10-725/36-725: Convex Optimization **Spring 2015**

Lecture 20: Case study: generalized lasso problems (continued)

Lecturer: Ryan Tibshirani *Scribes: Mark Cheung, Haewon Jeong*

Note: *LaTeX template courtesy of UC Berkeley EECS dept.*

This lecture's notes illustrate some uses of various LaTeX macros. Take a look at this and imitate.

20.1 Review: Graph fused lasso

Essentially the same story holds (vs. linear trend filtering) when D is the fused lasso operator on an arbitrary graph:

- Primal subgradient is slow, primal prox is intractable i.e., $prox_z(\beta) = \arg\min_z \frac{1}{2t}||\beta - z||_2^2 + ||Dz||_1$
- Dual prox is cheap to iterate, but slow to converge
- Dual interior point method solves structured linear systems, so its iterations are efficient, and is preferred

20.1.1 What we need for the dual prox

- Conjugate loss of f^*
- Solve for primal solution β from dual: $-\nabla f(\beta) + D^T u = 0$

The above items are simple, but it's slow to converge.

20.1.2 What we need for interior point method

$tf^*(-D^T u) + \phi(u)$ Hessian: $tD\nabla^2 f^*(-D^T u)D^T + \nabla^2\phi(u)$ For gaussian logistic loss, $\nabla^2 f^*(-D^T u)$ and $\nabla^2\phi(u)$ are diagonal matrices, and D and D^T are structured matrices.

Reminder: for fused lasso over graph,

$$D = \begin{bmatrix} -1 & \dots & 1 \\ -1 & \dots & 1 \\ \dots & \dots & \dots \\ -1 & \dots & 1 \end{bmatrix}$$

$D^T D = L$ is the graph Laplacian, and solving this (and interior point method) can be done very efficiently.

20.2 Regression Problems

Consider the same D, and the same Gaussian and logistic losses, but with regressors or predictors $x_i \in \mathbb{R}$, $i = 1, ...n$

$$f(\beta) = \frac{1}{2}\sum_{i=1}^{n}(y_i - x_i^T\beta)^2 \tag{20.1}$$

and

$$f(\beta) = \sum_{i=1}^{n}(-y_i x_i^T\beta + log(1 + exp(x_i^T\beta))) \tag{20.2}$$

respectively. If the predictors are connected over a graph, and D is the graph fused lasso matrix, then the estimated coefficients will be constant over regions of the graph. Assume that the predictors values are arbitrary. Everything in the dual is more complicated now.

Example: x_i representing pixels of brain scan, y_i 1 or 0 indicating presence of disease. $x_i^T\beta$ regularlized according to $\sum_{l,j}|\beta_l - \beta_j|$ with l and j being neighbors.

We can rewrite the problem. For the primal case:

$$\min_{\beta} h(X\beta) + \lambda||D\beta||_1 \tag{20.3}$$

where $f(\beta) = h(X\beta)$ is the loss, and $X \in \mathbb{R}^{nxp}$ is the predictor matrix.

Solving for the dual:

Let $\alpha = X\beta$ and $z = D\beta$, so we can rewrite the primal equation as:

$$\begin{aligned} \iff \quad & \min_{\beta,\alpha,z} && h(\alpha) + \lambda||z||_1 \\ & \text{subject to} && X\beta = \alpha,\ D\beta = z \end{aligned}$$

$$L(\beta, \alpha, z, u, v) = h(\alpha) + \lambda||z||_1 + u^T(D\beta - z) + v^T(X\beta - \alpha) \tag{20.4}$$

$$= h(\alpha) - v^T\alpha + \lambda(||z||_1 - u^Tz/\lambda) + (D^Tu + X^Tv)^T\beta \tag{20.5}$$

Hence, for the dual:

$$\begin{aligned} & \min_{u,v} && h^*(v) \\ & \text{subject to} && X^Tv + D^Tu = 0,\ ||u|| \leq \lambda \end{aligned}$$

From the dual, We can derive the dual proximal gradient:

$$\begin{aligned} prox_t(u, v) = & \operatorname*{argmin}_{z,u} && \frac{1}{2t}||\begin{pmatrix} u \\ v \end{pmatrix} - \begin{pmatrix} z \\ w \end{pmatrix}||_2^2 \\ & \text{subject to} && D^Tz + X^Tu = 0,\ ||z|| \leq \lambda \end{aligned}$$

This is finding the projection of (u, v) onto the intersection of a plane and a (lower-dimensional) box. This is not a problem that we can solve in closed form.

