Implementation of Sparse-Group Lasso Methods on Gene Expression Dataset

A useful R package

Shuting Liao, Shikun Wang, Ying Xie

The Publisher $\boxed{\mathcal{UM}}$

1. THEOREM OVERVIEW

Our data consist of an n response vector y, and an n by p matrix of features X. In many recent applications, we have $p >> n$: a case where standard linear regression fails. It finds a solution with few nonzero entries in β. In Group Lasso, we suppose our predictor variables were divided into m different groups. We focus on the "sparse-group lasso", we look for beta to minimize

$$\hat{M}(\beta) = \frac{1}{2t}||\beta - (\beta_0 - t\nabla(r_{(-k)}, \beta_0) + \frac{1}{2t}||\beta - \beta_0||_2^2 \tag{1}$$

where $\alpha \in [0, 1]$, a convex combination of the lasso and group-lasso penalties, and $r_{(-k)} = y - X^{(l)}\beta^{(l)}$ Combining the subgradient conditions with basic algebra, we get that $\beta = 0$ if

$$||S(\beta_0 - t\nabla l(r_{(-k)}, \beta_0, t\alpha\lambda)||_2 \leq t(1-\alpha)\lambda. \tag{2}$$

and otherwise β satisfies

$$(1 + \frac{t(1-\alpha)\lambda}{||\hat{\beta}||_2})\hat{\beta} = S(\beta_0 - t\nabla l(r_{(-k)}, \beta_0), t\alpha\lambda) \tag{3}$$

Taking the norm of both sides, we see that our generalized gradient step (i.e., the solution to Equation (10)) is

$$||\hat{\beta}||_2 = (||S(\beta_0 - t\nabla l(r_{(-k)}), \beta_0), t\alpha\lambda)||_2 - t(1-\alpha)\lambda)_+ \tag{4}$$

If we plug this into Equation (11), we see that our generalized gradient step (i.e., the solution to Equation (10)) is

$$\hat{\beta} = (1 - \frac{t(1-\alpha)\lambda}{||S(\beta_0 - t\nabla l(r_{(-k)}), \beta_0), t\alpha\lambda)||_2})_+ S(\beta_0 - t\nabla l(r_{(-k)}), \beta_0), t\alpha\lambda) \tag{5}$$

If we iterate Equation (12), and recenter each pass at $(\beta_0)_{new} = (\beta_0)_{old}$, then we will converge on the optimal solution for $\hat{\beta}^{(k)}$ given fixed values of the other coefficient vectors. If we apply this per block, and cyclically iterating through the blocks we will converge on the overall optimum. For ease of notation in the future, we let $U(\beta_0, t)$ denote our updated formula

$$U(\beta_0, t) = (1 - \frac{t(1-\alpha)\lambda}{||S(\beta_0 - t\nabla l(r_{(-k)}), \beta_0), t\alpha\lambda)||_2})_+ S(\beta_0 - t\nabla l(r_{(-k)}), \beta_0), t\alpha\lambda) \tag{6}$$

Note that in our case (linear regression)

$$\nabla l(r_{(-k)}, \beta_0) = -X^{(k)T} r_{(-k)}/n \tag{7}$$

2. ALGORITHM OVERVIEW

1. (Outerloop) Cyclically iterate through the groups; at each group(k) execute Step2.

2. Check if the group's coefficients are identically 0, by seeing if they obey

$$||S(X^{(k)T} r_{(-k)}, \alpha\lambda)||_2 \leq (1-\alpha)\lambda \tag{8}$$

3. If not, within the group apply Step 2.

4. (Inner loop) Until convergence iterate:

Start with $\beta^{(k,l)} = \theta^{(k,l)} = \beta_0^{(k)}$, step size $t = 1$, and counter $l = 1$. Repeat 0.
The following until convergence:

4.1 Update gradient g by $g = \nabla l(r_{(-k)}, \beta^{(k,l)})$;

4.2 Optimize step size by iterating $t = 0.8 * t$ until

$$l(U(\beta^{(k,l)}, t)) \leq l(\beta^{(k,l)}) + g^T \Delta_{(l,t)} + \frac{1}{2t}||\Delta_{(l,t)}||_2^2 \tag{9}$$

