PRIMAL: An Linear Programming-based Sparse Learning Library in R and Python

Qianli Shen[*], Zichong Li[*], Yujia Xie, Tuo Zhao [†]

Abstract

Linear Programming (LP) based sparse learning methods, such as the Dantzig selector (for linear regression)[2], sparse quantile regression [8], sparse support vector machines [3], have been widely used in machine learning for high dimensional data analysis [8, 9, 10]. Despite of their popularity, their software implementations in R are quite limited. We describe a new library in both R and Python – PaRametric sImplex Method for spArse Learning (PRIMAL), for the aforementioned LP-based sparse learning methods with the following two key features: 1) It provides a highly efficient optimization engine based on the parametric simplex method, which can efficiently solve large scale sparse learning problems; 2) Besides the estimation procedures, it provides additional functional modules such as data-dependent model selection and model visualization.

1 Introduction

TO DO: citation, affiliation, check grammar, typo

Many popular sparse learning methods involve solving an optimization problem. Here we are interested in those that can be casted into the following linear programs,

$$\max_x (c+\lambda\mathbf{1})^T x, \qquad \text{subject to } Ax \le b + \lambda\mathbf{1}, x \ge 0. \tag{1}$$

where A, b, and c are known variables, λ is a tuning parameter. There are several general methods for solving Linear Programs (LP). In particular, Interior point method [5] is proven to solve LPs in polynomial time, however, its total computational cost is cubically dependent on the scale of the problem, which is not scalable in high dimensions. Moreover, the computation at every iteration cannot take the advantage of the sparsity in the underlying models to boost the computation. The reason behind is that interior point method uses the log barrier to handle the constraints, thus, cannot yield sparse iterates. On the other hand, simplex method has stood the test of various practical problems by efficiently finding optimal solutions, though its worst-case iteration complexity has been shown to scale exponentially with the problem scale in existing literature.

[*]Equal contribution.

[†]!!!!!!Authors are affiliated with Georgia Institute of Technology. Emails: {xieyujia, mchen393, jianghm, tourzhao}@gatech.edu

These methods, though popular, are usually designed for solving (1) for one single tuning parameter. This is not satisfactory, since an appropriate choice of is usually unknown. Thus, one usually expects an algorithm to obtain multiple solutions tuned over a reasonable range of values for λ. For each value of λ, we need to solve a linear program from scratch, and it is therefore often very inefficient for high dimensional problems.

To overcome the above drawbacks, we adopt a variant of Simplex method — Parametric Simplex Method (PSM) [6] to efficiently solve LP-based sparse learning problems. PSM parametrizes (1) using the unknown regularization factor as a ”parameter”. This eventually yields a piecewise linear solution path for a sequence of regularization factors. PSM relies some special rules to iteratively choose the pair of variables to swap, which algebraically calculates the solution path during each pivoting. PSM terminates at a value of parameter, where we have successfully solved the full solution path to the original problem. Although in the worst-case scenario, PSM can take an exponential number of pivots to find an optimal solution path. Our empirical results suggest that the number of iterations is roughly linear in the number of nonzero variables for large regularization factors with sparse optima. This means that the desired sparse solutions can often be found using very few pivots.

2 Algorithm and Package Design

The PSM engine is the core optimization engine that aims to solve the LP problem with regularization parameters efficiently. Unlike the traditional generic LP solver, the PSM engine is specially designed for statistical methods that has a regularization parameter that requires tuning. Therefore, the PSM engine handles the tuning process efficiently. Furthermore, in order to achieve efficient and reliable matrices and vectors computation, we will exploit Eigen library, an open-source C++ library for linear algebra.

More specifically, the PSM algorithm is briefly described as follows:

- Initialize $\lambda = (|\min_j c_j|, |\min_j b_j|)$, which is large enough such that $x = 0$ is an optimal solution to (1). Let $\lambda^* = \lambda$, $\lambda_{\min}$ denote the minimal regularization parameter of our interest.
- While $\lambda^* > \lambda_{\min}$ do
  - Decrease λ until x violates the constraints, i.e., x is infeasible. Denote by λ^* the threshold that breaks the feasibility of x.
  - Update x by swapping a nonzero variable with a zero variable as in the simplex method. More precisely, select the variable that attains λ^* as the entering variable, which becomes nonzero afterwards; select the variable that first violates the constraint as the leaving variable, which becomes zero.

