Comparing Algorithms for the Sparse Group Lasso

Aaron Cohen

December 16, 2020

Abstract

In this paper we explore two algorithms for the sparse group lasso.

1 Introduction

The lasso is "a regression analysis method that performs both variable selection and regularization" which uses an $l1$ penalty [3] to select a subset of predictors to be nonzero. The minimization problem is of the form

$$\min_{\beta} \frac{1}{2} \|y - X\beta\|_2^2 + \lambda \|\beta\|_1 ,$$

where X is an n by p data matrix, β is the unknown coefficient vector, and λ is a hyperparameter that enforces the regularization/variable selection. In this problem it is assumed that the response y is linearly related to the predictors (the columns of the data matrix X), with the goal being to estimate the coefficient vector β responsible for this linear relationship. In general it will return a solution with some nonzero entries in the β vector, and the rest zero.

A variant of this, the group lasso [6] is appropriate when there is a natural grouping structure for the coefficients; that is, we assume that the vector of coeffiecients is partitioned into groups, or subvectors, and, by analogy with regular lasso regression, only a few of the groups are active, i.e., have nonzero coefficients. The group lasso thus performs regularization that has the effect of discarding groups of predictors rather than the predictors themselves; it is of the form

$$\min_{\beta} \frac{1}{2} \left\| y - \sum_{l=1}^{m} X^{(l)} \beta^{(l)} \right\|_2^2 + \lambda \sum_{l=1}^{m} \sqrt{p_l} \left\| \beta^{(l)} \right\|_2 .$$

Note that the grouping structure is explicit in the above equation: the vector of coefficients, β, is thought of as a concatination of the coefficient subvectors of the various groups $\beta^{(l)}$, and similarly the data matrix X is the concatination of submatrices, each submatrix $X^{(l)}$ being composed of the columns that correspond to the particular group. Thus the first part of the equation, $y - \sum_{l=1}^{m} X^{(l)}\beta^{(l)}$, is identical to the more simply-written equation $y - X\beta$, but is written with this partition in mind.

The second part of the equation, however—$\sum_{l=1}^{m} \sqrt{p_l}||\beta^{(l)}||_2$—is different than the corresponding part in the original lasso equation. Rather than being an $l1$ norm of the vector β, it is a sum of the (non-squared) $l2$ norms of the coefficient vectors of the various groups. It is interesting to note that it is the non-differentiability of this expression at 0 that accounts for the group-discarding property of the problem, similar to how the $l1$ norm's non-differentiability at 0 is responsible for the discarding property of coefficients in the original lasso.

As with the original lasso equation, there is only a single tuning parameter λ, whose value determines the strength of regularization. Within the second summation are the relative weights of the groups, p_l, though these are usually determined by the size of the corresponding groups, and so in the rest of this paper are not considered as a tuning parameter.

Finally, in a group-structured problem as above, it may be desirable to enforce sparsity, not only among the groups, but also within the groups. The sparse group lasso [2] does this by solving the following minimization problem

$$\min_{\beta} \frac{1}{2} \left\| y - \sum_{l=1}^{m} X^{(l)} \beta^{(l)} \right\|_2^2 + (1-\alpha)\lambda \sum_{l=1}^{m} ||\beta^{(l)}||_2 + \alpha\lambda||\beta||_1. \tag{1}$$

Note that the factors $\sqrt{p_l}$ have been suppressed–see the previous paragraph. This equation is very similar to that of the group lasso, and in fact one sees that the only difference is that it adds a second regularization term, the overall $l1$ norm of the full β vector, which we first saw in the original lasso formulation. There is now a second tuning parameter α, which controls the relative emphasis of intra- vs inter- sparsity in the predictors.

The sparse group lasso, like the other two lassos, does not have a closed form solution and thus relies on computational algorithms, which may be very intensive for large datasets and/or many values of λ. The rest of this paper therefore concerns the mitigation of this issue, in particular the following question: if we are solving the sparse group lasso, not for a single value of the tuning parameter λ but a whole parameter space—in this context, a range of values $(\lambda_1, \dots \lambda_s)$—then can we shorten the total computational time by

making use of the already-computed solution at previous λ's in the parameter space to speed up computation at a given λ?

In this paper we make use of a heuristic called the Strong Rule [4] to help solve this problem, with some success. The strong rule uses the solution at a previous λ_{k-1} to predict which groups will remain inactive upon solution at the current λ_k, and discards those groups before entering into the algorithm. If a significant number of groups are thrown out before entering the algorithm, convergence time can improve significantly.