From the dual, the dual interior point method has to respect the equality constraint, $X^T v + D^T u = 0$. And when we augment the inner linear systems, their structure is ruined, since X is assumed to be dense. The newtop step is hence much slower.

By the KKT conditions, we can see that the dual and primal are related by the following equations:

$$\nabla h(X\hat{\beta}) - \hat{v} = 0 \tag{20.6}$$

$$\iff X^T \nabla h(X\hat{\beta}) + D^T \hat{u} = 0 \tag{20.7}$$

For the gaussin loss: $h(\theta) = \frac{1}{2}||y - \theta||_2^2$ So,

$$X^T(y - X\beta) = D^T u \tag{20.8}$$

$$X^T X\beta = X^T y - D^T u \tag{20.9}$$

Computing from the dual requires solving a linear system in $X^T X$, which is very expensive for generic X.

Consider the primal subgradient method:

$$g = X^T \nabla h(X\beta) + \lambda D^T \gamma \tag{20.10}$$

where $\gamma \in ||D\beta||_1$ This works, but very slow.

In fact, for large and dense X, our best option is probably to use primal proximal gradient descent:

$$prox_t(\beta) = \arg\min_z \frac{1}{2t}||\beta - z||_2^2 + \lambda||D\beta||_1 \tag{20.11}$$

where the gradient is

$$\nabla f(\beta) = X^T \nabla h(X\beta) \tag{20.12}$$

The prox operator is not evaluated in closed-form. We can solve this with a dual interior point until convergence. We have freed ourselves entirely from solving linear systems in X.

20.3 Examples

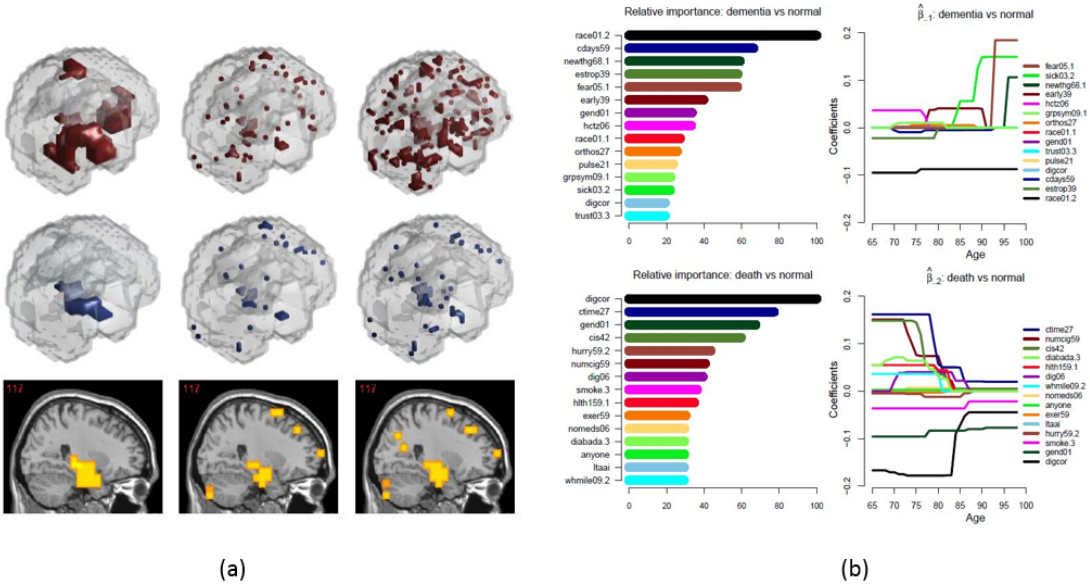

Figure 20.1: (a) Xin et al. "Efficient generalized fused lasso and its application to the diagnosis of Alzheimers disease"(2014) (b) Adhikari et al. (2015) "High-dimensional longitudinal classification with the multinomial fused lasso"

Figure 20.1 (a) shows an example of applying a fused lasso over MRI images to to predict the onset of Alzheimer's disease.

