \text{otherwise}
\end{cases}\]

9. \(k = k + 1\).

4 Library Features

4.1 Basics

improved glmnet uses the armadillo C++ library for all matrix/vector data structures and linear algebra [San10]. So, our library is compatible with dense and sparse matrices. armadillo can be compiled against a host of low-level linear algebra libraries, and so it’s just as easy to use BLAS, Intel MKL, or OpenBLAS as the core for linear algebra routines.
4.2 Regularization Path Generation

The main use case of glmnet is in generating a regularized path, meaning we solve (2) for a window of \( \lambda \) values from \( \lambda_{\text{max}} \) to \( \lambda_{\text{min}} \). Calculation of this path allows us to see the effects of regularization on our specific data set.

We note that, once we have found \( z_{\lambda}^{\star} \), we can use that point as a warm start to solve (2) for the next \( \lambda \) value in our sequence. Thus, calculation of the regularization path is very efficient.

The library glmnet evaluates a geometrically-spaced window of 100 values of \( \lambda \) where \( \lambda_{\text{max}} \) is the \( \lambda \) value for which 0 is the solution of (2). \( \lambda_{\text{min}} \) would then be \( .01\lambda_{\text{max}} \). To approximate this, we assume \( \eta > 0 \) and set \( \lambda_{\text{max}} = (1/\eta)\|A^Tb\|_{\infty} \).

4.3 \( \lambda \) Selection via Cross Validation

As with glmnet, improved glmnet can select a \( \lambda \) value that has excellent generalization performance. A subset of the dataset \( A_{\text{train}}, b_{\text{train}} \) is created, and then the regularization path for it is calculated. For each \( \lambda \), we find \( z_{\lambda}^{\star} \), and we then evaluate the cost function of (2) using this point and the remaining data set \( A \setminus A_{\text{train}}, b \setminus b_{\text{train}} \). The chosen \( \lambda \) value is then the one which achieves the lowest cost function value using the remaining data set.

4.4 Usage

Let \( A \) and \( b \) be the problem data, and let \( z \) be our initial guess.

To find \( z^{\star} \) for \( A, b, \eta, \) and \( \lambda \), one would write the following code.

```cpp
GLM *g = makeGLM(A, b, \eta);
g->solve(z, \lambda);
```

To compute the regularization path for \( A \) and \( b \), one would write the lines of code below.

```cpp
std::map<double, double> lambda_to_optval;
regularizationPath(A, b, z, lambda_to_optval, \eta);
```

To obtain the \( \lambda \) that generalizes best for \( A, b, \) and \( \eta \), the commands to invoke would be the ones below.

```cpp
double split_ratio = 0.8;
double cv_lambda = crossValidate(A, b, z, \eta, split_ratio);
```

5 Performance Optimizations

5.1 Problem Dimensionality Consideration

Since \( K \) is symmetric, it is inefficient to explicitly create it or the submatrix \( \tilde{K}^{(k)} \). Indeed, when \( A \) is fat, then it is even inefficient to compute \( A^T A \). We therefore have separate solvers that are used depending on whether \( A \) is skinny or fat.
5.1.1 SkinnyGLM Solver

Should $A$ be skinny or square, then the SkinnyGLM solver is used. This solver will explicitly compute $A^T A$ once and cache it. For the calculation of $g^{(k)}$, we can also compute and cache the bias term $(-A^T b + \lambda \eta 1, A^T b + \lambda \eta 1)$. We can then use this bias, $A^T A$, $u^{(k)}$, $l^{(k)}$, and $\lambda (1-\eta)$ to compute $g^{(k)}$.

Once we calculate $A^{(k)}$ for iteration $k$, we note that because we have explicitly calculated $g^{(k)}$, we can calculate $\hat{g}^{(k)}$ efficiently. Calculating $\hat{K}^{(k)}$ is more challenging: as we only calculate $A^T A$, which is effectively one quadrant of $K$, this requires figuring out which quadrant of $K$ a particular element in $A^{(k)} \times A^{(k)}$ would have mapped to. We thus create three submatrices: one for those elements in $A^{(k)} \times A^{(k)}$ that map to the top-left quadrant of $K$, one for the bottom-right quadrant, and then one for either the top-right or bottom-left quadrant. We will also need to add $\lambda (1-\eta)$ to the diagonal elements of the first two submatrices.

5.1.2 FatGLM Solver

When $A$ is fat, the FatGLM solver is used. This solver won’t calculate $A^T A$. Instead, we use the fact that vector operations are cheap and, for a vector $w \in \mathbb{R}^n$, we have $A^T A w = ((A w)^T A)^T$. Thus, as with the SkinnyGLM solver, calculation of $g^{(k)}$ as well as subvector $\hat{g}^{(k)}$ is very efficient.