The `R` package we have create has two algorithms, both using the strong rule, but in slightly different ways: the three-step algorithm and the four-step algorithm. Although they perform similarly for the simulated data set used in later sections of this paper, it is suspected that they will each be useful for different scenarios, depending on the size of the data, size of the predictors, and the true underlying group structure. This is explored in the last section before the conclusion.

There are already two other existing `R` packages that perform group lasso, gglasso and sgl. The advantage of our package over those two are as follows: the sgl package has sparsity, but does direct coordinate descent and does not have a heuristic like the strong rule, so it does not take advantage of such a computational speedup. The gglasso package, on the other hand, incorporates such a rule and is computationally fast, but does not incorporate within-group sparsity. Our contribution, then, is to provide a package that performs sparse group lasso, and is faster than existing algorithms. This algorithms for this package are written in Fortran, and are based on the algorithm from the gglasso package.

In section 2, we describe the algorithm in detail, paying particular attention to the strong rule. In section 3, we show how to use the package, running through an example with simulated data. In section 4, we compare our package to the other existing sgl packages, and also compare the two variants of our algorithm with each other.

2 Methodology

2.1 Overview of the algorithm

There is no closed-form solution to the optimization problem in Equation 1 above, so we need a numerical algorithm to find the optimal solution. It can be shown that the problem defined by Equation 1 is convex, so there is a unique optimal solution, and so a variety of methods may be used.

Our package has two different algorithms to accomplish this, which we refer to as the three-step and four-step algorithms. The general framework for both of our algorithms is based on a majorized coordinate descent algorithm (see [2, 5]). What this mean is that we loop over the groups and, for a given group, update only those variables while holding all other groups constant. The term 'majorized' comes from the idea that, instead of using the exact equation to determine the step size and direction in every update step, we update according to a simpler expression than Equation 1 that majorizes, or bounds from above, the expression we want to solve for. This is explained in more detail in the next paragraph.

To begin with, consider the first term in Equation 1, the quadratic loss $\frac{1}{2}\left\|y - \sum_{l=1}^{m} X^{(l)}\beta^{(l)}\right\|_2^2$, by itself; we will denote this by ℓ for the rest of this section. Since this is a quadratic function in β, it is equal to its second order Taylor expansion about any point β^* in the parameter space—indeed, this is the definition of a function being quadratic. We thus start with the following equality for any given β^*:

$$\forall \beta,\ \ell(\beta) = \ell(\beta^*) + (\beta - \beta^*)^T \nabla \ell(\beta^*) + \frac{1}{2}(\beta - \beta^*)^T H (\beta - \beta^*), \tag{2}$$

where the gradient $\nabla(l)$ is the first total derivative of ℓ (evaluated at β^*) and H, the Hessian, is the second total derivative. Recall that ∇ is the vector of first partial derivatives of ℓ with respect to each β_i , and H is a pxp matrix of mixed partial derivatives of ℓ. In this particular case, a short computation shows that the Hessian is simply written in terms of X: $H = X^T X$.

Already there is a problem, and an opportunity. The matrix X is, for the problems we are interested in solving, very large, and so computing $X^T X$, storing it in memory, and inverting, is going to be an issue, especially since we will be doing update steps many times. What we can do, and will do, is replace this matrix with a much simpler one, tI, a diagonal matrix with the value of t selected to be such that this dominates the Hessian (in the sense that $tI - H$ is positive definite). For our algorithm we choose the

largest eigenvalue of the Hessian (ask Dan/think when not so tired) and use that for t. We get the following inequality:

$$\forall \beta, \beta^*,\ \ell(\beta) \leq \ell(\beta^*) + (\beta - \beta^*)^T \nabla \ell(\beta^*) + \frac{t}{2}(\beta - \beta^*)^T(\beta - \beta^*). \tag{3}$$

We take the original minimization problem (Equation 1) and replace the loss with the right-hand side of Equation 3. Thus Equation 1 is now dominated by the following expression

$$\ell(\beta^*) + (\beta - \beta^*)^T \nabla \ell(\beta^*) + \frac{t}{2} \left\|\beta^* - \beta\right\|_2^2 + (1-\alpha)\lambda \sum_{l=1}^{m} \left\|\beta^{(l)}\right\|_2 + \alpha\lambda \left\|\beta\right\|_1, \tag{4}$$

which can be solved.

The general idea at this point is that we pick a starting point β^* in the parameter space, find a much simpler function that dominates it, then minimize that function by setting the derivative to zero and solving for β. This becomes the new β^*, and we repeat the whole process. There is a complication, which is a blessing in disguise, in the fact that the last two factors are non-differentiable; however, we can still implement this update-step process by using sub-differentials.

[Need to talk about sub-differentials.]