Figure 20.1 (b) is another example. In this case, an algorithm that was used is exactly what we discussed. It precisely uses proximal gradient to get rid of the X matrix and they do something clever with the prox.

You can look into [XK2014], [AL2015] for more details if you are interested.

20.4 1d Fused Lasso

20.4.1 1d Fused Lasso

Let's go to a very special case of the things we talked about. We talked about fused lasso over the graphs with regressors. What happens if the graph is just a chain? What we are doing become a 1d fused lasso. We just have to penalize the difference between adjacent nodes.

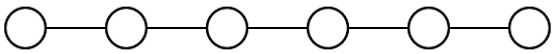

Figure 20.2: Chain Graph

We could do what we learned before: do proximal gradient to get rid of the X matrix, and solve each prox step by going to the dual and running interior point method. But in this special case, there is a very specialized algorithm for solving the prox that does not require you to go to the dual. Because of the linear structure you can see in Figure 20.2, you can use a direct method to solve the prox which is not iterative like interior point method. It's not only true for the chain graph but also for the complete graph. The point here is for certain graph structures, chain, complete, or other kind of very structured graphs, you might spend a few minutes look up the literature to see whether or not you can solve the prox in a more efficient manner than the interior point method.

$$\text{prox}_t(\beta) = \arg\min_z \frac{1}{2t}||\beta - z||_2^2 + \lambda \sum_{i=1}^{n} |z_i - z_{i+1}| \tag{20.13}$$

For this prox, the answer is dynamic programming. In some sense this method goes up to Bellman in 1960's, but there's more recent kind of formulations of dynamic programming (reference!!) and they are all linear time algorithms. Using these algorithms, we don't need to do anything which is more involved with the dual. Rather than running the generic algorithm like interior point tmethod, we can find some faster codes from the literature.

20.4.2 Performance Comparison

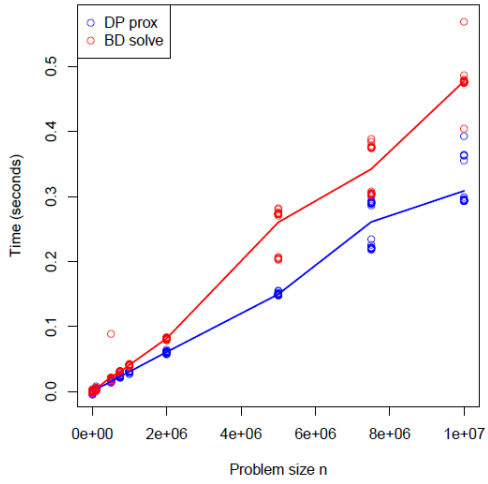

Figure 20.3: Dynamic programming vs. Banded matrix solve

Figure 20.3 is an empirical evaluation showing how fast this proximal operator is. It's order of n but it's order of n with a very small constant. You'll see how small that constant is just by comparing that proximal operator in blue to a banded matrix solve. The x-axis is a problem size, n, and the y-axis is the amount of time it took me to solve either the prox with the dynamic programming algorithm or to perform a single banded matrix solve. Since banded matrix solve corresponds to one iteration of interior point method, we're comparing the one iteration of interior point method vs. one evaluation of prox with dynamic progamming. They're both linear, but you can see that constant scaling is more favorable towards the dynamic programming. There's no reason why you should go to ineterior point method especially after you have these preliminary analysis. Also, it goes to prox directly with no worries about how accurate you are.

20.4.3 Convergence Behavior Comparison

Now we know dynamic programming is faster than interior point method, but it's even more interesting to compare convergence behavior of the two as λ varies, where λ is a regularization parameter determining the strength of regularization.

