Algebra and Statistical Methods for Big Data with Bioconductor

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

[1]Universitat de Vic - Universitat Central de Catalunya (UVic)
[2]ISGlobal, Centre for Research in Environmental Epidemiology (CREAL)
[3]Bioinformatics Research Group in Epidemiolgy (BRGE)

*juanr.gonzalez@isglobal.org

2020-08-30

Abstract

Description of functions to perform matrix operations, algebra and some statistical models using *DelayedMatrix* objects and HDF5 data files.

Package

BigDataStatMeth 1.1

Contents

1 Prerequisites

The package requires other `Bioconductor` packages to be installed. These include: `beachmat`, `HDF5Array`, `DelayedArray`, `rhdf5`. Other required R packages are: `Matrix`, `RcppEigen` and `RSpectra`.

2 Overview

This package implements seveal matrix operations using `DelayMatrix` objects and HDF5 data files 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)
library(rhdf5)
library(DT)
```

These other packages are required to reproduce this vignette

```
library(microbenchmark)
```

4 Previous knowledge

4.1 HDF5 data files

Hierarchical Data Format (HDF) is a set of file formats designed to store and organize large amounts of data. It is supported by The HDF Group, a non-profit corporation whose mission is to ensure continued development of HDF5 technologies and the continued accessibility of data stored in HDF.

HDF5 is a technology suite that makes possible the management of extremely large and complex data collections, can accomodate virtually every kind of data in a single file, sequences, images, SNP matrices, and every other type of data and metadata associated with an experiment.

There is no limit on the number or size of data objects in the collection, giving great flexibility for omic data. Is high-performance I/O with a rich set of integrated performance features that allow for access time and storage space optimizations

HDF5 file structure include only *two major types of object*:

- `Datasets`, which are multidimensional arrays of a homogeneous type, in datasets we can store omics data associated to genomics, transcriptomics, epigonomics, proteomics, metabolomics experiments
- `Groups`, which are container structures which can hold datasets and other groups

This results in a truly hierarchical, filesystem-like data format

4.2 Basics in rhdf5

Create hdf5 file

The `Create_HDF5_matrix_file` function implemented in this package creates an hdf5 file with a group and a dataset in one step. This function allows to create datasets from `DelayedMAtrix` objects using `beachmat` library as well as standard R matrices.

```
library(rhdf5)
library(DelayedArray)

set.seed(5234)
n <- 100
m <- 10000
A <- matrix(rnorm(n*m,mean=0,sd=1), n,m)

Ad <- DelayedArray(A)

# Create a file with a dataset from DelayedMatrix Ad in INPUT group
Create_HDF5_matrix_file("delayed.hdf5", Ad, "INPUT", "A")
[1] 0

# We also can create a dataset from R matrix object
Create_HDF5_matrix_file("robjects.hdf5", A, "INPUT", "A")
[1] 0
```

Add datasets in hdf5 file

The `Create_HDF5_matrix` allows to create a dataset in existing file. We can create the dataset in any group, if group doesn't exists in file, the group is created before append the dataset.

```
set.seed(5234)
n <- 50
m <- 12000
B <- matrix(rnorm(n*m,mean=3,sd=0.5), n,m)

Bd <- DelayedArray(B)

# Create dataframe B with Bd matrix in delayed.hdf5 file at INPUT group
Create_HDF5_matrix(Bd, "delayed.hdf5", "INPUT", "B");
[1] 0

# Create dataframe data with Ad matrix in delayed.hdf5 file at OMIC group
set.seed(5234)
n <- 150000
```

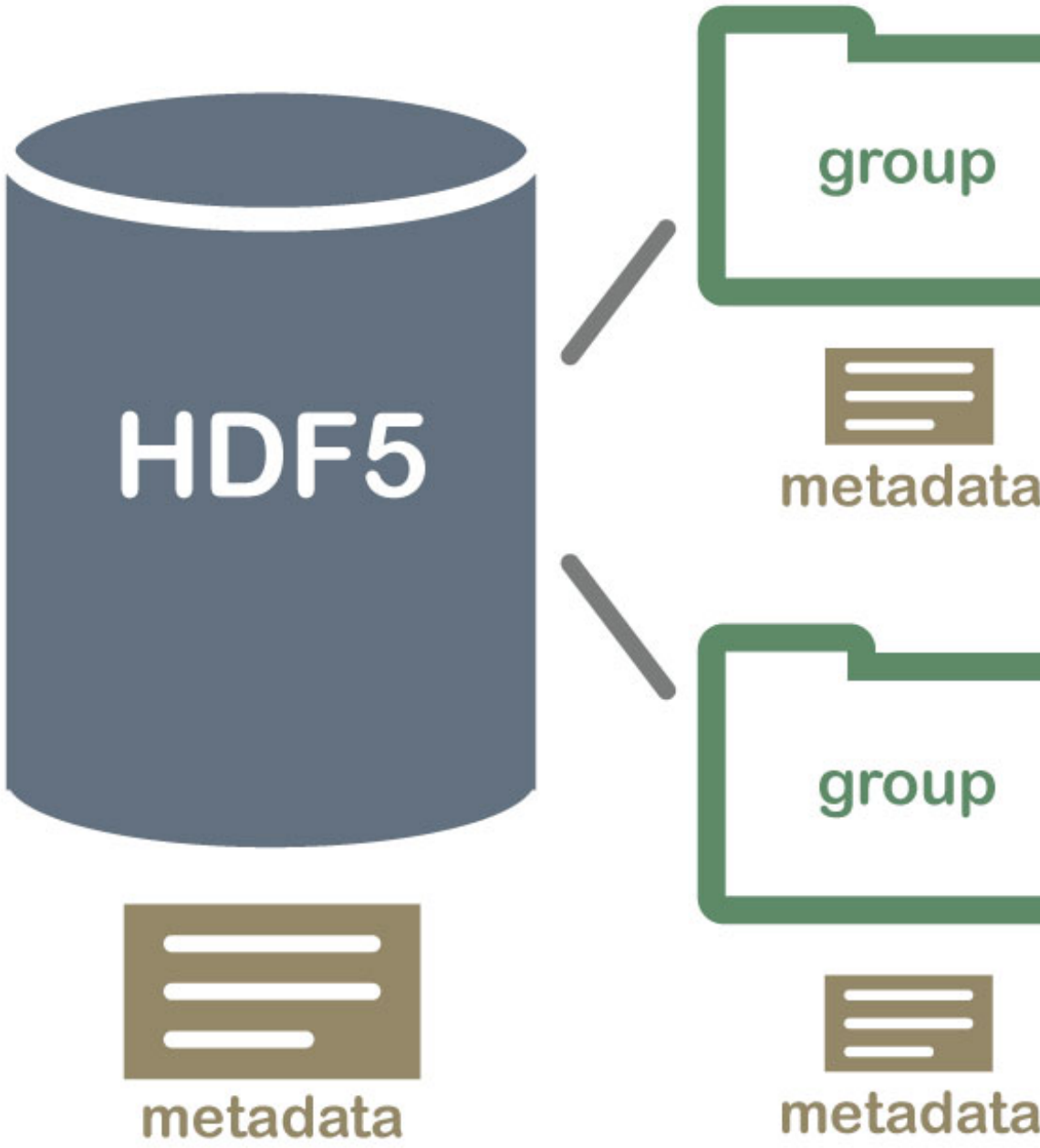


Figure 1: HDF5 hierarchical structure

