Network Inference via Elastic Net Penalized Structural Equation Models

Anhui Huang

June, 2023

Contents

Introduction

This vignette demonstrate the usage of `sparseSEM` package via a simulated network as well as a Gene Regulatory Network (GRN) in budding yeast (Saccharomyces cerevisiae) [1].

simulated network

The simulated network is relatively small and sparse for demonstration purpose. A total of 30 nodes with 30 connected edges were simulated following the description in Huang 2014 [4]:

- Number of nodes in the network: $N_g = 30$;
- Number of expected edges $N_e = 1$;
- Edge weight B_{ij} was generated randomly from uniform distribution in $(0.5, 1)$ or $(-1, -0.5)$ if there was an edge from node j to node i; and
- Noise E_{ij} was sampled from a Gaussian distribution with zero mean and variance 0.01

Yeast Gene Regulatory Network (GRN)

The data contains expression levels of 6,126 yeast ORFs and 2,956 genetic markers from 112 yeast segregants from the cross of a laboratory strain (BY4716) and a wild strain (RM11-1a) [2].

The following steps were followed to arrived at the final dataset in `data(yeast)`:

- Step 1. Screen out ORFs not in the Yeast Comparative Genomics database [3], resulting 5225 names of ORFs in annotation list;
- Step 2. Screen out ORFs with more than 5% of missing expression data, resulted in 3,380 ORFs;
- Step 3. Associated gene markers with an ORF if they were in distance $\leq$ 20 kb representing the QTL resolution for this cross [1].
- Step 4. Mapping of gene expression levels with their associated gene markers through Wilcoxon rank-sum test with adjusted `q-value` following the procedure described in [1].
- Step 5. Keep 1 gene marker with the smallest p-value if multiple cis-eQTLs were found for an ORF, leads to 1,162 ORFs having cis-eQTLs.

The gene expression profile of 3,380 ORFs and genetic marker of 1,162 eQTLs were then used for network inference by `sparseSEM`.

Back to Top

Quick Start

We will give users a general idea of the package by using the simulated example to demonstrates the basis package usage. Through running the main functions and examining the outputs, users may have a better idea on how the package works, what functions are available, which parameters to choose, as well as where to seek help. More details on the GRN are given in later sections.

Let us first clear up the workspace and load the `sparseSEM` package:

```
rm(list = ls())
library(sparseSEM)
```

The simulated network includes matrices of `X`, `Y` , `B`, and `Missing` with 30 rows and 200 columns giving the following measurements:

- `Y`: The observed node response data with dimension of M (nodes) by N (samples);
- `X`: The network node attribute (for causal effect inference) with dimension of M (nodes) by N (samples);
- `B`: A sparse M by M matrix which contains the actual edges among the network of M nodes.
- `Missing`: M by N matrix corresponding to elements of Y to denote any missing values.

```
data(B);
data(Y);
data(X);
data(Missing);
cat("dimenstion of Y: ",dim(Y) )
```

```
## dimenstion of Y:  30 200
```

```
cat("dimenstion of X: ",dim(X) )
```

```
## dimenstion of X:  30 200
```

Note that both `B` and `Missing` are optional to run all functions in the package.

We fit the model using the most basic call to `elasticNetSEM`:

```
set.seed(1)
output = elasticNetSEM(Y, X, Missing, B, verbose = 1);
```

```
##  elastic net SML; 30 Nodes,  200 samples; verbose:  1
##
##  0/4 lambdas.    lambda_factor: 0.630957 lambda: 91.453813
##  1/4 lambdas.    lambda_factor: 0.398107 lambda: 57.703455
##  2/4 lambdas.    lambda_factor: 0.251189 lambda: 36.408419
##  3/4 lambdas.    lambda_factor: 0.158489 lambda: 22.972159
##   computation time: 2.284 sec
```

```
names(output)
```