4.3 Update $\theta^{(k,l)}$ by

$$\theta^{(k,l+1)} \leftarrow U(\beta^{(k,l)}, t)$$

4.4 Update the center via a Nesterov step by

$$\beta^{(k,l+1)} \leftarrow \theta(k,l) + \frac{l}{l+3}(\theta(k,l+1) - \theta(k,l)) \tag{10}$$

4.5 Set $l = l + 1$; where Δ is the change between our old solution and new solution

$$\Delta_{(l,t)} = U(\beta^{(k,l)}, t) - \beta^{(k,l)}.$$

Algorithm 1 OneDim

Require: *Matrix* X, *Matrix* y
Ensure: β

procedure INITIALIZATION(X,y)
    Use *West algorithm* to calculate mean and variance,
    Set y'=y-mean to avoid calculating the intercept.
    Return scale as a vector.
end procedure

function BETTERPATHCALC(X, y)
    Calculate Lambda
    return *Lambda*
end function

function LINNEST(OuterLoop)
    while (**do** groupChange==1) ▷ to check if we need to keep computing beta
        groupChange=0;
    end while
    while (outer limitation: outermostCounter < outerIter[0] && outermostCheck < outerThresh[0]) **do**
        linSolver; ▷ set constraints to execute linSolver and update β
    end while
end function

function LINSOLVER(inner loop)
    for $i \leftarrow 0, np$ **do** ▷ for each group
        compute gradient grad;
        if ($zeroCheck <= pow(lambda2, 2) * lp$) **then** ▷ check if this group is zero group
            β=0
        else ▷ start inner loop to compute beta
            while ($innerlimitation : count <= innerIter \&\& check > thresh$) **do**
                set the constraints to update β by innerIter and innerThresh
                norm=$|z|$;
                compute gradient U;
                diff=-1;
                check the inequation :
                diff=RHS-LHS;
                while $diff < 0$ **do**
                    continue the loop
                    t=t*g; get new diff; ▷ optimize step size
                end while
                update θ by U;
                update beta by θ and U by Nesterov-style momentum
            end while
        end if
    end for
end function

variable	**comment**
eta	$r_{(-k)} = y - X^{(l)}\hat{\beta}^{(l)}$
grad	the Gradient
zeroCheck	to check if group is zero group
betaIsZero	indicator for beta is zero
rangeGroupInd[i]	the positon for the first element in group i
theta	to store the updated beta
U	updated formula to compute beta
t	step size
l	counter
np	number of groups
lp	length of each group
n	number of observation
groupChange	check if beta is ok, otherwise need to do linSolver
isActive	indicator if groupChange is 1;

Table 1: **Other Definitions**

Algorithm 2 Supplimental Functions

Require: *Matrix* X, *Matrix* y

Ensure: β

1: **function** LINGRADCALC(X, y)
2: compute $ldot = -r_{(-k)}/n$
3: **end function**
4: **function** LINNEGLOGLIKELIHOODCALC(X, y)
5: compute unpenalized loss function
6: **end function**

3. SIMULATION IN C++

We implemented the algorithm in paper in C++ with *Matrix615.h* and did simulations with randomly generated data. Our project is written by xcode on macbook pro. It can read data from file or use randomly generated data to calculate $\hat{\beta}$, and predict y. It also output R^2 and RMSE, with which we can test the accuracy of prediction.

Output of C++ Example

```
estimated beta=
-8.83592  -1.19675  4.07672  -8.51844  6.68252  1.2149  -6.05425  -2.07061  3.66666  -8.41446
-8.23652  -1.48738  -7.90107  3.58552  -4.16999  230803  -1.96936  -2.30416  -4.27065  -4.19624
3.42028  -1.13221  3.9326  3.78405  -8.59944  4.62309  3.34582  -2.03059  6.10709  1.84812
1.72567  1.14444  4.66506  1.3533  -4.11551  1.42855  2.72132  -4.51502  9.08728  -2.66784
7.66825  -3.81311  -3.65004  1.54093  2.49248  1.25922  4.44533  1.6477  -2.01475  -1.47798
2.69199  3.36841  1.3401  -9.23822  7.84053  -3.32986  2.00972  -4.95939  2.09015  -7.63303
-1.51501  -1.30075  1.73774  -1.01883  -1.35253  1.07117  -4.37588  -3.28805  1.25195  -4.11551
**********
Predicting value y...
fitted y:
-0.110665
0.435365
-1.454465
-0.768562
0.543765
R-square = 0.9473
rmse = 0.88622
total time: 0.000663s
```

In the SGL R-package from paper, the author did not calculate mean and variance in standardization step. We write a cpp edition which has much better efficiency.