```
m <- 50
odata <- matrix(rnorm(n*m,mean=0,sd=1), n,m)

Create_HDF5_matrix(odata, "delayed.hdf5", "OMICS", "data")
[1] 0
```

Open and get hdf5 content file

We can open an existing file show contents and access data using functions from `rhdf5` package. `rhdf5` is an R interface for HDF5. The file must always be opened before working with it.

```
# Examine hierarchy before open file
h5ls("delayed.hdf5")
   group  name       otype dclass         dim
0      / INPUT   H5I_GROUP
1 /INPUT     A H5I_DATASET  FLOAT 100 x 10000
2 /INPUT     B H5I_DATASET  FLOAT  50 x 12000
3      / OMICS   H5I_GROUP
4 /OMICS  data H5I_DATASET  FLOAT 50 x 150000

# Open file
h5fdelay = H5Fopen("delayed.hdf5")
# Show hdf5 hierarchy (groups)
h5fdelay
HDF5 FILE
        name /
    filename

   name     otype dclass dim
0 INPUT H5I_GROUP
1 OMICS H5I_GROUP
```

Access datasets data

The $ operator can be used to access the next group level, this operator reads the object from disk. We can assign the dataset contents to an R object in order to work with it.

```
Adata = h5fdelay$OMICS$data
Adata[1:3,1:6]
           [,1]       [,2]       [,3]       [,4]       [,5]        [,6]
[1,]  1.1245693 -0.6018349  1.6859794 0.80048363 0.82490435  0.84301868
[2,] -0.5946380 -1.3731891 -0.5045485 0.08992737 0.07191119  0.02315468
[3,]  0.1152881  0.5035297  0.5205231 0.64510203 1.47849346 -0.76030543
```

Close hdf5 file

After work with hdf5 we always must close the file. We can close only one file with `H5Fclose` function or close all hdf5 opend files with `h5closeAll` function

```
# Close delayed.hdf5 file
H5Fclose(h5fdelay)

# Open 2 files and close all
```

```
h5fdelay = H5Fopen("delayed.hdf5")
h5fr = H5Fopen("robjects.hdf5")

h5closeAll()
```

More information about working with hdf5 files in r (Fischer, Pau, and Smith 2019)

5 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.

5.1 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,
  noparal = blockmult(A, B),
  paral = blockmult(A, B, paral=TRUE),
  times = 10)
Unit: milliseconds
    expr       min        lq      mean    median        uq       max neval cld
       R 118.09329 118.39102 126.50714 119.30112 124.62642 171.76494    10   b
 noparal  19.23592  20.15665  22.25904  21.78953  23.67284  28.16974    10  a
   paral  19.38160  20.59651  24.93529  21.31785  30.90419  34.90519    10  a
```

With `paral=TRUE`, an optional parameter threads can be used to indicate the number of `threads` to launch simultaneously, if `threads=NULL` the function takes the available threads - 1, leaving one available for user.

```
AxB <- blockmult(A, B, paral = TRUE, threads = 2)
AxBDelay <- blockmult(DA, DB, paral = TRUE, threads = 3)

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

We can show how the number of threads affects performance.

```
microbenchmark(
  AxBnThread2 = blockmult(DA, DB, paral = TRUE, threads = 2) ,
  AxBnThread3 = blockmult(DA, DB, paral = TRUE, threads = 3) ,
  AxBnThread4 = blockmult(DA, DB, paral = TRUE, threads = 4) ,
  times = 10)
Unit: milliseconds
        expr      min       lq     mean   median       uq      max neval cld
```

```
AxBnThread2 28.99276 29.40934 31.24777 30.42192 31.99013 37.38304    10   b
AxBnThread3 29.45354 29.79419 31.00747 30.40579 31.77683 35.36662    10   b
AxBnThread4 21.77878 21.88681 24.81579 24.53696 26.98818 30.05085    10  a
```

5.2 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. the implementation has been made from the adaptation of the Fox algorithm [1].

Matrix multiplication using block matrices is implemented in the `blockmult()` function. It only requires to setting the argument `block_size` different from 0, by default `block_size = 128`.

