Algebra and Statistical Methods for Big Data with Bioconductor

Dolors Pelegri [1,2,3] ***and Juan R. Gonzalez*** *[1,2,3]

[1]Autonomous University of Barcelona (UAB)
[2]ISGlobal, Centre for Research in Environmental Epidemiology (CREAL)
[3]Bioinformatics Research Group in Epidemiolgy (BRGE)

*juanr.gonzalez@isglobal.org

2019-06-13

Abstract

Description of functions to perform matrix operations, algebra and some statistical models using *DelayedMatrix* objects.

Package

BigDataStatMeth 1.0

Contents

1 Prerequisites . . . 3
2 Overview . . . 3
3 Getting started . . . 3
4 Matrix Multiplication . . . 3
4.1 Simple matrix multiplication . . . 3
4.2 Block matrix multiplication . . . 5
5 Cross-product and Transposed Cross-product . . . 6
6 Matrix Vector Multiplication . . . 7
6.1 Weighted Cross-product and Weighted Transposed Cross-product . . . 7
6.2 Weighted Transposed Cross Product . . . 7
7 Inverse Cholesky . . . 8
8 Singular Value Decomposition (SVD) . . . 9
9 Using algebra procedure for `DelayedMatrix` objects to implement basic statistical methods . . . 10

9.1 Principal component analysis (PCA) 10

10 Statistical methods implemented for `DelayedMatrix` objects . . . 12

10.1 Lasso regression 12

11 Session information 14

1 Prerequisites

The package requires other `Bioconductor` packages to be installed. These include: `beachmat`, `HDF5Array`, `DelayedArray`. Other required R packages are: `Matrix`, `RcppEigen` and `RSpectra`.

2 Overview

This package implements seveal matrix operations using `DelayMatrix`objects, as well as some basic algebra operations that can be used to carry out some statistical analyses using standard methodologies such as principal component analyses or least squares estimation. The package also contains specific statatistical method mainly used in `omic` data analysis such as lasso regression.

3 Getting started

First, let us start by loading the required packages to describe the main capabilities of the package

```
library(BigDataStatMeth)
library(DelayedArray)
```

These other packages are required to reproduce this vignette

```
library(microbenchmark)
```

4 Matrix Multiplication

In this section, different products of matrices and vectors are introduced. The methods implement different strategies including block multiplication algorithms and the use of parallel implementations.

4.1 Simple matrix multiplication

The function `blockmult()` performs a simple sequential matrix multiplication. The function allows the use of `DelayedMAtrix` objects as well as standard R matrices.

Let us simulate to set of matrices to illustrate the use of the function accross the entire documment. First, we simulate a simple case with to matrices A and B with dimensions 500x500 and 500x750, respectively. Second, another example with dimensions 1000x10000 are use to evaluate the performance in large matrices. The examples with big datasets will be illustrated using real data belonging to different omic settings. We can simulate to matrices with the desired dimensions by

```
set.seed(123456)
n <- 500
p <- 750
A <- matrix(rnorm(n*n), nrow=n, ncol=n)
B <- matrix(rnorm(n*p), nrow=n, ncol=p)
```

```
n <- 1000
p <- 10000
Abig <- matrix(rnorm(n*n), nrow=n, ncol=n)
Bbig <- matrix(rnorm(n*p), nrow=n, ncol=p)
```

They can be converted into `DelayedMatrix` object by simply

```
DA <- DelayedArray(A)
DB <- DelayedArray(B)
```

Matrix multiplication is then done by

```
AxB <- blockmult(A, B)
AxBDelay <- blockmult(DA, DB)
AxBDelay [1:5,1:5]
           [,1]      [,2]      [,3]       [,4]      [,5]
[1,] -24.798007  25.79546  12.36855 -48.377086 19.950507
[2,]  24.121860 -14.53789  30.79320 -26.773787 15.879197
[3,]  26.791564 -19.11443 -10.44961 -13.119789 -9.882985
[4,]   1.310851 -30.86637  24.89040   8.327157 -3.912149
[5,] -28.554251  14.63744  23.84582 -19.025553  7.251492
```