As can be seen, we are essentially keeping the iterates to be the optimal solution corresponding to different values of . As a result, PSM naturally takes the advantage of the regularization parameter tuning and obtains the optimal solution path in one run. Note that the subproblems involved in each iteration of the above algorithm – (2) for determining the decreased value of and

(3) for determining the update for x– both admit closed-form intermediate solutions. Also, the update in 3) only involves part of the variables and yields sparse iterates, and the computational cost can be significantly reduced by using sparse matrix vector multiplication. Therefore, the algorithm has a low per-iteration cost. Furthermore, although in the worst-case scenario, PSM can take an exponential number of operations to find an optimal solution path, empirical results suggest that the number of iterations is roughly linear in the number of nonzero variables for large regularization factors with sparse optima. This implies that the desired sparse solution can be found in very few iterations – roughly speaking O(s), where s is the number of nonzero entries in the optimal solution. For sparse learning problems, s is usually much smaller than the number of variables. As a result, the algorithm is highly efficient.

We have four basic modules in our package design: the core optimization engine using parametric simplex method, parameter tuning engine, visualization engine, and application engine for sparse learning such as Dantzig selector and differential graph estimation.

PSM engine. The PSM described in the previous section is implemented in a pure C/C++ fashion. The code is supported by Eigen library, which is an efficient headers only C++ linear algebra library.

User Interface. The user interface for Dantzig selector, sparse support vector machine, and differential graph estimation is implemented. We also include the sample code for the applications in the package.

Visualization engine. Visualization functions for visualizing solution path is implemented.

3 Examples

Now we illustrate the user interface using Danzig selector as an example.

3.1 R user Interface

The following example solves

$$\min_{\beta} \|\beta\|_1 \quad \text{subject to } \|X^T(y - X\beta)\|_\infty \le \lambda \tag{2}$$

for any λ.

```
>  library(PRIMAL)
>  # Generate a synthetic dataset: the design matrix and coefficient vector
> n = 100 # number of samples
> d = 250 # sample dimension
> c = 0.5 # correlation parameter
> s = 20  # support size of coefficient
> set.seed(1024)
> X = scale(matrix(rnorm(n*d),n,d)+c*rnorm(n))/sqrt(n-1)*sqrt(n)
> flag = runif(s,-1,1)
> beta1 <- c()
> for(i in 1:s){
+ if(flag[i]>=0) beta1[i]=rnorm(1,1,1)
+ if(flag[i]<0)  beta1[i]=rnorm(1,-1,1)
+ }
> beta = c(beta1, rep(0, d-s))
> # Generate response using Gaussian noise
> noise = rnorm(n)
> Y = X%*%beta + noise
>
> # Dantzig selection solved with parametric simplex method
fit.dantzig = Dantzig_solver(X, Y, max_it = 100, lambda_threshold = 0.01)
```

This line will return a list containing the information of the solution. From `fit.dantzig`, we can recover solutions corresponds to any regularization parameter λ.

```
> # Now let's see the regularization parameters along the path
> print(fit.dantzig$lambda)
  [1] 195.22461 176.80544 135.96638 116.72507 104.60908 102.70803
  [7]  89.74968  82.94148  78.33078  75.14407  74.65001  64.26936
 [13]  58.49683  53.85248  49.89716  49.71839  47.68773  42.22191
 [19]  40.88262  40.79184  38.71504  38.71244  38.07740  37.72982
 [25]  37.72246  37.51079  37.47705  37.31156  37.09234  36.65785
 [31]  36.51983  35.61435  34.75857  34.47443  34.22125  34.16291
 [37]  34.08827  33.72415  33.62017  33.40794  33.33360  33.10474
 [43]  32.65156  32.26433  29.46930  28.98127  28.01981  27.95761
 [49]  27.27683  27.01244  26.16928  26.15484  26.14264  26.05776
 [55]  25.90875  25.58693  24.42879  24.39550  24.33576  24.16818
 [61]  24.06451  23.96691  23.70723  23.40995  23.27554  22.48808
 [67]  22.08780  21.24536  20.62978  20.43703  20.43107  20.40643
 [73]  19.56953  19.39416  19.29465  18.91443  18.81399  18.41681
 [79]  18.37950  18.30446  18.04614  17.93162  17.79798  17.70347
 [85]  17.68071  17.66381  16.97636  16.67124  16.55067  16.41163
 [91]  15.36333  15.25158  15.10068  15.06699  15.05869  14.92861
 [97]  14.82495  14.39728  14.32287  14.05490
```