```
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` and optionally indicating the number of threads to use in parallel computing .

```
AxB <- blockmult(A, B, block_size = 10, paral = TRUE)
AxBDelay <- blockmult(DA, DB, block_size = 10, paral = TRUE )
AxBDelayT <- blockmult(DA, DB, block_size = 10, paral = TRUE, threads =  3 )

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

To work with big matrices `blockmult()` saves matrices in hdf5 file format in order to be able to operate with them in blocks and not overload the memory, by default are considered large matrices if the number of rows or columns is greater than 5000, but it can be changed with `bigmatrix` argument.

```
DAbig <- DelayedArray(Abig)
DBbig <- DelayedArray(Abig)
```

```
# We consider a big matrix if number of rows or columns are > 500
AxBBig3000 <- blockmult(DAbig, DBbig, bigmatrix = 500)

# We want to force it to run in memory
AxBNOBig <- blockmult(DAbig, DBbig, bigmatrix = 100000)
```

Depending on whether the calculation has been performed directly in memory or from an hdf5 file, the returned object type is different.

If we work in memory results are returned as a current r matrix object,

```
class(AxBNOBig)
[1] "matrix"
AxBNOBig[1:5,1:5]
           [,1]       [,2]      [,3]       [,4]      [,5]
[1,]   6.906941  56.240861 -81.68187 -36.925101 -42.36525
[2,]   8.353795  79.371434  -3.13127  19.532005 -29.78580
[3,] -28.130259 -22.960250  10.39253  -1.524004  13.95016
[4,]   6.917426  36.972565 -18.90454   4.413286  10.56029
[5,] -58.796247   8.764894  26.81915  68.773862  28.42892
```

if we work in disk, we return an hdf5 object with two groups (hdf5 group is a container mechanism by which HDF5 files are organized), INPUT and OUTPUT, in INPUT group we have the input matrices A and B, in OUTPUT group we get the resulting matrix C.

To work in R with hdf5 data object we can use the `rhdf5` function library from `bioconductor`,

```
class(AxBBig3000)
[1] "list"

# Assign hdf5 result content to R variable reshdf5
reshdf5 <- AxBBig3000$res$OUTPUT$C
reshdf5[1:5,1:5]
           [,1]       [,2]      [,3]       [,4]      [,5]
[1,]   6.906941  56.240861 -81.68187 -36.925101 -42.36525
[2,]   8.353795  79.371434  -3.13127  19.532005 -29.78580
[3,] -28.130259 -22.960250  10.39253  -1.524004  13.95016
[4,]   6.917426  36.972565 -18.90454   4.413286  10.56029
[5,] -58.796247   8.764894  26.81915  68.773862  28.42892

all.equal(reshdf5, AxBNOBig)
[1] TRUE
```

Like in rhdf5 it is important to close all dataset, group, and file handles when not used anymore,

```
bdclose(AxBBig3000)
```

Here, we compare the performace of the block method with the different options.

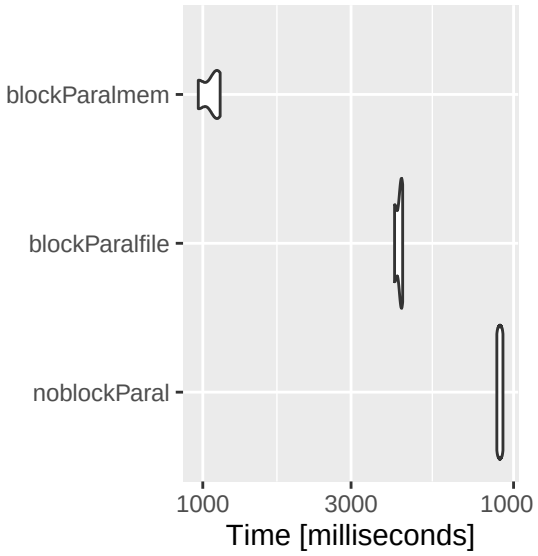
```
bench1 <- microbenchmark(
  noblockParal = blockmult(Abig, Bbig, paral = TRUE),
  blockParalfile = blockmult(Abig, Bbig, block_size = 256, paral=TRUE),
```

```
  blockParalmem = blockmult(Abig, Bbig, block_size = 256, paral=TRUE, bigmatrix = 100000),
  times = 3 ) # Tornar-ho a posar a 10 quan tot OK per estalviar temps d'execució !!!

bench1
Unit: milliseconds
           expr       min       lq     mean   median       uq      max neval cld
   noblockParal 8834.7909 8935.988 9038.554 9037.186 9140.435 9243.684     3   c
 blockParalfile 4141.2553 4232.043 4287.099 4322.831 4360.021 4397.211     3  b
  blockParalmem  967.1867 1026.736 1063.596 1086.286 1111.800 1137.315     3 a
```

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

```
ggplot2::autoplot(bench1)
```



Plot shows that the execution time in hdf5 files is greater than memory execution time, this is due to the read and write time required to access to disk data. At higher disk speed better performance.

6 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
```

We can show that the implemented version really improves the R implementation computational speed.

```
bench <- microbenchmark(
  bdcrossp = bdcrossprod(DA, transposed = TRUE),
  rcrossp = tcrossprod(A),
  times = 1)  # Only DEBUG MODE
  # times = 10) # PRODUCTION MODE
bench
Unit: milliseconds
     expr       min        lq      mean    median        uq       max neval
 bdcrossp  7.553357  7.553357  7.553357  7.553357  7.553357  7.553357     1
  rcrossp 19.440293 19.440293 19.440293 19.440293 19.440293 19.440293     1
```

7 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.