Calculation of $\hat{K}^{(k)}$ is a bit less difficult for the FatGLM solver as we didn’t create $A^T A$. This means that our submatrices can just consist of subsets of the columns of $K$. We thus partition $A^{(k)}$ into two groups: those indices at least as small as $n$ (which correspond to indices of $u^{(k)}$), and those greater than $n$ (corresponding to indices of $l^{(k)}$). We then create two submatrices using the indices within these partitions.

5.2 Complementarity Enforcement

Our reformulation of (1) relied on the fact that, for all $i = 1, \ldots, n$, we have $u_i^{(k)} l_i^{(k)} = 0$. We note that enforcing complementarity on each iteration of our algorithm drastically speeds up its time to convergence. After calculation of $z^{(k+1)}$ during iteration $k$, we compute $x^{(k+1)} = u^{(k+1)} - l^{(k+1)}$. We then form:

$$\hat{u}^{(k+1)} = (x^{(k+1)})_+,$$
$$\hat{l}^{(k+1)} = (-x^{(k+1)})_+,$$
$$\hat{z}^{(k+1)} = \left(\hat{u}^{(k+1)}, \hat{l}^{(k+1)}\right).$$

We then use $\hat{z}^{(k+1)}$ in place of $z^{(k+1)}$. Upon examining the regularizers of (2)’s cost function, we see that using $\hat{z}^{(k+1)}$ can only help us converge to $z^*$ faster; for all $i = 1, \ldots, n$ we have:

$$\eta \left(u_i^{(k+1)} + l_i^{(k+1)}\right) + \frac{(1-\eta)}{2} \left(u_i^{(k+1)2} + l_i^{(k+1)2}\right) \geq \eta \left(\hat{u}_i^{(k+1)} + \hat{l}_i^{(k+1)}\right) + \frac{(1-\eta)}{2} \left(\hat{u}_i^{(k+1)2} + \hat{l}_i^{(k+1)2}\right).$$
6 Results

We present the runtime of glmnet and improved glmnet in calculating the regularization path for various dense, normally-distributed data sets and $\eta$ values. These tests were performed on a machine with one quad-core Intel Core i5-2400 3.1 GHz CPU, 16 GB RAM, and running Ubuntu 14.04. The results can be seen in Table 1.

We used an older version of glmnet (specifically, version 1.5) for our benchmarks that doesn’t incorporate recently-added heuristics that prematurely determine whether elements of $x^{(k)}$ will be 0 at convergence. We think this is a closer comparison against our current algorithm, which doesn’t yet incorporate these heuristics.

<table>
<thead>
<tr>
<th>A Size</th>
<th>$\eta$</th>
<th>glmnet Runtime (secs)</th>
<th>improved glmnet Runtime (secs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,000 x 1,000</td>
<td>0.1</td>
<td>10.2</td>
<td>1.82</td>
</tr>
<tr>
<td>1,000 x 1,000</td>
<td>0.5</td>
<td>14.8</td>
<td>2.9</td>
</tr>
<tr>
<td>1,000 x 1,000</td>
<td>0.9</td>
<td>16.5</td>
<td>2.9</td>
</tr>
<tr>
<td>1,000 x 3,000</td>
<td>0.1</td>
<td>5.7</td>
<td>7.0</td>
</tr>
<tr>
<td>1,000 x 3,000</td>
<td>0.5</td>
<td>5.8</td>
<td>6.1</td>
</tr>
<tr>
<td>1,000 x 3,000</td>
<td>0.9</td>
<td>5.8</td>
<td>6.3</td>
</tr>
<tr>
<td>1,000 x 12,000</td>
<td>0.1</td>
<td>26.0</td>
<td>23.4</td>
</tr>
<tr>
<td>1,000 x 12,000</td>
<td>0.5</td>
<td>20.6</td>
<td>22.94</td>
</tr>
<tr>
<td>1,000 x 12,000</td>
<td>0.9</td>
<td>20.4</td>
<td>22.76</td>
</tr>
</tbody>
</table>

Table 1: Runtime comparison between glmnet and improved glmnet.

7 Future Work

- The library is competitive with previous versions of glmnet. However, to be competitive with modern versions of the library, we need to adopt the heuristics glmnet uses to prematurely clamp variables.
- If $\eta = 1$, then $K$ might not be positive definite. So, we cannot use conjugate gradient descent to solve (6). We need to address this edge case.
- One of the other primary use cases of glmnet is in solving the logistic regression classification problem on very large data sets. We’d like to incorporate this functionality.
- We’d like to be able to leverage multiple machines to solve instances of (2).

8 Acknowledgments

I thank Hristov Paskov, my research partner (not in EE 364B), for his work in creating the algorithm we use and for notes that served as a guide for writing sections 2, 3, and 5.2.
References