Comparison of standardization code

```
library(microbenchmark)

> cppFunction('int stdize(NumericMatrix x){
+ double mean=x(0,0), var=0;
+ int count=2;
+ for(int i=0;i<x.ncol();i++){
+ for(int j=0;j<x.nrow();j++){
+ if(i! = 0||j! = 0){
+ var+=1.0*(count-1)count*(x(j,i)-mean)*(x(j,i)-mean);
+ mean=mean+(x(j,i)-mean)/count;
+ }
+ count++;
```

```
+ }
+ var=sqrt(var(x.nrow()*x.ncol()-1));
+ for(int k=0;k<x.nrow();k++)
+ x(k,i)=(x(k,i)-mean)/var;
+ }
+ return(x);
+ }')

> origin=function(x){ X < − x
+ means < − apply(X,2,mean)
+ X < − t(t(X) - means)
+ var < − apply(X,2,function(x)(sqrt(sum(x*x))))
+ X < − t(t(X) / var)
+ x < − X
+ X.transform < − list(X.scale = var, X.means = means)
+ }

> x = matrix(sample (100), 10)

> microbenchmark(stdize(x),origin(x))
Unit: microseconds
expr        min        lq          mean        median      uq       max       neval
stdize(x)   2.968      3.2160      5.23288     4.0525      5.537    24.139    100
origin(x)   101.172    107.1185    133.48869   111.7765    139.781  375.977   100
```

Short path			
method	1 group	2 groups	3 groups
n = 150, p = 1500, m = 10			
new	0.0341	0.08	0.0913
origin	1.02	1.031	1.052
n = 200, p = 2000, m = 200			
new	0.0543	0.0462	0.0498
origin	0.4307	0.4367	0.4032
n = 150, p = 10,000, m = 100			
new	0.5329	0.4625	0.5197
origin	1.277	1.171	1.263

Table 2: **Short path simulation**

Long path			
method	1 group	2 groups	3 groups
n = 150, p = 1500, m = 10			
new	0.2602	0.9926	1.227
origin	1.138	1.757	2.325
n = 200, p = 2000, m = 200			
new	0.0882	0.1384	0.2118
origin	1.188	1.154	1.534
n = 150, p = 10,000, m = 100			
new	1.188	1.188	1.188
origin	2.3	2.74	3.202

Table 3: **Long path simulation**

4. TIMING

The time complexity of our algorithm is $O(n^2p^2+n^2)$ where n is the number of rows and and p is the number of predictors.

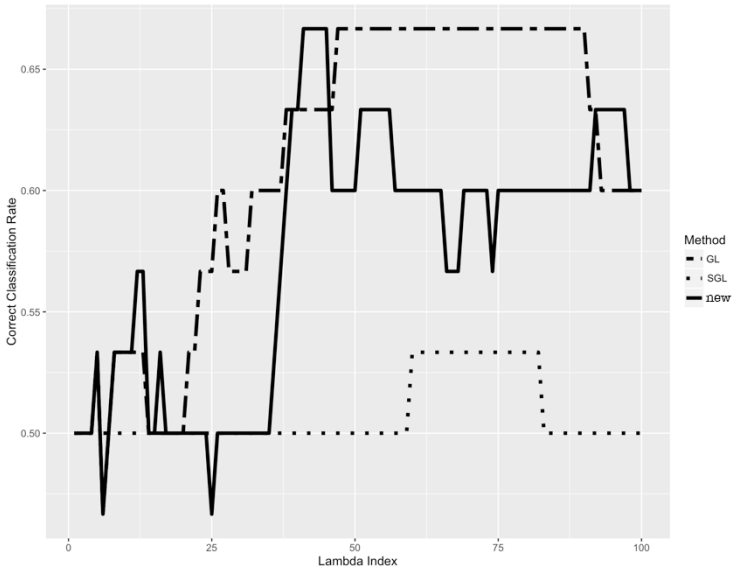
In the timing simulation step, we apply the data generated from the paper, but we compare our model and the original code instead. "New" is the timing of our model while "origin" is the original model in the paper. We can see clear improvement in our model From Table 2 and Table 3.