This verifies that computations are fine, even when comparing with matrix multiplication with R (i.e. using '%*%)

```
all.equal(AxB, AxBDelay)
[1] TRUE
all.equal(A%*%B, AxBDelay)
[1] TRUE
```

The process can be speed up by making computations in parallel using `paral=TRUE`.

```
AxB <- blockmult(A, B, paral = TRUE)
AxBDelay <- blockmult(DA, DB, paral = TRUE)

all.equal(AxB, AxBDelay)
[1] TRUE
```

We can show that the parallel version really improves the computational speed and also how the blockmult function improves the R implementation.

```
microbenchmark(
  R = A%*%B,
  noparallel = blockmult(A, B),
  parallel = blockmult(A, B, paral=TRUE),
  times = 10)
Unit: milliseconds
       expr      min       lq     mean   median       uq       max neval cld
          R 74.19056 75.64359 81.91089 78.98127 85.29771  97.19538    10   b
 noparallel 41.15552 46.07957 79.19409 51.21616 73.25150 285.61242    10   b
   parallel 23.92136 24.34092 26.41831 25.73766 28.57321  30.78714    10  a
```

4.2 Block matrix multiplication

A block matrix or a partitioned matrix is a matrix that is interpreted as having been broken into sections called blocks or submatrices. Intuitively, a block matrix can be visualized as the original matrix with a collection of horizontal and vertical lines, which break it up, or partition it, into a collection of smaller matrices. More information can be found here.

Matrix multiplication using block matrices is implemented in the `blockmult()` function. It only requires to setting the argument `block_size` different from 0.

```
AxB <- blockmult(A, B, block_size = 10)
AxBDelay <- blockmult(DA, DB, block_size = 10 )
```

As expected the results obtained using this procedure are the correct ones

```
all.equal(AxBDelay,A%*%B)
[1] TRUE
all.equal(AxB, AxBDelay)
[1] TRUE
```

Note that when the argument `block_size` is larger than any of the dimensions of matrix A or B the `blocks_size` is set to `min(cols(A), rows(A), cols(B), rows(B))`.

As in the case of using a simple matrix multiplication, one can make the operations in parallel with `paral = TRUE` .

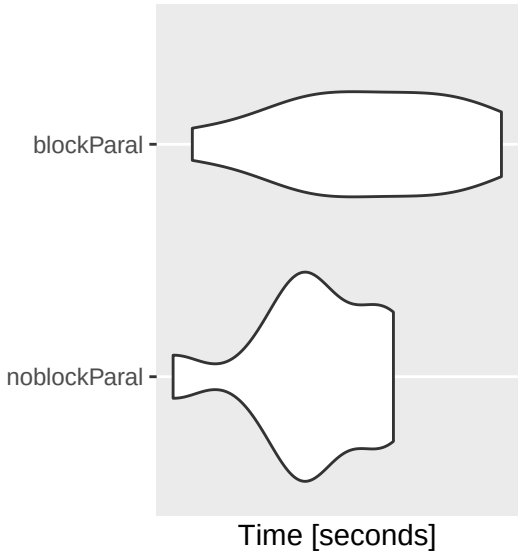
```
AxB <- blockmult(A, B, block_size = 10, paral = TRUE)
AxBDelay <- blockmult(DA, DB, block_size = 10, paral = TRUE )

all.equal(AxBDelay,A%*%B)
[1] TRUE
all.equal(AxB, AxBDelay)
[1] TRUE
```

Here, we compare the performace of the block method with the one implemented with the simplest case.

```
bench1 <- microbenchmark(
  noblockParal = blockmult(Abig, Bbig, paral = TRUE),
  blockParal = blockmult(Abig, Bbig, block_size = 100,
                         paral=TRUE),
  times = 10)
bench1
Unit: seconds
         expr      min       lq     mean   median       uq      max neval
 noblockParal 1.190175 1.380861 1.410880 1.394545 1.513259 1.544131    10
   blockParal 1.217531 1.385174 1.505026 1.500299 1.643682 1.754333    10
 cld
   a
   a
```

The same information is depicted in the next Figure which illustrates the comparison between the different assessed methods

```
autoplot.microbenchmark(bench1)
```