7.1 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,] 44.566643 -1.1443388  2.008108 -1.8534484  2.8048958
[2,] -1.144339 44.7375240  4.331431  0.5470362 -0.5210717
[3,]  2.008108  4.3314306 44.334277 -3.6845822  4.6351765
[4,] -1.853448  0.5470362 -3.684582 35.1744289  0.6967215
[5,]  2.804896 -0.5210717  4.635177  0.6967215 38.6863330
```

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

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

7.2 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,] 42.6182878  0.1713293 -3.4083986 -0.1255156 -0.8229967
[2,]  0.1713293 36.6035590  2.4290292 -0.9855219  0.8330007
[3,] -3.4083986  2.4290292 34.1964223  0.4676414 -0.6890005
[4,] -0.1255156 -0.9855219  0.4676414 40.5834290 -2.0438188
[5,] -0.8229967  0.8330007 -0.6890005 -2.0438188 34.8288107
```

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

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

8 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,]  0.01481736 -0.00308175 -0.00272473 -0.00038127  0.00208968
[2,] -0.00308175  0.01394900  0.00423320  0.00198972  0.00135516
[3,] -0.00272473  0.00423320  0.01292554 -0.00028941 -0.00070686
[4,] -0.00038127  0.00198972 -0.00028941  0.01344189  0.00130492
[5,]  0.00208968  0.00135516 -0.00070686  0.00130492  0.01457558
```

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

```
all.equal(Dinvchol, solve(A))
[1] TRUE
```

9 Singular Value Decomposition (SVD)

The SVD 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.

9.1 Simple Singular Values Decomposition

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
set.seed(413)
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$d[1:5]
[1] 44.27037 43.95609 43.31037 43.01980 42.81728
bsvd$u[1:5,1:5]
             [,1]        [,2]         [,3]         [,4]         [,5]
[1,]  0.028712445  0.10169472  0.019783036  0.032804622 -0.048915416
[2,]  0.001193891  0.02296487  0.009024529  0.059683600  0.027888834
[3,] -0.008878497 -0.02414059 -0.006035909 -0.004867886  0.024570252
[4,]  0.037323916 -0.04675198  0.035928628 -0.021097593 -0.073485962
[5,] -0.054073083  0.04382626 -0.009027050 -0.002269443 -0.003071978
bsvd$v[1:5,1:5]
            [,1]        [,2]        [,3]        [,4]         [,5]
[1,] -0.07105335  0.04554495 -0.03850442  0.07270812  0.078124410
[2,]  0.05989691 -0.02957792 -0.07508850 -0.01171385  0.001581823
[3,] -0.00200688 -0.04179449  0.02798939 -0.05686766 -0.073573833
[4,] -0.01284403 -0.05024617 -0.09038993 -0.05725360  0.023632656
[5,] -0.01751359  0.07320492 -0.02120536 -0.02397367 -0.067643671
```

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 `bcenter = false` and `bscale = false`.

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


bsvd$d[1:5]
[1] 44.44007 43.89640 43.38384 43.23563 42.82658
```

```
bsvd$u[1:5,1:5]
              [,1]        [,2]         [,3]         [,4]        [,5]
[1,]  3.075740e-02  0.09861737 -0.026705796  0.001569226 -0.04384078
[2,]  6.480471e-05  0.04151230 -0.049215808  0.054763599 -0.01436232
[3,] -1.857597e-02 -0.02463540 -0.001246071  0.015838141  0.05725309
[4,]  3.555600e-02 -0.05715730 -0.009596880 -0.040247700 -0.03888504
[5,] -5.290501e-02  0.04784627 -0.006624289  0.010168748  0.01934154
bsvd$v[1:5,1:5]
              [,1]        [,2]         [,3]        [,4]        [,5]
[1,] -0.0677291931  0.03646765 -0.072013679 -0.06460296  0.03834539
[2,]  0.0558086446 -0.02816709 -0.067619533  0.02478961  0.01316491
[3,]  0.0024305441 -0.04679000  0.064115513  0.04134591 -0.03519768
[4,] -0.0007861777 -0.06602592 -0.062833191  0.06307396  0.03651487
[5,] -0.0165396063  0.07316212  0.002600021  0.07030477 -0.07916813

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
```

9.2 Block Singular Values Decomposition

This method was developed in June 2016 by M. A. Iwen and B. W. Ong. The authors describe it as a distributed and incremental SVD algorithm that is useful for agglomerative data analysis on large networks. The algorithm calculates the singular values and left singular vectors of a matrix A, by first, partitioning it by columns. This creates a set of submatrices of A with the same number of rows, but only some of its columns. After that, the SVD of each of the submatrices is computed. The final step consists of combining the results obtained by merging them again and computing the SVD of the resulting matrix.

This method is implemented in `bdSVD_hdf5` function, this function works directly on hdf5 data format, loading in memory only the data to perform calculations and saving the results again in the hdf5 file for later use.

We have to indicate the file to work with, the dataset name and the group under the dataset is located :

```
# Direct from hdf5 data file
svdh5 <- bdSVD_hdf5("delayed.hdf5", "OMICS", "data")
svdh5$d[1:7]
[1] 393.8452 393.2876 392.7751 392.2776 392.1270 391.8639 391.7740