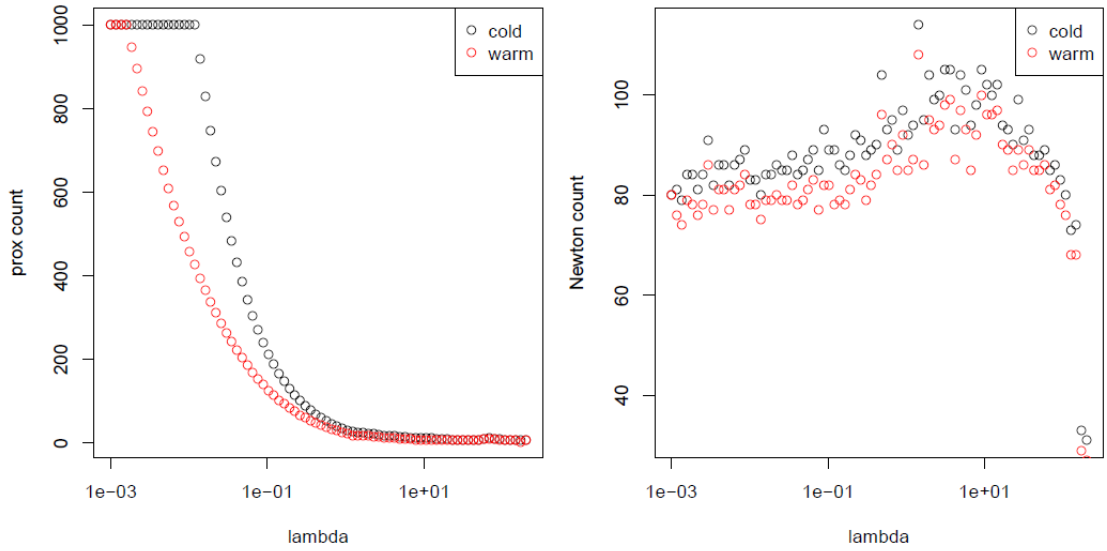

Figure 20.4: Convergence behavior comparison of dynamic programming and interior point method

If we solve the sequence of problems across of the the varying λ, we would find that the primal approach solve this prox directly quite well when lambda is large and it has increasingly more trouble when λ gets small. You can see the comparison in Figure 20.4. In this case it's solving a logistic problem without predictors and using primal proximal gradient given by dynamic programming algorithm across the greater λ's. The x-axis is the λ value, and the y-axis number of proximal operations needed to get an accuracy of certain amount, let's say 10^-5. The algorithm quits at 1000 iterations. So as λ gets smaller, the problem gets harder. We need more proximal operations to get an accurate solution. The difference between red and black is not so important so you can ignore this for now.

In the dual, the general trend is not as definite as in primal but we see that it does get easier as λ gets smaller. It's not easy to compare the two because they're not in the same scale, but let's think about drawing

a line at 100. If we use the primal approach and we solve the prox with the dynamic programming then number of prox to converge is very small for $\lambda = 0.1$ or higher. However, dual method might start winning beyond that. We can see that primal is better for a large *lambda* and dual is good for small λ.

To intuitively explain why this is the case, if you look at the primal and dual problem, the sparsity pattern is reversed between the two. In the primal, when lambda is large, there are many components that are zero, whereas for a large lambda in the dual, there are many components that are strictly inside the box. For small lambda, many components are nonzero in the primal and $D\beta$ gets denser and denser but in the dual, the box gets tighter and tighter which makes more components of the dual solution to lie on the boundary. In a very rough sense, when many of components are zero, there are fewer effective parameters in the primal. It iterates over only nonzero components, so effectively it's solving optimization problem in a smaller dimension. in the dual, it's the similar story except for now the important notion is whether they are on the boundary or not. When the λ value is samll, many components are on the boundary so the algorithm iterates over the fewer and fewer number of points. This is a very handy way of explaining but there is a more refined analysis you can show that this is actually true in terms of convergence rates. So the the point to keep in mind is that the large λ is easier for primal and the small λ is easier for dual algorithm because they have opposite notion of sparsity.