5 Cross-product and Transposed Cross-product

To perform a cross-product $C = A^t A$ you can call `bdcrossprod()` function.

```
n <- 500
m <- 250
A <- matrix(rnorm(n*m), nrow=n, ncol=m)
DA <- DelayedArray(A)

# Cross Product of a standard R matrix
cpA <- bdcrossprod(A)
# Result with DelayedArray data type
cpDA <- bdcrossprod(DA)
```

We obtain the expected values computed using `crossprod` function

```
all.equal(cpDA, crossprod(A))
[1] TRUE
```

you may also set `transposed=TRUE` (default value transposed=false) to get a transposed cross-product $C = AA^t$

```
# Transposed Cross Product R matrices
tcpA <- bdcrossprod(A, transposed = TRUE)
# With DelayeArray data types
tcpDA <- bdcrossprod(DA, transposed = TRUE)
```

We obtain the expected values computed using `tcrossprod` function

```
all.equal(tcpDA, tcrossprod(A))
[1] TRUE
```

6 Matrix Vector Multiplication

You can perform a weighted cross-product $C = X^t w X$ with `bdwcrossprod()` given a matrix X as argument and a vector or matrix of weights, w.

6.1 Weighted Cross-product and Weighted Transposed Cross-product

```
n <- 250
X <- matrix(rnorm(n*n), nrow=n, ncol=n)
u <- runif(n)
w <- u * (1 - u)
DX <- DelayedArray(X)
Dw <- DelayedArray(as.matrix(w))

wcpX <- bdwproduct(X, w,"xwxt")
wcpDX <- bdwproduct(DX, Dw,"xwxt") # with DelayedArray

wcpDX[1:5,1:5]
           [,1]       [,2]       [,3]       [,4]       [,5]
[1,] 40.3295560  2.0580419 -5.2602942  1.1959968  0.8017478
[2,]  2.0580419 44.3292643 -0.6334476  2.5054101 -1.7216231
[3,] -5.2602942 -0.6334476 40.5930654 -3.9879412  1.4271090
[4,]  1.1959968  2.5054101 -3.9879412 42.4654763  0.2056653
[5,]  0.8017478 -1.7216231  1.4271090  0.2056653 37.7384466
```

Those are the expected values as it is indicated by executing:

```
all.equal( wcpDX, X%*%diag(w)%*%t(X) )
[1] TRUE
```

6.2 Weighted Transposed Cross Product

With argument `transposed=TRUE`, we can perform a transposed weighted cross-product $C = AwA^t$

```
wtcpX <- bdwproduct(X, w,"xtwx")
wtcpDX <- bdwproduct(DX, Dw,"xtwx") # with DelayedArray

wtcpDX[1:5,1:5]
           [,1]       [,2]       [,3]       [,4]       [,5]
[1,] 43.1219962  0.1696051 -2.7444637  2.1840958  1.9790205
[2,]  0.1696051 39.8331423  2.4548163 -4.6186496 -0.9269378
[3,] -2.7444637  2.4548163 32.3755398  0.2445101  3.0163303
```

```
[4,]  2.1840958 -4.6186496  0.2445101 37.9577426 -3.1118048
[5,]  1.9790205 -0.9269378  3.0163303 -3.1118048 38.0599703
```

Those are the expected values as it is indicated by executing:

```
all.equal(wtcpDX, t(X)%*%diag(w)%*%X)
[1] TRUE
```

7 Inverse Cholesky

The Cholesky factorization is widely used for solving a system of linear equations whose coefficient matrix is symmetric and positive definite.