# with R implementation from data in memory
fprova <- H5Fopen("delayed.hdf5")
omdata <- fprova$OMICS$data
h5closeAll()

svd <- svd(scale(omdata))
svd$d[1:7]
[1] 393.8452 393.2876 392.7751 392.2776 392.1270 391.8639 391.7740
```

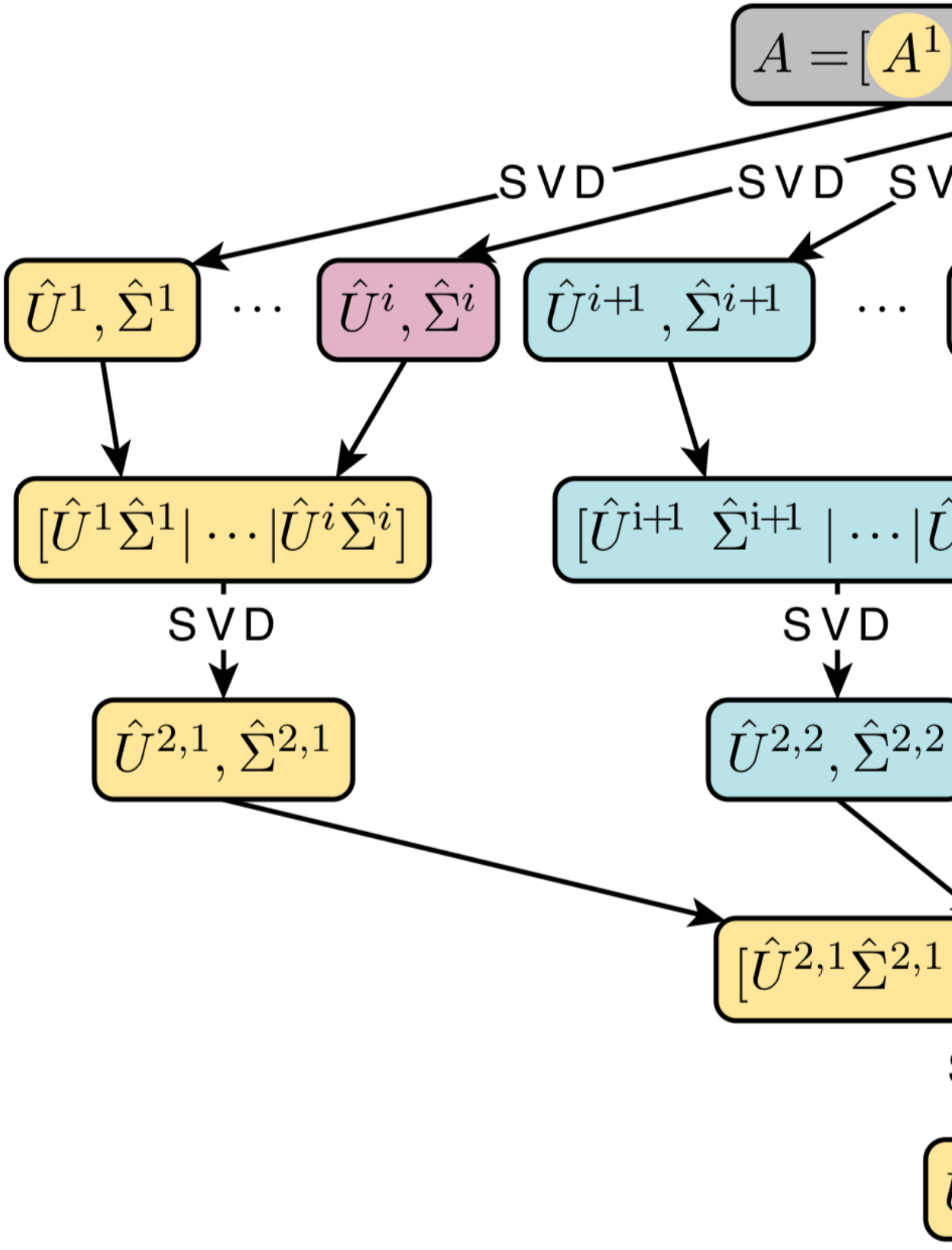


Figure 2: Flowchart for a two-level hierarchical Block SVD algorithm

```
all.equal(svd$d,svdh5$d )
[1] TRUE
```

Like in Simple Singular Value Decomposition we can normalize, center or scale data before proceed with SVD decomposition with `bscale` and `bcenter` parameters, by default this parameter are TRUE, data is normalized before SVD decomposition. To proceed with SVD without normalization :

```
# Direct from hdf5 data file
svdh5 <- bdSVD_hdf5("delayed.hdf5", "OMICS", "data", bcenter = FALSE, bscale = FALSE)
svdh5$d[1:7]
[1] 394.0060 393.7752 392.8768 392.4745 392.1387 391.9469 391.5861

# with R implementation from data in memory
svd <- svd(omdata)
svd$d[1:7]
[1] 394.0060 393.7752 392.8768 392.4745 392.1387 391.9469 391.5861

all.equal(svd$d,svdh5$d )
[1] TRUE
```

In the SVD decomposition by blocks we can indicate the number of decomposition levels and number of local SVDs to concatenate at each level with parameters `q` and `k` respectively, by default `q = 1` one level with `k=2`.

```
# Block decomposition with 1 level and 4 local SVDs at each level
svdh5 <- bdSVD_hdf5("delayed.hdf5", "OMICS", "data", q=1, k=4 )
svdh5$d[1:7]
[1] 393.8452 393.2876 392.7751 392.2776 392.1270 391.8639 391.7740

# with R implementation from data in memory
svd <- svd(scale(omdata))
svd$d[1:7]
[1] 393.8452 393.2876 392.7751 392.2776 392.1270 391.8639 391.7740

all.equal(svd$d,svdh5$d )
[1] TRUE
```

We can show performance for distinct SVD functions.

SVD decomposition with normalized data and different parameters executed in memmory and hdf5 files with functions from R base library and BigDataStatMeth library :

```
bdSVD_hdf5_Normalized_k2q1 <- "bdSVD_hdf5('delayed.hdf5',group='OMICS',dataset='data')"
bdSVD_hdf5_Normalized_k4q1 <- "bdSVD_hdf5('delayed.hdf5',group='OMICS',dataset='data',k=4)"
bdSVD_hdf5_Normalized_k4q2 <- "bdSVD_hdf5('delayed.hdf5',group='OMICS',dataset='data',q=2,k=4)"
bdSVD_memory_Normalized <- 'bdSVD_lapack(omdata)'
svd_R_memory_Normalized <- 'svd(scale(omdata))'

