

Learning the topology of graphs

Convex Group-HKUST
2018-08-18

Contents

1	Installation	1
2	Problem Statement	1
3	Usage of the package	1
4	Explanation of the algorithms	5
4.1	learnGraphTopology: Learning the topology of graph	6
	References	6

1 Installation

For installation instructions, please visit <https://github.com/dppalomar/spectralGraphTopology>

2 Problem Statement

The Laplacian matrix Θ of a graph contains the information of its topology and weight connections. By definition a Laplacian matrix is positive semi-definite, symmetric, and with sum of rows equal to zero. The Laplacian linear operator $\mathcal{L}\mathbf{w}$ maps a vector of weights \mathbf{w} into a valid Laplacian matrix so that the conditions are satisfied by construction.

The underlying optimization problem of learning a K-component graph may be expressed as follows:

$$\begin{aligned} \underset{\mathbf{w}, \mathbf{\Lambda}, \mathbf{U}}{\text{minimize}} \quad & -\log \det(\mathbf{\Lambda}) + \text{tr}(\mathbf{K}\mathcal{L}\mathbf{w}) + \frac{\beta}{2} \|\mathcal{L}\mathbf{w} - \mathbf{U}\mathbf{\Lambda}\mathbf{U}^T\|_F^2 \\ \text{subject to} \quad & \mathbf{w} \geq 0, \mathbf{\Lambda} \in \mathcal{S}_{\mathbf{\Lambda}}, \text{ and } \mathbf{U}^T\mathbf{U} = \mathbf{I} \end{aligned}$$

where $\mathcal{S}_{\mathbf{\Lambda}}$ further constrains the eigenvalues of Laplacian matrices $\Theta = \mathcal{L}\mathbf{w}$ according to some topology (e.g., a K -component graph has K zero eigenvalues).

In order to solve this problem, we use a block coordinate descent algorithm to iteratively optimize each variable while holding the others fixed. Please, for details check the paper or the algorithm description at the end of this document.

3 Usage of the package

We illustrate the usage of the package with simulated data, as follows:

```
library(spectralGraphTopology)
set.seed(123)

# Number of samples
T <- 200

# Vector to generate the Laplacian matrix of the graph
w <- runif(10)
```

```

# Laplacian matrix
Theta <- L(w)
# Sample data from a Multivariate Gaussian
N <- ncol(Theta)
Y <- MASS::mvrnorm(T, rep(0, N), MASS::ginv(Theta))
# Number of components of the graph
K <- 1
# Learn the Laplacian matrix
res <- learnGraphTopology(Y, K, beta = 10)

```

Let's visually inspect the true Laplacian and the estimated one:

```

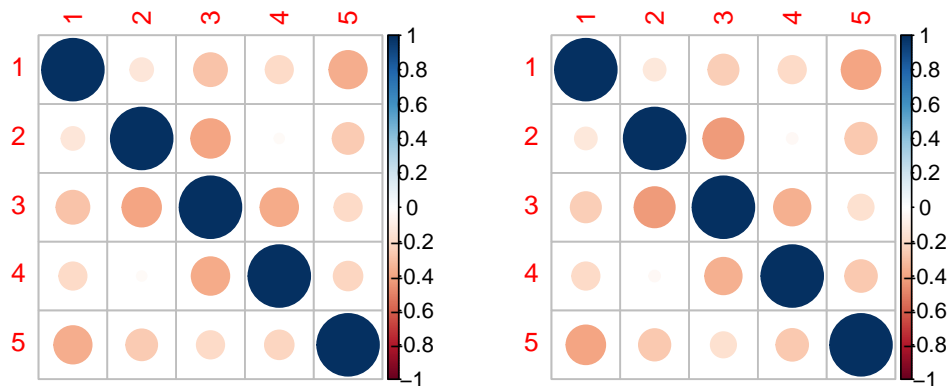
Theta
#>           [,1]      [,2]      [,3]      [,4]      [,5]
#> [1,]  2.3678770 -0.2875775 -0.7883051 -0.4089769 -0.8830174
#> [2,] -0.2875775  1.8017068 -0.9404673 -0.0455565 -0.5281055
#> [3,] -0.7883051 -0.9404673  3.1726265 -0.8924190 -0.5514350
#> [4,] -0.4089769 -0.0455565 -0.8924190  1.8035672 -0.4566147
#> [5,] -0.8830174 -0.5281055 -0.5514350 -0.4566147  2.4191726
res$Theta
#>           [,1]      [,2]      [,3]      [,4]      [,5]
#> [1,]  2.2953931 -0.27079062 -0.6305267 -0.41598450 -0.9780912
#> [2,] -0.2707906  1.95134562 -1.0254774 -0.06127561 -0.5938020
#> [3,] -0.6305267 -1.02547743  2.9706229 -0.83893311 -0.4756856
#> [4,] -0.4159845 -0.06127561 -0.8389331  1.91751385 -0.6013206
#> [5,] -0.9780912 -0.59380196 -0.4756856 -0.60132063  2.6488995