[Need to go from talking about β to talking about $\beta^{(l)}$]

The resulting update step is that $\beta^{(l)}$ is replaced by

$$\beta^{(l)} \leftarrow \Gamma(\beta^{(l)}) = \left(1 - \frac{t(1-\alpha)\lambda}{||S(\beta^{(l)} - t\nabla\ell(r_{-l}, \beta^{(l)}),\ t\alpha\lambda)||_2}\right)_+ S(\beta^{(l)} - t\nabla\ell(r_{(-l)}, \beta^{(l)}), t\alpha\lambda) \tag{5}$$

where S is the coordinate-wise soft-threshold operator $(S(a,b))_j = \text{sign}(a_j)(|a_j| - b)_+$. *For ease of notation, denote the right side of Equation 5 by* $\Gamma(\beta^{(k)})$. Note that in updating $\beta^{(k)}$ it is possible for the whole group to be set to zero (made inactive) because of the threshold operator $()_+$ in the first part of the expression, and/or for individual components of $\beta^{(k)}$ to be zeroed out from the coordinate-wise threshold $S()$. So this algorithm setup performs variable selection at both the group and individual level.

Because the optimization problem is convex, it can be shown that this type of descent algorithm is guaranteed to converge to the optimal solution. That is, it does not get 'stuck' at any inflection points. Therefore, if that was all the algorithm did, there would be no need to check the KKT conditions for optimality.

However, our algorithms both make use of the sequential strong rule, a heuristic that predicts, before computation, which groups will remain inactive, and throws them all out; this results in significant computational savings, but because of this pre-processing prediction, we will need to do a KKT check [1] to make sure the prediction was accurate. This is described in the next section.

2.2 Strong Rule

As mentioned above, the strong rule [4] is a heuristic that is easy and fast to perform, throwing away large numbers of predictors before computation, but it is not guaranteed to be correct, so it becomes necessary to double-check the solution with the KKT check afterwards.

It is important to note first of all that the version of the strong rule used here, the sequential strong rule, makes critical use of the fact that we are solving for a sequence of λ parameters. At each λ_k, we rely on the fact that we have solved the problem for the previous λ_{k-1}, and use this information to quickly discard many predictors. Without loss of generality, for the rest of this section assume that the problem has already been solved for the previous λ_{k-1}.

The motivation for the strong rule comes from the KKT check, so we start with this. The KKT stationarity condition [1] is essentially the first-derivative test from calculus, and for this problem is

$$||S(X^{(k)T} r_{(-k)}/n,\ \alpha\lambda)||_2 \leq (1-\alpha)\lambda.$$

The left-hand side of this inequality is the norm of the (sub)gradient of the loss $\ell(r_{(-k)}, \beta)$, where $r_{(-k)} = y - \sum_{\ell \neq k} X^{(l)}\beta^{(l)}$. Denote

$$c_j = c_j(\lambda) = S(X^{(j)T} r_{(-j)}/n,\ \alpha\lambda),$$

i.e., c_j at a given λ in the sequence is the subgradient with respect to the j-th group. The (sequential) strong rule checks if the following inequality holds, and discards the j-th group if so:

$$|c_j(\lambda_{k-1})| < (1-\alpha)(2\lambda_k - \lambda_{k-1}) \tag{6}$$

The reasoning is as follows: consider $c_j(\lambda)$ as a function of lambda. The key point is that, if this function c_j were $(1-\alpha)$-Lipschitz, by definition it would satisfy

$$|c_j(\lambda) - c_j(\tilde{\lambda})| \leq (1-\alpha)|\lambda - \tilde{\lambda}|$$

Now suppose it were Lipschitz and also that $c_j(\lambda_{(k-1)}$ were known, i.e. we have solved the problem for the previous lambda in the sequence. Then, when the strong rule is satisfied, we combine the two previous inequalities to get

$$|c_j(\lambda_k)| \leq (1-\alpha)(|c_j(\lambda_k) - c_j(\lambda_{k-1})| + |c_j(\lambda_{k-1})|) < (1-\alpha)(\lambda_{k-1} - \lambda_k + 2\lambda_k - \lambda_{k-1})$$

which is precisely the KKT stationarity condition at λ_k. Thus, if the strong rule condition were satisfied at this group (and assuming the Lipschitz condition, which is not always true), it follows that β_j is inactive for the current lambda, λ_k. Since we made the Lipschitz assumption, which essentially says that c_j does not change "too fast" as a function of λ, we need to check after performing the algorithm on the other groups that this group is indeed inactive. In fact, as shown in the strong rules paper (), it is very often the case that the c_j's are $(1-\alpha)$-Lipschitz.