# With normalization process
res <- microbenchmark( eval(parse(text=bdSVD_hdf5_Normalized_k2q1)),
                       eval(parse(text=bdSVD_hdf5_Normalized_k4q1)),
```

```
                          eval(parse(text=bdSVD_hdf5_Normalized_k4q2)),
                          eval(parse(text=bdSVD_memory_Normalized)),
                          eval(parse(text=svd_R_memory_Normalized)),
                          times = 3, unit = "s")

print(summary(res)[, c(1:7)],digits=3)
                                          expr  min   lq mean median   uq  max
1 eval(parse(text = bdSVD_hdf5_Normalized_k2q1)) 2.01 2.07 2.15   2.12 2.22 2.33
2 eval(parse(text = bdSVD_hdf5_Normalized_k4q1)) 1.89 1.91 1.96   1.94 1.99 2.05
3 eval(parse(text = bdSVD_hdf5_Normalized_k4q2)) 1.56 1.60 2.01   1.63 2.24 2.84
4    eval(parse(text = bdSVD_memory_Normalized)) 1.50 1.51 1.53   1.52 1.54 1.56
5    eval(parse(text = svd_R_memory_Normalized)) 2.60 2.67 2.79   2.75 2.89 3.02
```

SVD decomposition with not normalized data and different parameters executed in memmory and hdf5 files with functions from R base library and BigDataStatMeth library :

```
bdSVD_hdf5_Not_Normalized_k2q1 <- "bdSVD_hdf5('delayed.hdf5',group='OMICS',dataset='data',bcenter=FALSE,bsca
bdSVD_hdf5_Not_Normalized_k4q1 <- "bdSVD_hdf5('delayed.hdf5',group='OMICS',dataset='data',k=4,bcenter=FALSE,
bdSVD_hdf5_Not_Normalized_k4q2 <- "bdSVD_hdf5('delayed.hdf5',group='OMICS',dataset='data',q=2,k=4,bcenter=FA
bdSVD_memory_Not_Normalized <- 'bdSVD_lapack(omdata, bcenter = TRUE, bscale = TRUE)'
svd_R_memory_Not_Normalized <- 'svd(omdata)'

# Without normalization process
res <- microbenchmark( eval(parse(text=bdSVD_hdf5_Not_Normalized_k2q1)),
                       eval(parse(text=bdSVD_hdf5_Not_Normalized_k4q1)),
                       eval(parse(text=bdSVD_hdf5_Not_Normalized_k4q2)),
                       eval(parse(text=bdSVD_memory_Not_Normalized)),
                       eval(parse(text=svd_R_memory_Not_Normalized)),
                       times = 3, unit = "s")

print(summary(res)[, c(1:7)],digits=3)
                                              expr  min   lq mean median   uq  max
1 eval(parse(text = bdSVD_hdf5_Not_Normalized_k2q1)) 1.87 1.92 1.99   1.98 2.05 2.12
2 eval(parse(text = bdSVD_hdf5_Not_Normalized_k4q1)) 1.90 1.92 1.94   1.93 1.96 1.98
3 eval(parse(text = bdSVD_hdf5_Not_Normalized_k4q2)) 1.44 1.47 1.54   1.49 1.59 1.69
4    eval(parse(text = bdSVD_memory_Not_Normalized)) 1.50 1.50 1.52   1.50 1.53 1.57
5    eval(parse(text = svd_R_memory_Not_Normalized)) 1.47 1.47 1.49   1.47 1.50 1.54
```

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

10.1 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:

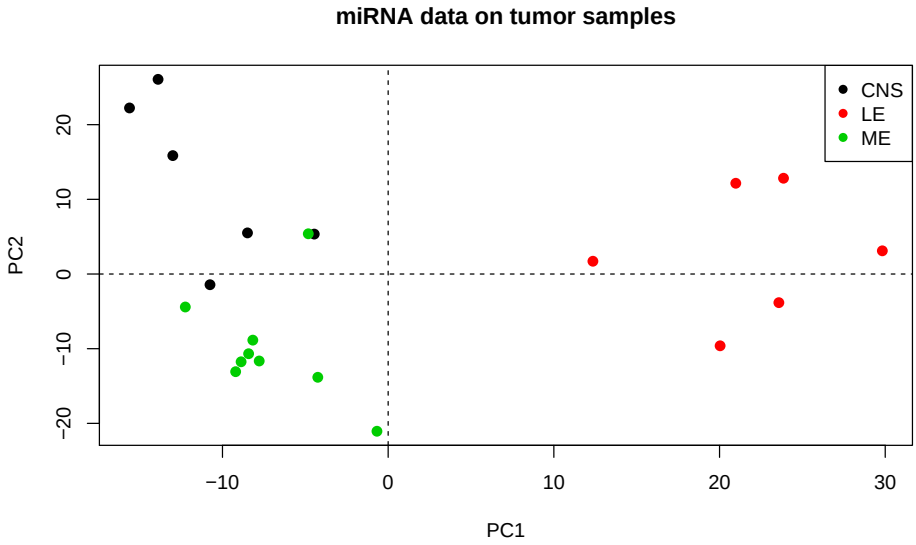
```
table(cancer)
cancer
CNS  LE  ME
  6   6   9
```

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)
```