```

```

library(corrplot)
par(mfrow = c(1, 2))
corrplot(cov2cor(Theta))
corrplot(cov2cor(res$Theta))

```



We evaluate the performance of the learning process in a more objective manner by computing two criterions:

1. relative error
2. percentage improvement in average loss (PRIAL)

```

relativeError <- function(Xtrue, Xest) {
  return (100 * norm(Xtrue - Xest, type = "F") / norm(Xtrue, type = "F"))
}

```

```

}

prial <- function(Xtrue, Xest) {
  Xnaive <- MASS::ginv(cov(Xtrue))
  return (100 * (1 - (norm(Xest - Xtrue, type = "F") /
                        norm(Xnaive - Xtrue, type = "F"))^2))
}

```

```

Theta_naive <- MASS::ginv(cov(Y))
rel_err <- c(proposed = relativeError(Theta, res$Theta),
             naive = relativeError(Theta, Theta_naive))
prial_avg <- c(proposed = prial(Theta, res$Theta),
              naive = prial(Theta, Theta_naive))

rel_err
#> proposed      naive
#> 8.869914 9.224060
prial_avg
#> proposed      naive
#> 98.91276 98.82421

```

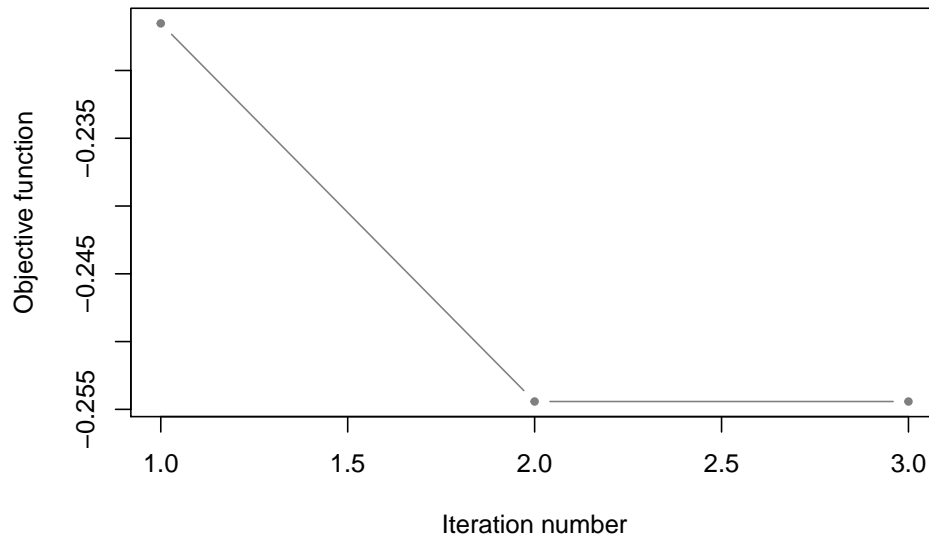
In this case, the naive estimation of the Laplacian matrix (i.e., generalized inverse of the sample covariance matrix) performs already quite well since the ratio T/N is large enough 40 for the sample covariance matrix to be accurately estimated.

Let's also look at the convergence of the objective function versus iterations:

```

N_iter <- length(res$fun)
plot(c(1:N_iter), res$fun, type = "b", pch=19, cex=.6, col = scales::alpha("black", .5),
     xlab = "Iteration number", ylab = "Objective function")

```



For $K > 1$, we can generate the Laplacian as a block diagonal matrix, as follows

```

library(spectralGraphTopology)
T <- 50
w1 <- runif(3)
w2 <- runif(3)

```

```

Theta1 <- L(w1)
Theta2 <- L(w2)
N1 <- ncol(Theta1)
N2 <- ncol(Theta2)
Theta <- rbind(cbind(Theta1, matrix(0, N1, N2)),
               cbind(matrix(0, N2, N1), Theta2))
Y <- MASS::mvrnorm(T, rep(0, N1 + N2), MASS::ginv(Theta))
K <- 2
res <- learnGraphTopology(Y, K, beta = 10)

```

```

Theta_naive <- MASS::ginv(cov(Y))
rel_err <- c(proposed = relativeError(Theta, res$Theta),
             naive = relativeError(Theta, Theta_naive))
prial_avg <- c(proposed = prial(Theta, res$Theta),
               naive = prial(Theta, Theta_naive))

rel_err
#> proposed      naive
#> 8.992709 14.027762
prial_avg
#> proposed      naive
#> 99.39504 98.52794