Another thing to keep in mind is that in the previous example, it seems like dual is winning when λ is smaller than 0.1, but it might not be interesting to use such small λ.

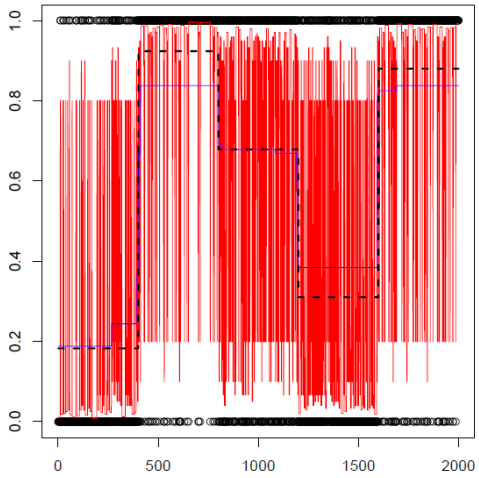

Figure 20.5: Red: $\lambda = 0.01$, Blue: $\lambda = 15$, Dotted black: true underlying mean

Figure 20.6 shows that when $\lambda = 0.01$, the optimization result is crazily under-regularized which is not interesting in any point of view.

20.5 Big Dense Matrix D

Our last example is a big dense D matrix. In this case, primal proximal approaxh is intractable, and dual is also expensive because of an expensive newton step. Solving linear systems with a dense matrix is much more expensive than one in a structured matrix. So in this case, dual proximal gradient is a good option.

$$\begin{aligned} \min_u \quad & f^*(-D^T u) \\ \text{subject to} \quad & ||u||_\infty \leq \lambda \end{aligned} \tag{20.14}$$

As you can see in the equation (20.14), it is projecting the smooth part onto the box. Hence the gradient of this is just given by

$$-D \bigtriangledown f^*(-D^T u). \tag{20.15}$$

This is possible to calculate even when D is dense. It just requires matrix multiplications. Steps of proximal gradient is just adding t times this quantity and it projects onto a box.

$$u + tD \bigtriangledown f^*(-D^T u). \tag{20.16}$$

project onto the box. D appears only when calcuating the gradient. Here, we don't have to solve any linear equation involving D, but D appears only when we are caculating the gradient, so compared to others, this method is very efficient. It is still going to suffer from an issue needing many iterations to converge to the high accuracy, but it seems like a only feasible approach in this case.

The last twist of it is when D is not only dense but is also so big that you couldn't fit it into the memory. You can't fit D into a memory for the matrix multiplication needed in equation (20.16).Now primal subgradient method is starting to look like a winner. Recall the subgradient calculation,

$$g = \bigtriangledown f(\beta) + \lambda \sum_{i \in S} \text{sign}((D\beta)_i) \cdot D_i \tag{20.17}$$

where $S = \{i | (D\beta)_i \neq 0\}$. It only needs nonzero elements of $D\beta$, se can just keep a list of which components of $D\beta$ is nonzero and only load them to the memory.

For such a big problem that we can't fit into the memory, primal subgradient method might be the best method. Even better way would be using something stochastic. You can calcuate the gradient $\bigtriangledown f(\beta)$ stochastically and also the subgradient. This is a practical method that people use for really massive problems.

We've gone through many situations where each algorithm may appear most favorable or a combination of algorithms may appear more favorable so that you can get a sense for an idea that no one algorithm dominates the other.

20.6 Takeaway Points

We looked in to a generalized lasso problem to go through these points but these ideas are generally applicable to any kind of problem.

- There is no single best method

  The performance gonna depend structure of penalty, or the regularizor you're using. if you're going to the dual it depends on the conjugate of the loss, is that possible to derive? depends on what accuracy level you want, depends on the sought regularization level - λ.