$$A = LL^t = U^tU$$

where L is a lower triangular matrix and U is an upper triangular matrix. To get the Inverse Cholesky we can use the function `bdInvCholesky()`. Let us start by illustrating how to do this computations in a simulated data:

```
# Generate a positive definite matrix
Posdef <- function (n, ev = runif(n, 0, 10))
{
  Z <- matrix(ncol=n, rnorm(n^2))
  decomp <- qr(Z)
  Q <- qr.Q(decomp)
  R <- qr.R(decomp)
  d <- diag(R)
  ph <- d / abs(d)
  O <- Q %*% diag(ph)
  Z <- t(O) %*% diag(ev) %*% O
  return(Z)
}

A <- Posdef(n = 500, ev = 1:500)
DA <- DelayedArray(A)

invchol <- bdInvCholesky(A)
Dinvchol <- bdInvCholesky(DA)

round(invchol[1:5,1:5],8)
            [,1]        [,2]        [,3]        [,4]        [,5]
[1,]  0.01250575 -0.00074024 -0.00118527  0.00671543 -0.00051009
[2,] -0.00074024  0.01391104 -0.00191608 -0.00004815  0.00040529
[3,] -0.00118527 -0.00191608  0.01336471 -0.00337010 -0.00196165
[4,]  0.00671543 -0.00004815 -0.00337010  0.01686427 -0.00049964
[5,] -0.00051009  0.00040529 -0.00196165 -0.00049964  0.00992266
```

We can check whether this function returns the expected values obtained with the standard R function `solve`:

```
all.equal(Dinvchol, solve(A))
```

8 Singular Value Decomposition (SVD)

The SBD of an $m \times n$ real or complex matrix A is a factorization of the form:

$$U\Sigma V^T$$

where :

-U is a $m \times m$ real or complex unitary matrix -Σ is a $m \times n$ rectangular diagonal matrix with non-negative real numbers on the diagonal -V is a $n \times n$ real or complex unitary matrix.

Notice that:

- The diagonal entries σ_i of Σ are known as the singular values of A.
- The columns of U are called the left-singular vectors of A.
- The columns of V are called the right-singular vectors of A.

He have implemented the SVD for R matrices and Delayed Array objects in the function `bdSVD()`. The method, so far, only allows SVD of real matrices A. This code illustrate how to perform such computations:

```
# Matrix simulation
n <- 500
A <- matrix(rnorm(n*n), nrow=n, ncol=n)
# Get a Delayed Array object
DA <- DelayedArray(A)

# SVD
bsvd <- bdSVD(A)
Dbsvd <- bdSVD(DA)

# Singular values, and right and left singular vectors
bsvd$d[1:5]
[1] 44.37367 44.10619 43.54784 43.06513 42.80271
bsvd$u[1:5,1:5]
             [,1]         [,2]         [,3]         [,4]         [,5]
[1,]  0.004974032 -0.005436497 -0.027983648  0.031127774 -0.013121734
[2,]  0.041644408 -0.002126621 -0.008594309 -0.055505949  0.014637588
[3,] -0.044234679  0.007019281 -0.026461123  0.012142166 -0.033046183
[4,]  0.042124039  0.036970495 -0.072668517 -0.008657028 -0.054262293
[5,]  0.066605320 -0.013733819  0.018173796  0.080117619 -0.005597533
bsvd$v[1:5,1:5]
            [,1]         [,2]         [,3]        [,4]          [,5]
[1,] -0.02516457 -0.085854351 -0.020345943  0.04086313  0.0125316365
[2,]  0.03300432  0.056061409 -0.007931696  0.01834868  0.0101005472
[3,] -0.02095691  0.011892097  0.050165799 -0.06891750 -0.0465905911
[4,] -0.01712536 -0.007154121  0.069159262  0.06628704 -0.0244923423
[5,]  0.10319287 -0.026417551 -0.086822587  0.04482255 -0.0004748714
```

We get the expected results obtained with standard R functions:

```
all.equal( sqrt( svd( tcrossprod( scale(A) ) )$d[1:10] ), bsvd$d[1:10] )
[1] TRUE
all.equal( sqrt( svd( tcrossprod( scale(A) ) )$d[1:10] ), Dbsvd$d[1:10] )
[1] TRUE
```