2.3 The three-step and four-step algorithm

Algorithm 1 Three-step Algorithm

Input: $\Lambda = \{\lambda_0 > \lambda_1 > \cdots > \lambda_K\}$, $\mathcal{E}$, $\mathcal{A}$
Set $\mathcal{E} = \mathcal{A} = \emptyset$; set $d = \epsilon + 1$
Step 1: $\mathcal{E} \leftarrow \mathcal{E} \cup \mathcal{S}(\lambda_{k-1},\ \mathcal{E}^c)$
Step 2:
while $d > \epsilon$ **do**
  for $l = 1$ to m **do**
    $\beta^{(l)} \leftarrow \Gamma(\beta^{(l)})$
    If $\beta^{(l)} \neq 0$ put $\beta^{(l)} \in \mathcal{A}$
  end for
  $d \leftarrow \max_l \left\| \beta_{k-1}^{(l)} - \beta_k^{(l)} \right\|$
end while
Step 3: $\mathcal{E} \leftarrow \mathcal{E} \cup KKT(\mathcal{E}^c)$; if $KKT(\mathcal{E}^c) \neq \emptyset$, go back to step 2.
return β vector for λ_k
$\lambda_k \leftarrow \lambda_{k+1}$
Return to step 1

There are two algorithms in our package that perform sparse group lasso: the three-step algorithm and the four-step algorithm. Both algorithms use the majorized coordinate-descent update step, and mainly differ in how they use the strong rule and how they store the active and potentially active groups. Before discussing these in detail, we need some notation.

First, let $\mathcal{S} = \mathcal{S}(\lambda,\ A)$ be a function which performs the strong rule check on the set of groups A at lambda value λ, and returns the set A' of groups in A that pass the strong rule check. For each individual group in the set A, it checks the inequality in Equation 6. Note the strong rule check at a given λ_k involves both that lambda and the previous one in the sequence; this is suppressed in this notation, with the understanding that the lambda that is passed into $\mathcal{S}$ is the current lambda λ_k.

Similarly, the function $KKT = KKT(\lambda, A)$ performs the KKT check described in the previous section, and returns the subset A' of groups that fail the KKT check. The idea for both of these functions is to keep track of which groups are potentially active (nonzero) for the given λ.

Note that both of these functions take in only a subset of the groups of coefficients, rather than being run on the whole set each time; this is to save time and reduce redundancy. It would not make sense to run a KKT check on those groups that are already known to be active (for the given λ), because the whole point is to figure out which groups are mistakenly put down as 'inactive'. Similarly for the strong rule check, it would be a waste of time to run that inequality on groups that are already known or suspected of being active, since the point of the strong rule check is to separate those suspected of being active from those that are not, and only updating the coefficient estimates for the suspected active groups.

We need to keep track of the active and suspected active groups. First, let $\mathcal{E}$ be the set of groups that are suspected of being active at a given lambda; this is the subset of groups that we will perform the update step on, repeatedly until convergence. It is important to note that there is no mention of λ for this set—this set is only increasing in membership, and gets passed on from one lambda to the next. In other words, once a group is suspected of being active and gets put in $\mathcal{E}$, it never leaves this set, and if group l is put in $\mathcal{E}$ for λ_{k-1}, it is automatically included in this set when we start the algorithm for the next value, λ_k.

The set $\mathcal{A}$ is the set of groups that are actually active, i.e. nonzero, for a given λ. This set is also never-decreasing, so if a group becomes active for one lambda, it stays in that set for every subsequent lambda. This set is used in addition to $\mathcal{E}$ in the three-step algorithm, because we use a warm-start for β^* every time we progress to the next λ, and we only want to keep track of the small set of nonzero coefficients for the warm start. This set is not used in the four-step algorithm.

let $\mathcal{E}$ be the set of indices for the groups that are potentially active (nonzero). There are two ways in which groups get added to $\mathcal{E}$: either they pass the strong rule check

First we discuss the three-step, Algorithm 1.

In the initialization (step 0) of the algorithm, we set the lambda path (the decreasing set of λ 's which comprise the parameter space), and start keeping track of two sets, $\mathcal{E} = \emptyset = \mathcal{A}$, which are initially empty. Both of these sets grow as the algorithm is run; predictors get put into these groups if they pass certain checks, but never leave once they are put in.

Without loss of generality assume that we have run the algorithm for a few lambdas, and are at $\lambda = \lambda_k$. The first step (step 1) of the algorithm is to run the strong rule check on all predictor groups in $\mathcal{E}^c$, and every group that passes the strong rule (which is computed using the gradient at the previous lambda) gets put in the set $\mathcal{E}$. So $\mathcal{E}$ is essentially keeping track of every predictor that has ever passed the strong rule check–these are candidates for active groups.