```

```

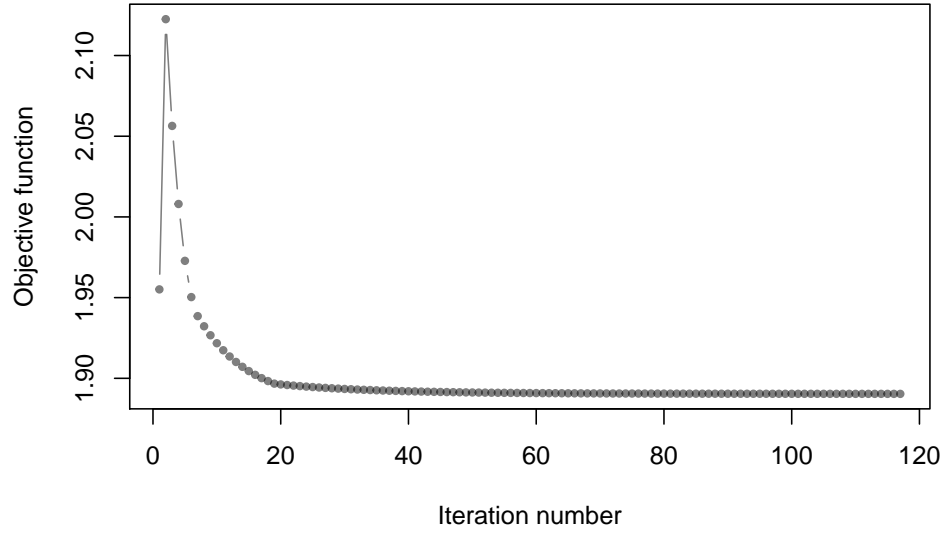
Theta
#>      [,1]      [,2]      [,3]      [,4]      [,5]      [,6]
#> [1,] 1.1183094 -0.8509632 -0.2673462 0.0000000 0.0000000 0.0000000
#> [2,] -0.8509632 1.4495635 -0.5986003 0.0000000 0.0000000 0.0000000
#> [3,] -0.2673462 -0.5986003 0.8659465 0.0000000 0.0000000 0.0000000
#> [4,] 0.0000000 0.0000000 0.0000000 1.6007581 -0.6085997 -0.9921584
#> [5,] 0.0000000 0.0000000 0.0000000 -0.6085997 0.7997897 -0.1911900
#> [6,] 0.0000000 0.0000000 0.0000000 -0.9921584 -0.1911900 1.1833484
res$Theta
#>      [,1]      [,2]      [,3]      [,4]      [,5]      [,6]
#> [1,] 1.1839191 -0.8535033 -0.33041578 0.0000000 0.0000000 0.0000000
#> [2,] -0.8535033 1.3384861 -0.48498280 0.0000000 0.0000000 0.0000000
#> [3,] -0.3304158 -0.4849828 0.86057122 0.0000000 0.0000000 -0.04517265
#> [4,] 0.0000000 0.0000000 0.0000000 1.6398148 -0.67676874 -0.96304605
#> [5,] 0.0000000 0.0000000 0.0000000 -0.6767687 0.74520251 -0.06843377
#> [6,] 0.0000000 0.0000000 -0.04517265 -0.9630461 -0.06843377 1.07665247

```

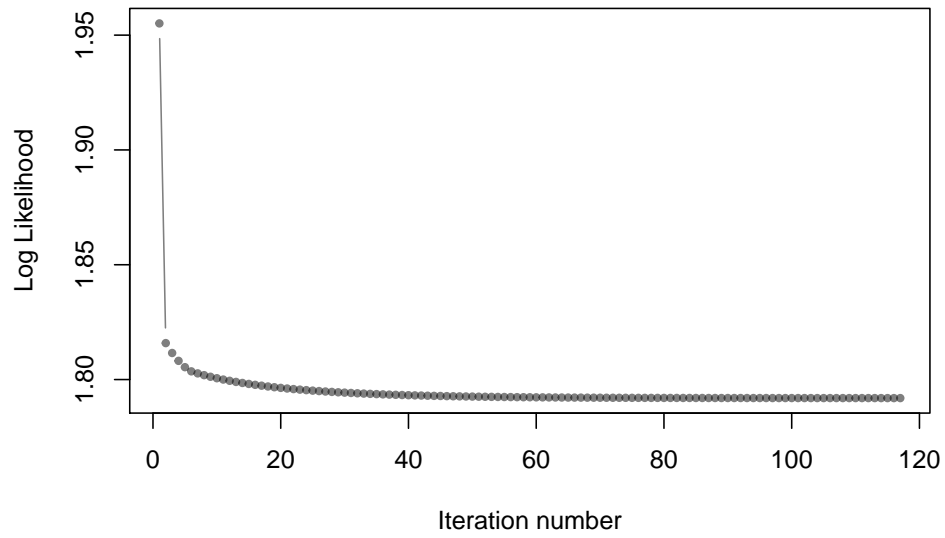
```

N_iter <- length(res$fun)
plot(c(1:N_iter), res$fun, type = "b", pch=19, cex=.6, col = scales::alpha("black", .5),
     xlab = "Iteration number", ylab = "Objective function")