you get the σ_i, U and V of normalized matrix A, if you want to perform the SVD from not normalized matrix A then you have to set the parameter `normalize = false`

```
bsvd <- bdSVD(A, normalize = FALSE)
Dbsvd <- bdSVD(DA, normalize = FALSE)

bsvd$d[1:5]
[1] 44.48473 44.07657 43.47049 43.13594 42.90882
bsvd$u[1:5,1:5]
             [,1]         [,2]        [,3]        [,4]         [,5]
[1,]  0.006904277  0.002695548 -0.02901511  0.01916610  0.025252109
[2,]  0.049478987  0.008109244 -0.01792460 -0.06256968  0.012133517
[3,] -0.045456805  0.016340642 -0.03806440  0.02469420 -0.001825009
[4,]  0.048267031  0.039362228 -0.07163863  0.01675956 -0.010596715
[5,]  0.060383947 -0.019886575  0.02251988  0.07434948  0.004867027
bsvd$v[1:5,1:5]
            [,1]          [,2]         [,3]         [,4]        [,5]
[1,] -0.01148633 -0.0859791338 -0.021813211  0.030703273 -0.02665914
[2,]  0.02610092  0.0440443821 -0.006534317 -0.001030926 -0.08331031
[3,] -0.02532011  0.0160447606  0.054294204 -0.060052911  0.03272444
[4,] -0.02741625 -0.0008363462  0.074549396  0.038471909 -0.06149153
[5,]  0.11901307 -0.0180469223 -0.097572899  0.085265505 -0.01592488

all.equal( sqrt(svd(tcrossprod(A))$d[1:10]), bsvd$d[1:10] )
[1] TRUE
all.equal( sqrt(svd(tcrossprod(A))$d[1:10]), Dbsvd$d[1:10] )
[1] TRUE
```

9 Using algebra procedure for `DelayedMatrix` objects to implement basic statistical methods

9.1 Principal component analysis (PCA)

Let us illustrate how to perform a PCA using miRNA data obtained from TCGA corresponding to 3 different tumors: melanoma (ME), leukemia (LEU) and centran nervous system (CNS). Data is available at `BigDataStatMeth` and can be loaded by simply:

```
data(miRNA)
dim(miRNA)
[1]  21 537
```

We observe that there are a total of 21 individuals and 537 miRNAs. The vector `cancer` contains the type of tumor of each individual. For each type we have:

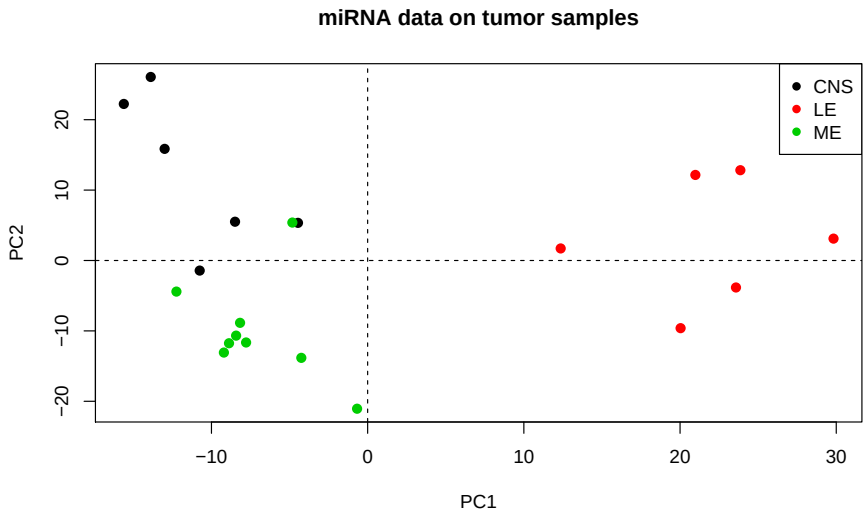
```
table(cancer)
cancer
CNS  LE  ME
  6   6   9
```

Now, the typical principal component analysis on the samples can be run on the `miRNA` matrix since it has miRNAs in columns and individuals in rows

```
pc <- prcomp(miRNA)
```

We can plot the two first components with:

```
plot(pc$x[, 1], pc$x[, 2],
     main = "miRNA data on tumor samples",
     xlab = "PC1", ylab = "PC2", type="n")
abline(h=0, v=0, lty=2)
points(pc$x[, 1], pc$x[, 2], col = cancer,
       pch=16, cex=1.2)
legend("topright", levels(cancer), pch=16, col=1:3)
```