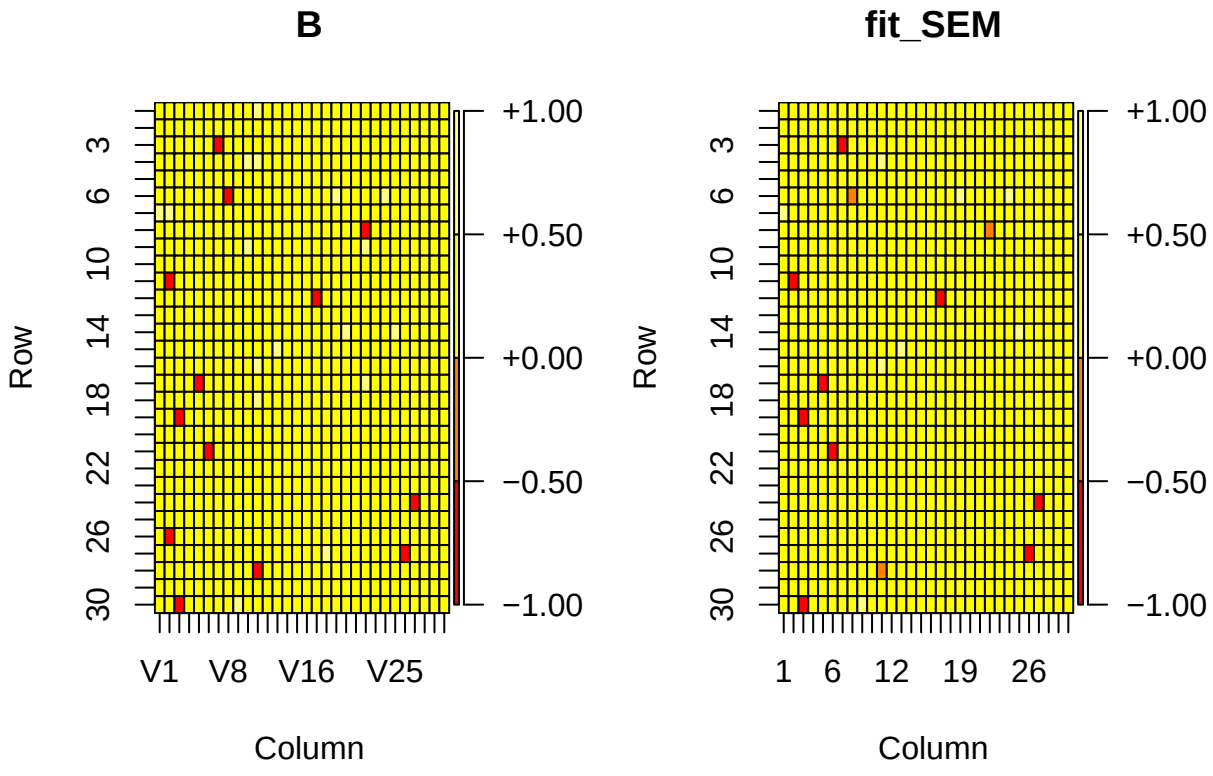
```
## [1] "weight"          "F"               "statistics"      "hyperparameters"
## [5] "runTime"         "call"
```

"output" is a list containing all the relevant information of the fitted model. Users can examine the output by directly looking at each element in the list. Particularly, the sparse regression coefficients can be extracted as showed below:

```
fit_SEM = output$weight
```

Here we can use package `plot.matrix` to visually examine the `sparseSEM` estimated network structure vs. the simulated ground truth:

```
library('plot.matrix')
par(mfrow = c(1, 2),mar=c(5.1, 4.1, 4.1, 4.1))
plot(B)
plot(fit_SEM)
```



The left plot is the simulation ground truth, and the right plot is the `sparseSEM` inferred network. The plot shows that 29 out of the 30 edges in the network are captured by `sparseSEM` (misses 1 edge connect node_2 (column 2) to node_26 (row 26)). Meanwhile, the estimated edge weights are relatively smaller in the fitted model due to the `elastic net` penalty.

Cross Validation (CV)

The hyperparameters in `elasitc net` are specified as (alpha, lambda) which control the number of non-zero effects to be selected, and cross-validation (CV) is perhaps the simplest and most widely used method in deciding their values. while `elasticNetSEM` performs all the computation behind the scene, `elasticNetSEMcv` is the function to allow users to control more details of CV, which can be called using the following code.

```
set.seed(1)
cvfit = elasticNetSEMcv(Y, X, Missing, B, alpha_factors = c(0.75, 0.5, 0.25),
                        lambda_factors=c(0.1, 0.01, 0.001), kFold = 5, verbose  = 1);
```

```
##  elastic net SML; 30 Nodes,  200 samples; verbose:  1
##
##  Adaptive_EN 5-fold CV, alpha: 0.750000.
##  0/2 lambdas.    lambda_factor: 0.100000 lambda: 18.359640
##  1/2 lambdas.    lambda_factor: 0.010000 lambda: 1.835964
##   computation time: 0.902 sec
```

```
names(cvfit)
```

```
## [1] "cv"  "fit"
```