In the second step (step 2), we loop over $\mathcal{E}$, performing the update step until convergence. Again, this is a type of coordinate descent, but we are saving a lot of time by keeping all groups in $\mathcal{E}^c$ zero and only updating on those suspected of being active. Now, although $\mathcal{E}$ is keeping track of the 'strong set', any time a group actually becomes active (i.e. is updated to be nonzero), it gets put in $\mathcal{A}$, the 'active' set. This is because while we save the indices of the potentially-active groups, we only save the values of the actually active groups.

Finally, after we have converged (the update step for all groups is changing values by less than a small epsilon), we need to perform the KKT check to make sure that we didn't accidentally leave some active groups at 0. Again, if we were not using the strong rule check, this would be unneccessary. The only potential violators are in the complement of $\mathcal{E}$, so: run the KKT check on $\mathcal{E}^c$, throw any violators into $\mathcal{E}$ and go back to step 2. If there are no violators, we have found the solution at the given λ, so we save the values in $\mathcal{A}$, as well as the gradients, and move on to the next lambda.

The four-step algorithm, Algorithm 2, has a slight refinement over the three-step. It is as follows. First note that there is only one set to keep track of, the set $\mathcal{E}$. The setup is the same.

Assume we have solved the problem for λ_{k-1}. We loop over $\mathcal{E}$ and update until convergence. The difference here is that we do not perform the strong check first. After we have run the update loop until convergence on our set, we then perform the strong rule on the complement, $\mathcal{E}^c$. For any groups that pass the strong rule, we run the kkt check; if there are any violators, put them in $\mathcal{E}$ and go back to step 1.

Otherwise, we run the full KKT check, the KKT check on the full complement $\mathcal{E}^c$. Again, if there are any violators, we put them in $\mathcal{E}$ and go back to step 1. If there are no violators, we are done, and we move on to the next λ in the sequence.

Algorithm 2 Four-step Algorithm

Input: $\Lambda = \{\lambda_0 > \lambda_1 > \cdots > \lambda_K\}$, $\mathcal{E}$
Set $\mathcal{E} = \emptyset$; set $d = \epsilon + 1$
Step 1:
while $d > \epsilon$ **do**
  for $l = 1$ to m **do**
    $\beta^{(l)} \leftarrow \Gamma(\beta^{(l)})$
    If $\beta^{(l)} \neq 0$ put $\beta^{(l)} \in \mathcal{A}$
  end for
  $d \leftarrow \max_l \left\| \beta_{k-1}^{(l)} - \beta_k^{(l)} \right\|$
end while
Step 2: Compute $S = \mathcal{S}(\lambda_{k-1}, \mathcal{E}^c)$
Step 3: $\mathcal{E} \leftarrow \mathcal{E} \cup KKT(S)$; if $KKT(S) \neq \emptyset$, go back to step 1.
Step 4: $\mathcal{E} \leftarrow \mathcal{E} \cup KKT(\mathcal{E}^c)$; if $KKT(\mathcal{E}^c) \neq \emptyset$, go back to step 1.
return β vector for λ_k
$\lambda_k \leftarrow \lambda_{k+1}$
Return to step 1

3 Example

In this section we provide a worked-through example of how the package is used.

First, we generate some random data. There will be $n = 100$ observations and $p = 200$ predictors. The predictors will be partitioned into groups of 5, so e.g. the first five predictors form the first group, and so on. Some of the groups will be inactive (0), and within a given active group, some of the variables will be 0. The goal is to learn exactly this information.

Once we make a random design matrix X and the true β vector (which has the group structure mentioned above), we form the response vector y according to $y = X\beta + \epsilon$. The following code generates the data for this example.

```
set.seed(1010)
n <- 100
p <- 200
X <- matrix(data = rnorm(n*p, mean=0, sd=1), nrow = n, ncol = p)
eps <- rnorm(n, mean = 0, sd=1)
beta_star <- c(rep(5,5), c(5,-5,2,0,0), rep(-5,5), c(2,-3,8,0,0), rep(0,(p-20)))
y <- X%*%beta_star + eps
groups <- rep(1:(p/5), each=5)
```

Since we generated the data according to the above equation, we know the true value of β, in particular which groups are active and which predictors within active groups are nonzero. We want to use the sparsegl package to backsolve: given X, y and the group partition information (how the predictors are divided up into groups, but not the values), how can we best estimate the predictors β?