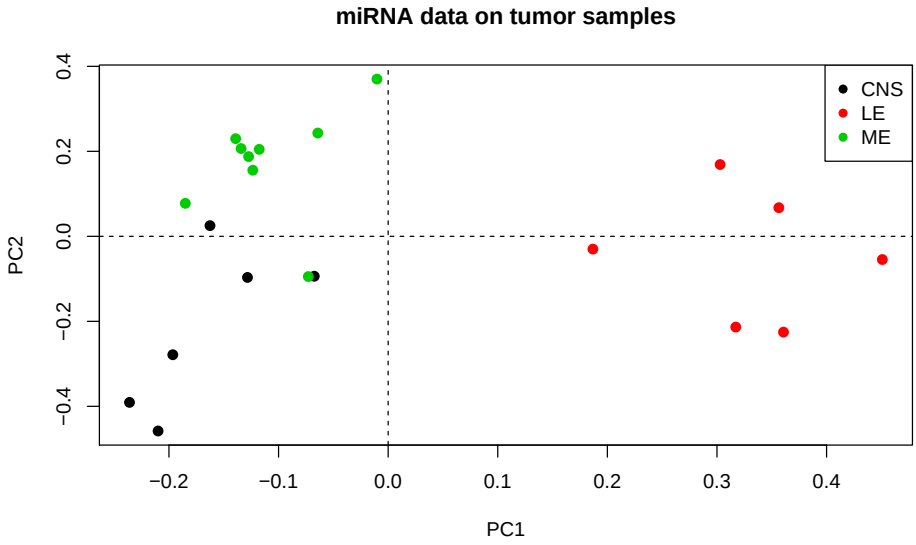


The same analysis can be performed using SVD decomposition and having miRNAs as a `DelayedMatrix` object. The PCA is equivalent to performing the SVD on the centered data, where the centering occurs on the columns. In that case the function `bdSVD` requires to set the argument `bcenter` and `bscale` equal to TRUE, the dafault values. 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, bcenter = FALSE, bscale = 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).

11 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.

11.1 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), 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]
```

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)
mod$coef[abs(mod$coef)>mean(mod$coef)]
[1]  2.4021139  1.6004763 -0.3989715
```

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[1:7,]
        V1         V2         V3         V4         V5         V6         V7
 2.3481950  1.5510646  0.0000000  0.0000000 -0.3517855  0.0000000  0.0000000
```

12 Utils for data analysis working directly in hdf5 files

12.1 SNP imputation

Imputation in genetics refers to the statistical inference of unobserved genotypes. In genetic epidemiology and quantitative genetics, researchers aim at identifying genomic locations where variation between individuals is associated with variation in traits of interest between individuals.

At the moment BigDataStatMeth has implemented snps imputation following the encoding used in Genomic Data Structure file format (GDS files) where SNPs are codded as 0 : two B alleles, 1 : one A allele and one B allele, 2 : two A alleles and 3 : missing data. BigDataStatMeth impute data where SNPs values are '3' (missing data).

The imputation method implemented is generated from the information of the SNP data that contains the missing data, it is performed by generating a random value following a discrete distribution. If in the SNP we find 70% of '0', 25% of '1' and 5% of '2', the value of the missing data will be imputed with a probability of 70% to be '0', 25% to be '1' and 5% to be '2'.

We first simulate a genotype matrix with 0, 1, 2 and 3 and store it in hdf5 file and show data stored in file:

```
set.seed(108432)
geno.sim <- matrix(sample(0:3, 10000, replace = TRUE), byrow = TRUE, ncol = 10)
Create_HDF5_matrix(geno.sim, "delayed.hdf5", "OMICS", "geno")
[1] 0

# Get data and show the first 5 rows
h5fsvd = H5Fopen("delayed.hdf5")
geno <- h5fsvd$OMICS$geno
h5closeAll()

geno[1:5,1:10]
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
[1,]    0    2    1    3    3    2    3    1    0     3
[2,]    2    3    2    2    1    3    3    0    2     3
[3,]    0    0    1    2    2    1    2    0    0     1
[4,]    2    0    1    1    3    0    1    1    3     3
[5,]    3    0    1    3    3    0    0    1    1     3
```

Remember, we allways have to close hdf5 data file. Now we impute values where genotype = '3' (missing data in GDS format). To apply imputation we only have to indicate the file where dataset is, the group and the dataset name, optionally, we indicate were we want to store results with `outputgroup` and `outputfolder`, by default the results are stored in the input dataset. An important parameter is `bycols`, with `bycols` we can inform if we want to impute data by cols `bycols = TRUE` or by rows `bycols = FALSE`, the default imputation implemented method is by cols.

In next example we perform the imputation in "imputedgeno" dataset inside the group "OMICS" and the imputation method applied takes into account the probabilities in columns (default).

```
bdImputeSNPHDF5("delayed.hdf5", group="OMICS", dataset="geno", outgroup="OMICS", outdataset="imputedgeno")
[1] 0
```

Now we get the imputation results

```
# Get imputed data and show the first 5 rows
h5fsvd = H5Fopen("delayed.hdf5")
imputedgeno <- h5fsvd$OMICS$imputedgeno
h5closeAll()

imputedgeno[1:5,1:10]
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
[1,]    0    2    1    2    0    2    0    1    0     0
[2,]    2    1    2    2    1    1    2    0    2     0
[3,]    0    0    1    2    2    1    2    0    0     1
[4,]    2    0    1    1    0    0    1    1    1     1
[5,]    1    0    1    1    1    0    0    1    1     0
```

12.2 Normalization, center and scale

Normalization is one of the most important procedures in genomics data analysis. A typical dataset contains more than one sample and we are almost always interested in making comparisons between these and in order to make comparisons we need to normalize data.

In BigDataStatMeth we implemented a normalization method that works directly with datasets stored in hdf5 file format, like in other functions implemented in BigDataSet that works with hdf5 data files, we have to indicate where the data is, the filename, the name of the group under the dataset is and the dataset name. In normalization function, we can indicate if we want to center data, scale or center and scale (default option). The applied formula to normalize data by default in implemented function is :

$$\frac{X - \mu}{\sigma}$$

The implemented function to normalize data directo from hdf5 data file is `Normalize_hdf5`.

To show you an example, we will use the imputed geno dataset *imputedgeno* created before in imputation method example. To normalize data we have to indicate where the dataset is stored and we have two optiona paramters `bcenter` and `bscale` with default value = TRUE.

In order to normalize data, we don't need to modify these optional parameters. If we only want to center data, then we have to put `bscale = FALSE` or if we only want to scale data `bcenter` parameter must be FALSE.

Important : The normalization results are stored under "NORMALIZED" group, under this group, we found all normalized datasets with the same structure under root, for example, if we normalize dataset genoimputed under OMICS group, the normalized results will be in "NORMALIZED"/"OMICS"/genoimputed

Let show an example normalizing imputed geno under group "OMICS" in delayed.hdf5 file:

```
Normalize_hdf5("delayed.hdf5", group="OMICS", dataset="imputedgeno")
[1] 0
```

The results will be under groups "NORMALIZED" - "OMICS" in dataset genoimputed

```
# Geno after normalization
h5fsvd = H5Fopen("delayed.hdf5")
genonormalized <- h5fsvd$NORMALIZED$OMICS$geno
h5closeAll()

genonormalized[1:5,1:10]
NULL
```

12.3 QC - Remove low data

Missing data in meta-analysis of genetic association studies are unavoidable. In BigDataStat-Meth we implemented a function to remove those SNPs with a hight percentage of missing data. The implemented function to remove SNPs wit a certain percentage of missing data is the function `bdRemovelowdata`

Like in imputation method, in remove low data method we have to inform the input data indicating the filename, the group and the dataset name and we also have to inform the dataset were data will be stored after removing the SNPs with parameters `outgroup` and `outdataset`.

To remove low data we have to infrorm if we want to remove data by cols `SNPincols = TRUE` or by rows `SNPincols = FALSE` and the percentage of missing data `pcent` parameter. `pcent` of missing data refers to missing percentage in columns if we defined `SNPincols = TRUE` or `pcent` refers to rows if `SNPincols = FALSE`.

To show how renove low data works we use the previous dataset stored in *delayed.hdf5* file, in that case, we will assume that SNPs are in columns and we will remove those SNPs where missing data is greater than 40%.

```
bdRemovelowdata("delayed.hdf5", group="OMICS", dataset="geno", outgroup="OMICS", outdataset="removedgeno", SN

 La entrada i sortida no seran iguals
Warning in evalq((function (..., call. = TRUE, immediate. = FALSE, noBreaks. = FALSE, : -235 Columns have bee
[1] 0
```

After remove the SNPs with pcent greater than 39% results have been saved in dataset "*removedgeno*" under group "*OMICS*", we observe the resulting dataset

```
# Get imputed data and show the first 5 rows
h5fsvd = H5Fopen("delayed.hdf5")
removedgeno <- h5fsvd$OMICS$removedgeno
h5closeAll()

removedgeno[1:5,1:10]
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
[1,]    0    2    1    3    2    3    1    0    1     2
[2,]    2    3    2    2    3    3    0    2    1     2
[3,]    0    0    1    2    1    2    0    0    1     3
[4,]    2    0    1    1    0    1    1    3    1     1
[5,]    3    0    1    3    0    0    1    1    3     2
```

13 References

[1] Fox L., Mayers D.F. (1987). Algorithms. In: Numerical Solution of Ordinary Differential Equations. (198-211) Springer, Dordrecht.

14 Session information

```
sessionInfo()
R version 3.6.3 (2020-02-29)
Platform: x86_64-apple-darwin15.6.0 (64-bit)
Running under: macOS Mojave 10.14.4

Matrix products: default
BLAS:   /Library/Frameworks/R.framework/Versions/3.6/Resources/lib/libRblas.0.dylib
LAPACK: /Library/Frameworks/R.framework/Versions/3.6/Resources/lib/libRlapack.dylib

locale:
[1] ca_ES.UTF-8/ca_ES.UTF-8/ca_ES.UTF-8/C/ca_ES.UTF-8/ca_ES.UTF-8

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

other attached packages:
 [1] glmnet_4.0-2         Matrix_1.2-18        microbenchmark_1.4-7 DT_0.15              rhdf5_2.30.1
 [9] S4Vectors_0.24.4     BiocGenerics_0.32.0  matrixStats_0.56.0   BigDataStatMeth_1.1  BiocStyle_2.14.4

loaded via a namespace (and not attached):
 [1] Rcpp_1.0.5          mvtnorm_1.1-1       lattice_0.20-41     png_0.1-7           zoo_1.8-8           fore
 [9] evaluate_0.14       ggplot2_3.3.2       pillar_1.4.6        rlang_0.4.7         multcomp_1.4-13     rma
[17] RcppEigen_0.3.3.7.0 htmlwidgets_1.5.1   munsell_0.5.0       beachmat_2.2.1      compiler_3.6.3      xfu
[25] htmltools_0.5.0     tidyselect_1.1.0    tibble_3.0.3        bookdown_0.20       codetools_0.2-16    cra
[33] grid_3.6.3          gtable_0.3.0        lifecycle_0.2.0     magrittr_1.5        scales_1.1.1        Rcp
```

```
[41] ellipsis_0.3.1      vctrs_0.3.2         generics_0.0.2      sandwich_2.5-1      TH.data_1.0-10      Rhdf
[49] glue_1.4.1          purrr_0.3.4         jpeg_0.1-8.1        survival_3.2-3      yaml_2.2.1          colo
```

Fischer, Bernd, Gregoire Pau, and Mike Smith. 2019. *Rhdf5: R Interface to Hdf5*. https://github.com/grimbough/rhdf5.