```



```
plot(c(1:N_iter), res$loglike, type = "b", pch=19, cex=.6, col = scales::alpha("black", .5),
     xlab = "Iteration number", ylab = "Log Likelihood")
```



4 Explanation of the algorithms

In this section we describe in detail the algorithms designed to solve the graph topology learning problem.

4.1 learnGraphTopology: Learning the topology of graph

The goal of `learnGraphTopology` is to estimate the Laplacian matrix generated by the weight vector of a graph, \mathbf{w} . The algorithm for the function `learnGraphTopology` is stated as follows:

Data: \mathbf{Y} (data matrix), K ($\#\{\text{components}\}$), β (regularization term), $\mathbf{w}^{(0)}$, $\boldsymbol{\lambda}^{(0)}$, $\mathbf{U}^{(0)}$ (initial parameter estimates), α_1 , α_2 (lower and upper bound on the eigenvalues of the Laplacian matrix), ρ (how much to increase beta per iteration).

Result: Θ (Laplacian matrix)

$N \leftarrow \text{ncol}(\mathbf{Y})$

while *objective function do not converged* **or** *max $\#\{\text{iterations}\}$ not reached* **do**

$k \leftarrow 0$

while *parameters do not converged* **or** *max $\#\{\text{iterations}\}$ not reached* **do**

$\mathbf{w}^{(k+1)} \leftarrow \text{w_update}(\mathbf{w}^{(k)}, \mathbf{U}^{(k)}, \boldsymbol{\lambda}^{(k)}, \beta, N, \mathbf{K})$

$\mathbf{U}^{(k+1)} \leftarrow \text{U_update}(\mathbf{w}^{(k+1)}, N)$

$\boldsymbol{\lambda}^{(k+1)} \leftarrow \text{lambda_update}(\mathbf{w}^{(k+1)}, \mathbf{U}^{(k+1)}, \alpha_1, \alpha_2, \beta, N, K)$

$k \leftarrow k + 1$

end

$\beta \leftarrow \beta(\rho + 1)$

end

return $\mathcal{L}(\mathbf{w}^{(k+1)})$

Function `w_update(w, U, λ, β, N, K):`

$\nabla_{\mathbf{w}} f \leftarrow \mathcal{L}^* \left(\mathcal{L}(\mathbf{w}) - \mathbf{U} \text{diag}(\boldsymbol{\lambda}) \mathbf{U}^T + \frac{\mathbf{K}}{\beta} \right)$

return $\max \left(0, \mathbf{w} - \frac{\nabla_{\mathbf{w}} f}{2N} \right)$

Function `U_update(w, N, K):`

return `eigenvectors(L(w))[K+1:N]` # *increasing order w.r.t. eigenvalues*

Function `lambda_update(w, U, α1, α2, β, N, K):`

$\mathbf{d} \leftarrow \text{diag}(\mathbf{U}^T \mathcal{L}(\mathbf{w}) \mathbf{U})$

$\boldsymbol{\lambda} \leftarrow \frac{1}{2} \left(\mathbf{d} + \sqrt{\mathbf{d} \odot \mathbf{d} + \frac{4}{\beta}} \right)$ # \odot *means element-wise multiplication*

if $\boldsymbol{\lambda}$ *has its elements in nondecreasing order* **and** $\min(\boldsymbol{\lambda}) \geq \alpha_1$ **and** $\max(\boldsymbol{\lambda}) \leq \alpha_2$ **then**

return $\boldsymbol{\lambda}$

else

 set to α_1 the elements of $\boldsymbol{\lambda}$ whose values are less than α_1

 set to α_2 the elements of $\boldsymbol{\lambda}$ whose values are greater than α_2

end

if $\boldsymbol{\lambda}$ *has its elements in nondecreasing order* **then**

return $\boldsymbol{\lambda}$

else

raise `Exception("eigenvalues are not in increasing order")`

end

References