For λ larger than the largest value listed above, i.e., 195.22, the optimal solution of (2) is $\beta = \mathbf{0}$. For λ smaller than 195.22 larger than `lambda_threshold`, the optimal solution is the same as the smallest value listed above that is larger than λ. For example, consider $\lambda = 24$, its solution is the same as $\lambda = 24.06$ in entry 61. We can get the corresponding β by calling `fit.dantzig$beta[,61]`, and the corresponding value of objective function by calling `fit.dantzig$value[61]`.

Moreover, we can visualize the regularization path by calling `plot.primal(fit.dantzig)`. The plots are shown in Figure 1.

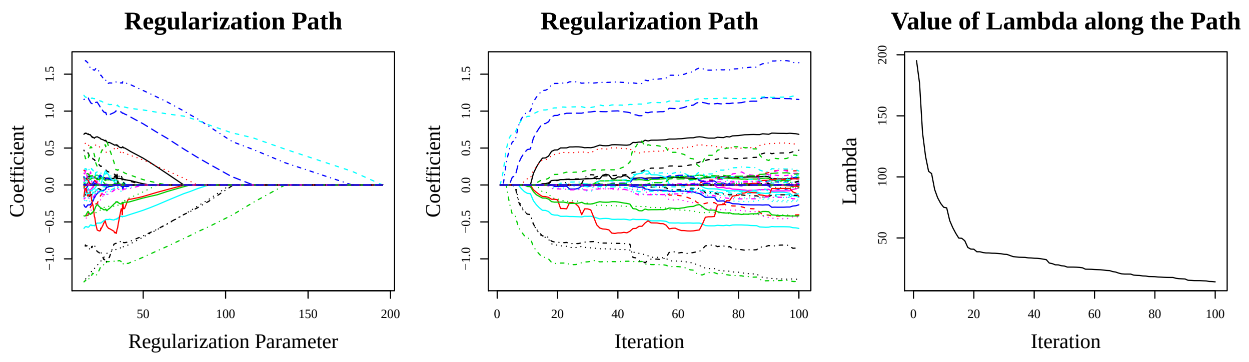

Figure 1: Visualizations of the solution path.

3.2 Python User Interface

Now we use the same example to illustrate the Python user interface.

```
> # load packages
> import numpy as np
> from sklearn.preprocessing import scale
> import pypsm
> from pypsm import Dantzig
> # Generate the design matrix and regression coefficient vector
> n = 100 # sample number
> d = 80 # sample dimension
> c = 0.5 # correlation parameter
> s = 20  # support size of coefficient
> X = scale(np.random.randn(n,d)+c* np.tile(np.random.randn(n),[d,1]).T )/ (n*(n-1))**0.5
> beta = np.append(np.random.rand(s), np.zeros(d-s))
> # Generate response using Gaussian noise, and fit sparse linear models
> noise = np.random.randn(n)
> y = np.matmul(X,beta) + noise
> # fit the model
> solver = Dantzig(X, y)
> solver.train()
result = solver.coef()
solver.plot()
```

The result can be similarly obtained by calling `solver.coef()['lambda_list']`, `solver.coef()['theta_list']` and `solver.coef()['target_list']`

4 Contact

If you have further questions regarding the package, please contact

Xie.Yujia000@gmail.com

References