To reiterate, we need to know beforehand how the predictors are divided up into groups; the algorithm takes that in, and does not learn it. Also note that the groups in β are contiguous. The data needs to be grouped in such a way before we can apply our package.

This package will estimate β for a whole path of λ weights. For each λ, it will find the optimal solution for that weight, according to equation ¡blank¿. The lambda path can be explicitly specified as an input, but otherwise it creates its own set of λ, first finding the smallest value of λ that forces all predictors to be zero, and decrementing logarithmically 100 times, to get a set of 100 decreasing λ's. For this particular dataset, the lambdas range from 0.636 to 0.00636, with log-constant decremental step-size.

In addition to lambda we also need to set α, which controls the relative emphasis on inter- vs intra-group sparsity. The default is $\alpha = 0.05$. Here is the code that runs the algorithm and stores the results in an object called mysparse1.

```
mysparse1 <- sparsegl(x = X, y = y, group = groups,
```

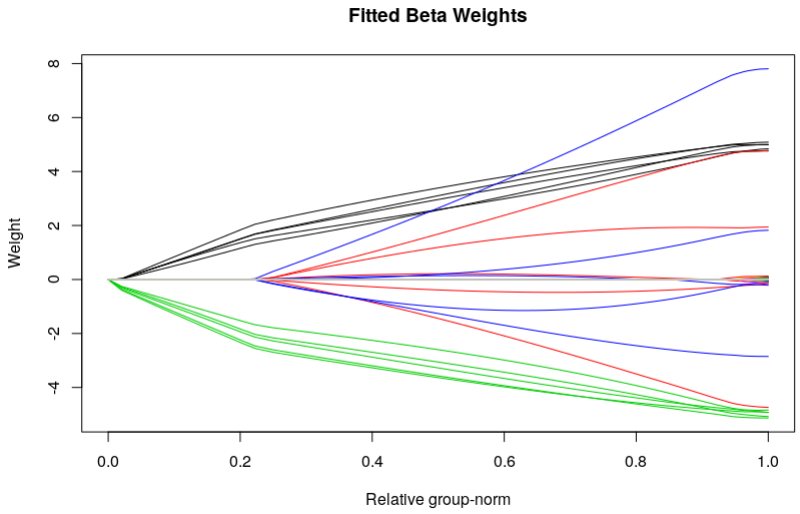

Figure 1: Graph of predictor weights, colored by group, against the relative group norm. This shows how the groups and predictors enter and exit the model as a function of the λ parameter.

```
pen = "sparsegl", algorithm = "threestep")
```

With the above line of code, we have run the algorithm that solves the problem for the default lambda space, and created a sparsegl object. The main data in that object is the B matrix, which gives the estimated beta vector for each value of λ. Note that, when calling this function, we must give it the group structure, and also specify the threestep vs the fourstep algorithm. Finally, the 'pen' flag specifies that we want to perform sparse group lasso; there is also the option to just perform group lasso.

If we look now at the B matrix, we see that, for a particular range of lambdas, the algorithm was able to deduce correctly the active groups, the active predictors within those groups, and accurately estimate their weights.

¡¡see beta matrix¿¿

It is illustrative to look at a graph of the predictors against lambda, to see when the various predictors become active. For visualization reasons, we instead plot on the x-axis, not lambda but an equivalent measure, the relative group norm. This is equivalent to graphing against lambda, since as lambda decreases, the group-norm (which is the sum of L_2 norms of all the groups in beta) of the fitted beta increases. This type of graph is produced by a built-in function of the sparsegl package. All predictors are plotted, and each are colored by group membership, which is supplied as input. See the figure below.

We can see that there are four active groups (all other groups remain 0 for all values on the x-axis): black, green, red and blue. By comparing this graph with the true beta in the code above, we can get a measure of how the algorithm is performing across the various lambda values.

The black group (which we see has 5 members) quickly becomes active as lambda is decreased, and eventually all values settle at a value of 5. This is evidently the first group, which was set to five 5's. Similarly, the green group becomes active and converges to five -5's; this is third group. The other two groups are active but have predictors that are 0–we can see that the red group, for instance, has a 5, a 2, and a -5, and must be the second group.

From comparing this output graph with the true β, with which the data was generated, one might ask what the optimal λ, is, which recovers the true structure. In this case, it appears the small end of the lambda path results in the most accurate estimates.