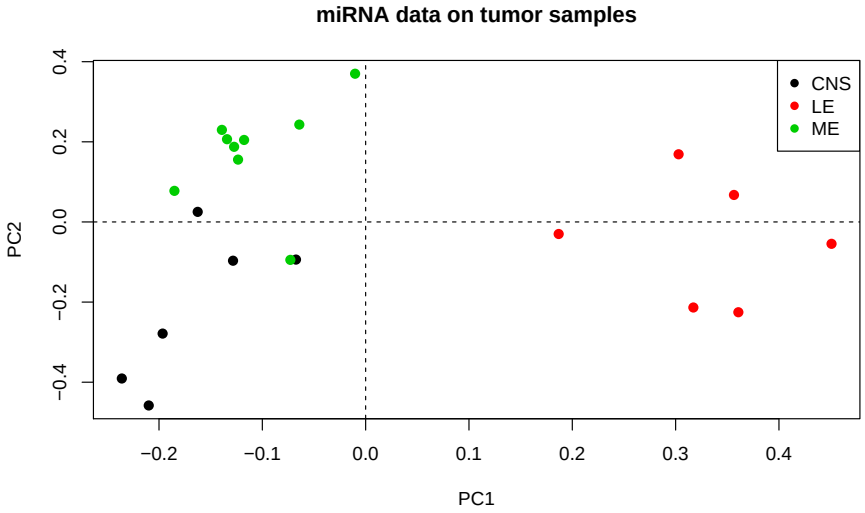


The same analysis can be performed using SVD decomposition and having miRNAs as a `DelayedMatrix` object. The PCA is equivalent to performing the SVD on the centered data, where the centering occurs on the columns. In that case the function `bdSVD` requires to set the argument `normalize` equal to TRUE. Notice that `sweep()` and `colMeans()` functions can be applied to a `DelayedMAtrix` object since this method is implemented for that type of objects in the `DelayedArray` package.

```
miRNAD <- DelayedArray(miRNA)
miRNAD.c <- DelayedArray::sweep(miRNAD, 2,
      DelayedArray::colMeans(miRNAD), "-")
svd.da <- bdSVD(miRNAD.c, normalize = FALSE)
```

The PCA plot for the two principal components can then be be obtained by:

```
plot(svd.da$u[, 1], svd.da$u[, 2],
     main = "miRNA data on tumor samples",
     xlab = "PC1", ylab = "PC2", type="n")
abline(h=0, v=0, lty=2)
points(svd.da$u[, 1], svd.da$u[, 2], col = cancer,
       pch=16, cex=1.2)
legend("topright", levels(cancer), pch=16, col=1:3)
```



We can observe that both figures are equal irrespective to a sign change of second component (that can happen in SVD).

10 Statistical methods implemented for `DelayedMatrix` objects

In this section we illustrate how to estimate some of the state of the art methods used in omic data analyses having a `DelayedMatrix` as the input object. So far, we have implemented the lasso regression, but other methods are going to be implemented in the near future.

10.1 Lasso regression

The standard linear model (or the ordinary least squares method) performs poorly in a situation, where you have a large multivariate data set containing a number of variables superior to the number of samples. This is the case, for instance, of genomic, transcriptomic or epigenomic studies where the number of variables (SNPs, genes, CpGs) exceed the number of samples (i.e. transcriptomic analysis deals with ~20,000 genes on hundreds of individuals).

A better alternative is the penalized regression allowing to create a linear regression model that is penalized, for having too many variables in the model, by adding a constraint in the equation. This is also known as shrinkage or regularization methods. Lasso regression is one of the methods which are based on this idea. Lasso stands for Least Absolute Shrinkage

and Selection Operator. It shrinks the regression coefficients toward zero by penalizing the regression model with a penalty term called L1-norm, which is the sum of the absolute coefficients.

In the case of lasso regression, the penalty has the effect of forcing some of the coefficient estimates, with a minor contribution to the model, to be exactly equal to zero. This means that, lasso can be also seen as an alternative to the subset selection methods for performing variable selection in order to reduce the complexity of the model, or in other words, the number of variables. Selecting a good value of λ is critical.

This tunning parameter can be estimated using cross-validation. However, this procedure can be very time consuming since the inversion of a matrix is highly time consuming and it must be computed for each cross-validated sub-sample. In order to avoid this, we have implemented a method proposed by Rifkin and Lippert who stated that in the context of regularized least squares, one can search for a good regularization parameter λ at essentially no additional cost (i.e. estimating only one inverion matrix).