`elasticNetSEMcv` returns a `cvfit` object, which is a list with all the ingredients of CV and the final fit results using CV selected hyperparameters. We can view the CV results, selected hyperparameters and the corresponding coefficients. For example, result using different hyperparameters and the corresponding prediction errors are shown below:

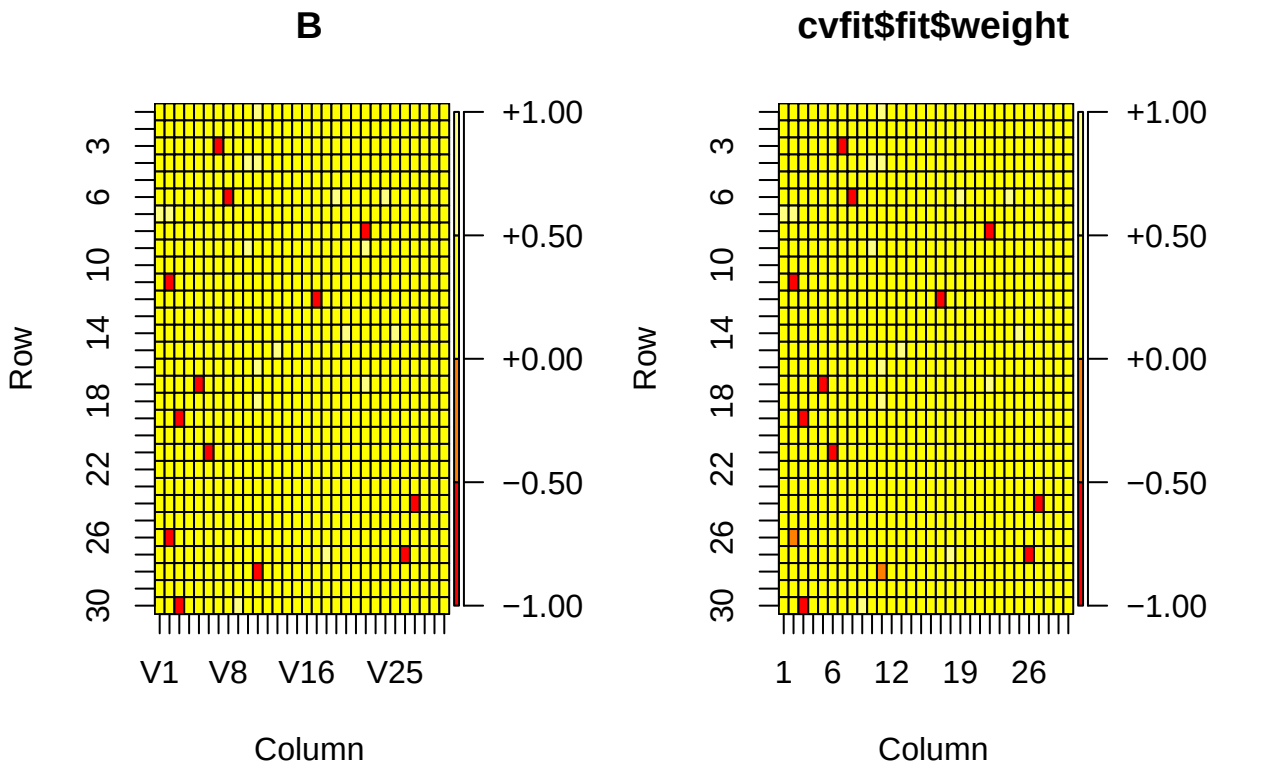
```
head(cvfit$cv)
```

```
##      alpha lambda mean Error         Ste
## [1,]  0.75  0.100 0.01027956 0.006643670
## [2,]  0.75  0.010 0.01027958 0.006643348
## [3,]  0.75  0.001 0.01027954 0.006643037
## [4,]  0.50  0.100 0.01027957 0.006643796
## [5,]  0.50  0.010 0.01027958 0.006643438
## [6,]  0.50  0.001 0.01027951 0.006643027
```

Note that the values for hyperparameters (alpha, lambda) are `alpha_factor` and `lambda_factor`, which are the weights of the actual (alpha, lambda) and should be in range of [0,1]. I.e., $\alpha = alpha\_factor * \alpha_0$ and $\lambda = lambda\_factor * \lambda_0$. Here $\alpha_0 = 1$ as in `elastic net`, $0 \leq \alpha \leq 1$. However, λ_0 is algorithm computed value such that there is one and only one non-zero element in matrix $\hat{\mathbf{B}}$. See Huang (2014) [4] for details on the `lasso` discarding rule for SEM.