- Duality is your friend

  You should always derive the dual even if you're confident how to solve the problem in the primal. It's never going to hurt. Just derive the dual and see what you can get out of it. What does the dual do? It takes the linear transformations of the one part of the criterion function and moves a non-smooth regularizor into the smooth criterion function. The dual also offers success in different reg regimes at different levels of regularization parameters because of that high-level reasoning i gave before.

- Regressors complicate duality

  Again, this is a persistent point. It does not have to do with just generalized lasso, but always true. Regressors are linear transformations in the smooth part of the loss and the dual moves them to the non-smooth part of the loss. we have a loss + regularizor and that's gonna complicate the role played by the non-smooth part in the dual. One way to get around this is to use proximal gradient in the primal and that will completely relive yourself of dealing with problems involving X, because you're taking the primal problem and you're approximating the smooth part by quadratic that does not involve X at all. That's a generic strategy to reduce the problem with the regressors to one without regressors.

- Recognize the easy sub-problems

  If there's something that is a special case of your problem, you can recognize the easy subproblem to solve and design the algorithm around it. That's what we in 1d fused lasso where we recognized that prox was really quite simple - dynamic programming over a line - and we designed the algorithm around that. We decided to use proximal gradient because the prox was so efficient. This is also important for ADMM which we will learn later.

- Limited memory at scale

  At scale, memory limitation problem becomes very serious so the active set/stochastic methods may be the only option you have. If you have a really large problem, method that deal with active sets and evaluate gradients and subgradients stochastically will work. We didn't stress this point at all in this class because it is more system-side work, but stil very important point to keep in mind.

- You dont have to find/design the perfect optimization algorithm

  As we go forward and learn more advance methods like ADMM, our toolbox will get bigger. All of them will be a viable option and may even be a better option than what we learned today depending on the situations. Finding the optimal algorithm among them may be interesting but it is very hard to exhaustively examine all the methods we learned. Just find the one that works well is really the point. Especially when you don't care about the performance at all and you just wanna do some prototyping, you can just use convex programming package such as `cvx` and `tfocs`.

20.7 Implementation

Now we want to focus on implementation which we didn't focus on at all. It's mostly engineering concern, but implementation skills are really under-valued. Here are some ideas to keep in the back of your head if you're implementing an algorithm like the ones we talked about today.

- Speed

  It's obviously an important point and we've been focused on this a lot so far.

- Robustness

Robustness refers to the stability over the various cases. For example, let's consider a fused lasso over the graphs where graphs can have the edge weights. Robustness refers to the idea how well it performs if some weights are huge and some are tiny.

- Simplicity

  These are rules of thumbs. A constant-factor speedup is probably not worth a much more complicated implementation. If it makes much more complicated code and gives you two times speed up, it's not worth it. It may be worth a loss of simplicity if you are able to make an implementation to converge to a higher degree of accuracy quickly, say $1/k$ to $1/k^2$ or c^k.

- Portability

  Portability refers to the idea of how well it can be ported to different problems or different languages. Writing a code in a lower level language like `C` is a good practice especially when you consider portability. Almost every language can take the code written in `C` and be wrapped around in like `R`, `MATLAB`, `Python`.

My last tip on the implementation is DO NOT re-implement standard routines. You don't have to reinvent the wheel every time you solve the optimization problem. This is especially true for numerical algebra. When i was a graduate student, i spent way too long time implementing updates for QR decomposition. When i look back, I wish somebody had told me don't do that. There's a tons of open-source free resources for the numerical algebra. Reimplementing things on your own makes you prone to bugs and it makes much harder to port to other languages and just not worth your time. You should always be looking at where you can find the high quality implementations of standard routines.

References and Further Readings

[XK2014] B. Xin, Y. Kawahara, Y. Wang and W. Gao (2014), "Efficient Generalized Fused Lasso and Its Application to the Diagnosis of Alzheimers Disease"

[AL2015] S. Adhikari, F. Lecci, J. T. Becker, B. W. Junker, L. H. Kuller, O. L. Lopez, and R. J. Tibshirani (2015), "High-Dimensional Longitudinal Classification with the Multinomial Fused Lasso"