Let us illustrate how this works in our package. Let us start by simulating a dataset with 200 variables when only 3 of them are associated with the outcome.

```
# number of samples
n <- 500
# number of variables
p <- 200
# covariates
M <- matrix(rnorm(n*p, mean=8, sd=2), nrow=n, ncol=p)
# outcome (only variables 1, 2 and 5 are different from 0)
Y <- 2.4*M[,1] + 1.6*M[,2] - 0.4*M[,5] + rnorm(n, sd=0.1)
```

Then the model can be fitted (using `DelayedMatrix` objects) by

```
# Get DealyedArray matrices
MD <- DelayedArray(M)
YD <- DelayedArray(as.matrix(Y))

# Model
mod <- LOOE(MD, YD, paral=FALSE, l=0.0001)
mod$coef[abs(mod$coef)>mean(mod$coef)]
[1]  2.4009548  1.6067177 -0.3980939
```

The argument `paral` is required to indicate whether paralell implementation is used or not. The `lambda` argument can be provided by the user, but it can also be estimated if `l` parameter is missing:

```
library(glmnet)
mod.cv <- cv.glmnet(M, Y)
mod.glmnet <- glmnet(M, Y, lambda=mod.cv$lambda.min)
mod.glmnet$beta[1:8,]
       V1        V2        V3        V4        V5        V6        V7
 2.350086  1.555629  0.000000  0.000000 -0.347607  0.000000  0.000000
       V8
 0.000000
```

11 Session information

```
sessionInfo()
R version 3.5.0 (2018-04-23)
Platform: x86_64-w64-mingw32/x64 (64-bit)
Running under: Windows >= 8 x64 (build 9200)

Matrix products: default

locale:
[1] LC_COLLATE=Spanish_Spain.1252  LC_CTYPE=Spanish_Spain.1252
[3] LC_MONETARY=Spanish_Spain.1252 LC_NUMERIC=C
[5] LC_TIME=Spanish_Spain.1252

attached base packages:
[1] stats4    parallel  stats     graphics  grDevices utils     datasets
[8] methods   base

other attached packages:
 [1] BigDataStatMeth_1.0  glmnet_2.0-16        foreach_1.4.4
 [4] Matrix_1.2-14        microbenchmark_1.4-4 DelayedArray_0.6.6
 [7] BiocParallel_1.14.2  IRanges_2.14.10      S4Vectors_0.18.3
[10] BiocGenerics_0.28.0  matrixStats_0.54.0   BiocStyle_2.10.0

loaded via a namespace (and not attached):
 [1] tinytex_0.13        RcppEigen_0.3.3.5.0 tidyselect_0.2.5
 [4] zoo_1.8-1           xfun_0.7            beachmat_1.2.1
 [7] purrr_0.2.4         HDF5Array_1.8.0     splines_3.5.0
[10] lattice_0.20-35     rhdf5_2.24.0        colorspace_1.3-2
[13] htmltools_0.3.6     yaml_2.2.0          survival_2.41-3
[16] rlang_0.3.4         pillar_1.4.0        glue_1.3.1
[19] multcomp_1.4-8      plyr_1.8.4          stringr_1.3.1
[22] munsell_0.5.0       gtable_0.2.0        mvtnorm_1.0-10
[25] codetools_0.2-15    evaluate_0.13       knitr_1.22
[28] TH.data_1.0-8       Rcpp_1.0.1          scales_1.0.0
[31] BiocManager_1.30.4  RcppParallel_4.4.1  ggplot2_3.0.0
[34] digest_0.6.15       stringi_1.2.2       dplyr_0.8.1
[37] bookdown_0.11       grid_3.5.0          tools_3.5.0
[40] sandwich_2.4-0      magrittr_1.5        tibble_2.1.1
[43] lazyeval_0.2.1      pkgconfig_2.0.2     crayon_1.3.4
[46] MASS_7.3-50         assertthat_0.2.1    rmarkdown_1.13
[49] iterators_1.0.9     R6_2.4.0            Rhdf5lib_1.2.0
[52] compiler_3.5.0
```