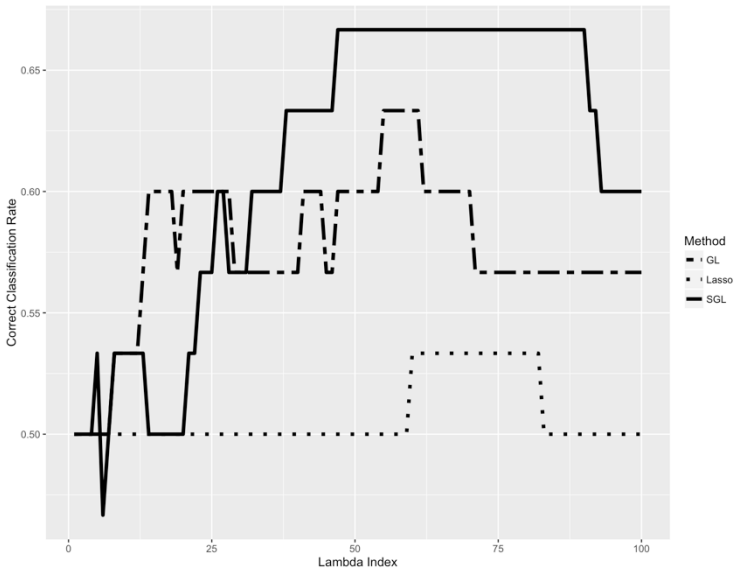
5. ACCURACY

The real dataset we used for comparisonwas the breast cancer data by Ma et al. (2004). This dataset contains gene expression values from 60 patients with estrogen positive breast cancer. The patients were treated with tamoxifen for 5 years and classified according to whether cancer recurred (there were 28 recurrences). Gene expression values were run on a GPL1223: Arcturus 22k human oligonucleotide microarray. Unfortunately, there was significant missing data. As a first pass, all genes with more than 50% missing data were removed. Other missing values were imputed by simple mean imputation. This left us with 12, 071 of our 22, 575 original genes. We again grouped genes together by cytogenetic position data, removing genes that were not recorded in the GSEA C1 dataset. Our final design matrix had 4989 genes in 270 pathways (an average of 18.5 genes per pathway). Thirty patients were chosen at random and used to build the three models. The remaining 30 were used to test their accuracies.

Referring to the second figure below, we see that in this example the sparse-group lasso outperforms the lasso and group lasso. The sparse-group lasso reaches 70% classification accuracy (though this is a narrow peak, so may be slightly biased high), while the group lasso peaks at 60% and the lasso comes in last at 53% accuracy. At its optimum the sparse-group lasso includes 54 genes from 11 bands, while the group lasso selects all 74 genes from 15 bands (again, largely smaller bands for the group lasso), and the lasso selects three genes all from separate bands. This example really highlights the advantage of the sparse-group lasso, it allows us to use group information, but does not force us to use entire groups.

The first figure compare the new method and the original model. We don't have a very stable result but on some interval, we can still have high enough accuracy. These two examples highlight two different possibilities for the sparse-group lasso. In the breast cancer data, the addition of group information is critical for classification, and the grouping may help give insight into the biological mechanisms. However, we should note that , the group whose "information" is largely just increases model variance the sparse-group lasso is certainly not perfect for every scenario with grouped data, but as evidenced in the cancer data it can sometimes be helpful.





REFERENCES

Simon N, Friedman J, Hastie T, et al. A sparse-group lasso[J]. Journal of Computational and Graphical Statistics, 2013, 22(2): 231-245.