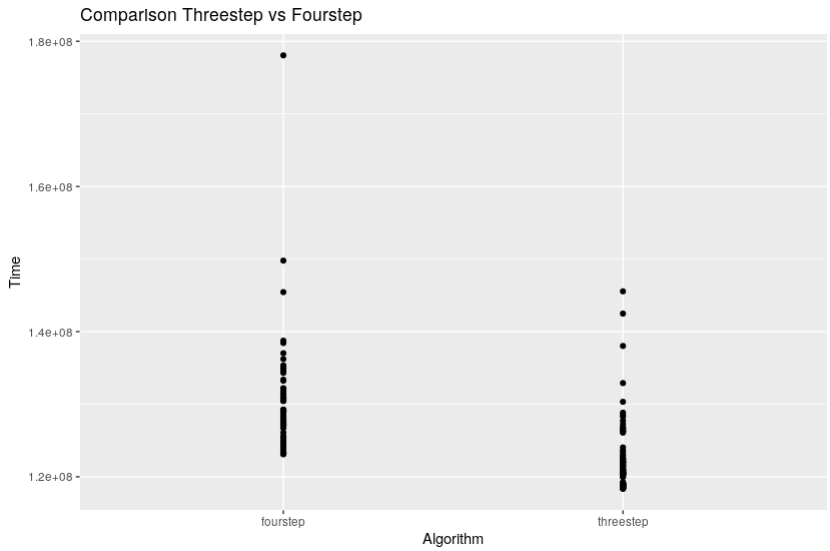

Figure 2: A graph comparing the run-times of the threestep and fourstep algorithms. The fourstep algorithm is slightly faster on average.

4 Comparison

First we will compare the two algorithms within the sparsegl function, threestep and fourstep, for both accuracy and speed. Then, we compare our package with two other competing packages, the SGL package (which performs sparse group lasso, without predictor-discarding rules) and the gglasso package (which only performs group lasso, but uses predictor-discarding rules). Since the main contribution of this package is a fast algorithm for sparse group lasso, we pay particular attention to the speed difference between our algorithms and SGL.

For all comparisons, we use the R package microbenchmark, which compares two or more operations by performing them repeatedly (100 for our analysis) and collecting the times to perform each operation.

We can see that there is not a big difference in the time it takes to perform the threestep algorithm vs the fourstep algorithm. This is possibly a result of the simulated data used, and we believe that for some large datasets, the fourstep algorithm will outperform the threestep.

Next, we compare our package with the gglasso package, which performs regular group lasso. For this comparison we set α to be zero in our algorithm, so that it also performs regular group lasso.

We see that our package actually outperforms gglasso, even though the gglasso incorporates a similar strong rule, and is overall very close to our threestep algorithm. This gives evidence that the fourstep algorithm can outperform the threestep in at least some circumstances. So even without considering the added functionality of sparsegl to incorporate sparseness, our sparsegl algorithm seems to be an improvement on the gglasso package for performing group lasso.

Finally, we look at the only other existing package that performs sparse group lasso, the SGL package.

Here we see that our fourstep algorithm is substantially faster than SGL in performing sparse group lasso. In fact, we needed to decrease the number of times that microbenchmark ran the two algorithms, since the default of 100 tests was too much for this computer, it did not finish the job in an hour.

Recall the dataset we used for these comparisons, with $n = 100$ and $p = 200$. It is clear that SGL cannot be used for very large datasets, and thus the sparsegl package represents real progress.

We also compare the actual output, in particular the beta matrix, to confirm that they are arriving at the same answers. For this comparison, the lambda path is about the same, but for the other packages (gglasso, SGL), the lambda path will in general be different, so some care must be taken in comparing the beta matrices.

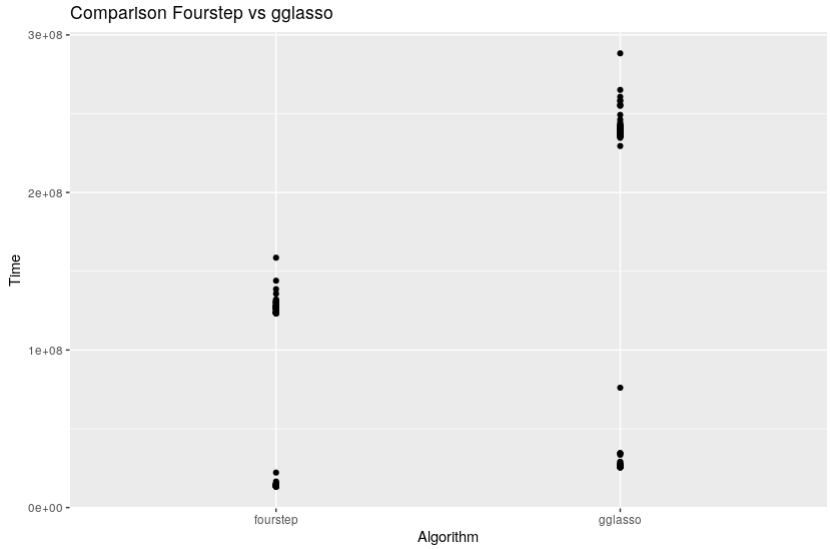

Figure 3: A graph comparing the run-times of the fourstep algorithm with the gglasso package. The fourstep algorithm is slightly faster.