As shown in the plot below, due to the granular choice of `alpha_factor` and `lambda_factor`, the performance result shows that all 30 edges are captured:

```
par(mfrow = c(1, 2),mar=c(5.1, 4.1, 4.1, 4.1))
plot(B)
plot(cvfit$fit$weight)
```



With ground truth `B` provided, the functions also compute the performance statistics in 'fit*statistics*, which is 6x1 array keeping record of:

1. correct positive
2. total positive
3. false positive
4. positive detected
5. Power of detection (PD) = correct positive/total positive
6. False Discovery Rate (FDR) = false positive/positive detected

```
cvfit$fit$statistics
```

```
##                     [,1]
## correct_positive      30
## total_ground truth    30
## false_positive         0
## true_positive         30
## Power                  1
## FDR                    0
```

Stability Selection (STS)

With a twist of the hyperparameters, the above plots showed that the non-zero edges can be slightly different. Stability Selection [5,6] is an algorithm to achieve stable effect selection with controlled False Discovery Rate

(FDR). The algorithm requires bootstrapping (typically n = 100 rounds), each round randomly selected half of the dataset to run through the lasso/elastic net selection path. Then the stable effects are those consistently selected in different bootstraps measured by the bounded pre-comparison error rate and FDR.

The following code shows the example of running STS:

```
tStart = proc.time()
set.seed(0)
output = enSEM_stability_selection(Y,X, Missing,B,
                                   alpha_factors = seq(1,0.1, -0.1),
                                   lambda_factors =10^seq(-0.2,-3,-0.2),
                                   kFold = 5,
                                   nBootstrap = 100,
                                   verbose = -1)
tEnd = proc.time()
simTime = tEnd - tStart;
print(simTime)
```

```
##    user  system elapsed
##  27.106   0.900  26.768
```

```
names(output)
```

```
## [1] "STS"        "statistics" "STS data"   "call"
```

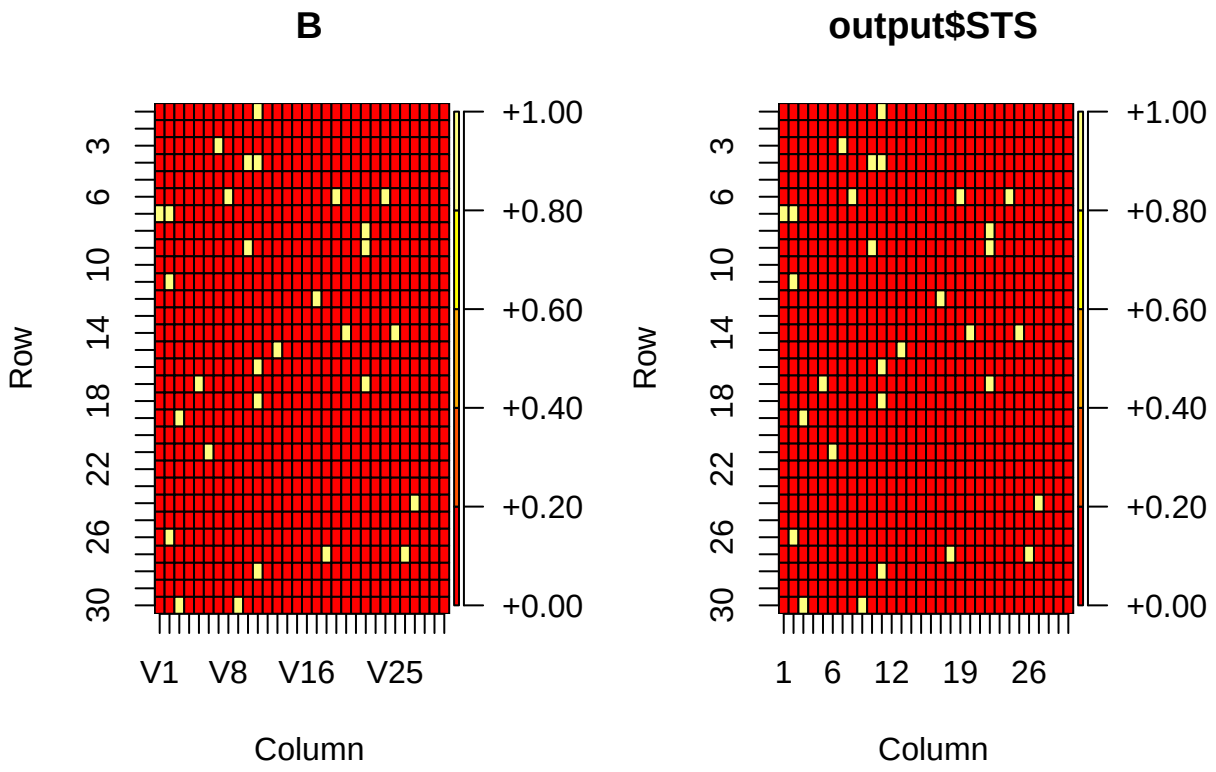
```
cat("nSTS = ", length(which(output$STS !=0)))
```

```
## nSTS =  30
```