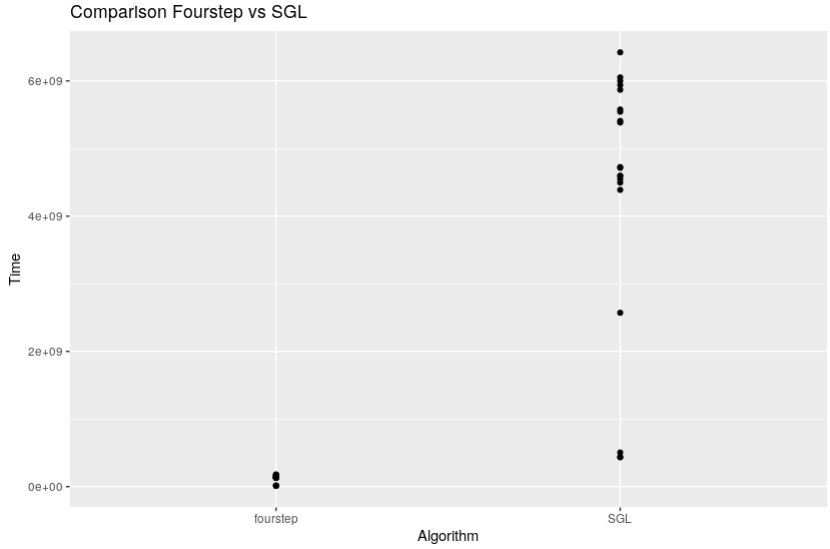

Figure 4: A graph comparing the run-times of the fourstep algorithm with the SGL package. The fourstep algorithm is substantially faster than SGL in performing sparse group lasso.

5 Conclusion

Putting this here in case I screw up [4]

Then I want to put this [5] here.

And finally I recall the work of [2] for the people.

References

[1] Stephen Boyd, Stephen P Boyd, and Lieven Vandenberghe. *Convex optimization*. Cambridge university press, 2004.

Algorithm 3 Three-step Algorithm

Input: $\lambda_0,\ \lambda_1,\ \ldots,\ \mathcal{E},\ \mathcal{A}$
Set $\mathcal{E}$ and $\mathcal{A}$ to $\emptyset$ ▷ Step 0
for $i = 1$ to n **do**
  $x_i \leftarrow d + Tx_{i-1|i-1}, \quad P_i \leftarrow Q + TP_{i-1|i-1}T^\top$ ▷ Predict current state
  $\widetilde{y}_i \leftarrow c + Zx_i, \quad F_i \leftarrow G + ZP_iZ^\top$ ▷ Predict current observation
  $v_i \leftarrow y_i - \widetilde{y}_i \quad K_i \leftarrow P_iZ^\top F^{-1}$ ▷ Forecast error and Kalman gain
  $x_{i|i} \leftarrow x_i + K_iv_i, \quad P_{i|i} \leftarrow P_i - P_iZ^\top K_i$ ▷ Update
  $\ell(\theta) = \ell(\theta) - v_i^\top F^{-1}v_i - \log(|F_i|)$
end for
return $\widetilde{Y} = \{\widetilde{y}_i\}_{i=1}^n,\ x = \{x_i\}_{i=1}^n,\ \widetilde{X} = \{x_{i|i}\}_{i=1}^n,\ P = \{P_i\}_{i=1}^n,\ \widetilde{P} = \{P_{i|i}\}_{i=1}^n,\ \ell(\theta)$

[2] Noah Simon, Jerome Friedman, Trevor Hastie, and Robert Tibshirani. A sparse-group lasso. *Journal of computational and graphical statistics*, 22(2):231–245, 2013.

[3] Robert Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society: Series B (Methodological)*, 58(1):267–288, 1996.

[4] Robert Tibshirani, Jacob Bien, Jerome Friedman, Trevor Hastie, Noah Simon, Jonathan Taylor, and Ryan J Tibshirani. Strong rules for discarding predictors in lasso-type problems. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 74(2):245–266, 2012.

[5] Yi Yang and Hui Zou. A fast unified algorithm for solving group-lasso penalize learning problems. *Statistics and Computing*, 25(6):1129–1141, 2015.

[6] Ming Yuan and Yi Lin. Model selection and estimation in regression with grouped variables. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 68(1):49–67, 2006.