One key part of STS method is that, the hyperparameters generally are set toward higher penalty to allow few non-zero edges. This is due to the fact that, if most of the hyperparameters are with small penalty, the number of non-zero edges will be high such that no matter how the threhold (pi) [5, 6] is set, there will be high FDR.

The above example shows that we set the values of lambda_factors to not too small, and all true effects are captured with no false positives as shown below:

```
B[which(B!=0)] =1
par(mfrow = c(1, 2),mar=c(5.1, 4.1, 4.1, 4.1))
plot(B)
plot(output$STS)
```



Parallel bootstrapping is also implemented with R package `parallel`:

```
library(parallel)
cl<-makeCluster(4,type="SOCK")
clusterEvalQ(cl,{library(sparseSEM)})
output = enSEM_stability_selection_parallel(Y,X, Missing,B,
                                            alpha_factors = seq(1,0.1, -0.1),
                                            lambda_factors =10^seq(-0.2,-3,-0.2),
                                            kFold = 3,
                                            nBootstrap = 100,
                                            verbose = -1,
                                            clusters = cl)
stopCluster(cl)
```

Back to Top

Yeast GRN

We next turn to the real world scenario to apply `sparseSEM` to the yeast GRN dataset analysis. While the above simulated network has 30 nodes, the yeast GRN has 3380 nodes, which increases computation time significantly. Roughly the computation time for the yeast GRN is about 30 min with `elasticNetSEM`, and 1.5 hour with `enSEM_stability_selection` running on an Intel i5 CPU.

```
rm(list = ls())
library(sparseSEM)
data(yeast)
output = elasticNetSEM(Y, X, verbose = 1)
# STS
STS = enSEM_stability_selection(Y,X,verbose = -1)
```

For even larger network, a parallel computation version of `sparseSEM` is also available at sparseSEM GitHub Repo.

Back to Top

References

1. Brem RB, Kruglyak L: The landscape of genetic complexity across 5,700 gene expression traits in yeast. Proceedings of the National Academy of Sciences of the United States of America 2005, 102:1572-1577

2. Brem RB, Yvert G, Clinton R, Kruglyak L: Genetic dissection of transcriptional regulation in budding yeast. Science 2002 296:752-755

3. Kellis M, Patterson N, Endrizzi M, Birren B, Lander ES: Sequencing and comparison of yeast species to identify genes and regulatory elements. Nature 2003, **423** : 241 − 254

4. Huang, A. (2014). "Sparse model learning for inferring genotype and phenotype associations." Ph.D Dissertation Chapter 7. University of Miami(1186).

5. Meinshausen, N. and Buhlmann, P., 2010. Stability selection. Journal of the Royal Statistical Society: Series B (Statistical Methodology), 72(4), pp.417-473.

6. Shah, R.D. and Samworth, R.J., 2013. Variable selection with error control: another look at stability selection. Journal of the Royal Statistical Society: Series B (Statistical Methodology), 75(1), pp.55-80.