Singular Value Decomposition for Federated Data

Claudia Serrano [1,2,3] ***and Juan R. Gonzalez*** [*1,2,3]

[1]Autonomous University of Barcelona (UAB)
[2]ISGlobal - Barcelona Institute for Global Health
[3]Bioinformatics Research Group in Epidemiolgy (BRGE)

[*]`juanr.gonzalez@isglobal.org`

2019-06-25

Package

svdParallel 0.1.0

Contents

1 Purpose of this package. 2

2 Getting started 2

3 Computing a PCA using Jacobi algorithm 3

4 Computing a PCA using Block method for DataSHIELD. 6

5 Computing a PCA using Block method for omic data. 10

References 17

1 Purpose of this package

Analyzing data across hospitals and institutions without the data leaving the hospitals and adding institutions to a trusted network is an important part of privacy preserving data analysis. In order to assure all of this we can use DataSHIELD, an open source solution that let researchers analyze microdata without physically sharing this data with them. It allows a standardized, monitored, indirect and secure access to data.

The aim of this package is to provide functions that compute the Singular Value Decomposition of large matrices and perform a Principal Component Analysis when we do not have access to the whole dataset, as in DataSHIELD [1]. At the same time, we are interested in the high computational cost that computing a SVD has, so we present some functions that calculate SVD's and PCA's in a more efficient way. The functions in this package can be classified in two different blocks: Jacobi algorithm and Block algorithm. The first group contains functions that perform a SVD using functions that apply the two sided Jacobi algorithm. This is an iterative algorithm that is also parallelizable and could perform well with large datasets and multi cores processors. However, it cannot be applied to DataSHIELD. The second group uses an incremental algorithm that we call Block method and consists of partitioning the matrix under study in submatrices in order to reduce their dimension and compute the SVD faster. This is applicable to DataSHIELD.

[1]http://www.datashield.ac.uk/

The Block method allows us to compute a PCA of different datasets without being able to join them. However, to do so we need that each dataset contains the same variables and a greater number of individuals than variables. This usually happens in most of the datasets. Nevertheless this is not the case for omic data. In this case, Block functions will not be useful for DataSHIELD but they will be good to perform efficient calculations in large datasets.

2 Getting started

First, you can install `svdParallel` from Github by typing

```r
devtools::install_github("isglobal-brge/svdParallel")
```

Then, you can load the required packages to reproduce this document.

```r
library(svdParallel)
library(FactoMineR)
library(irlba)
library(doParallel)
library(devtools)
library(Biobase)
library(MultiDataSet)
library(made4)
library(brgedata)
library(parallel)
library(SummarizedExperiment)
library(microbenchmark)
library(BiocStyle)
```

In the following sections of this document we will present some examples in order to make clear how the different functions work. To do so we will use three datasets: `iris`, `breast` and `breastTCGA`. The `iris` dataset is an R dataset and the other two, `breast` and `breastTCGA`, are saved in the data folder of this project, so you can access and manipulate all of them.

3 Computing a PCA using Jacobi algorithm

In this case we will use the `iris` dataset. We can see the head of this dataset as follows

```
head(iris)
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1          5.1         3.5          1.4         0.2  setosa
2          4.9         3.0          1.4         0.2  setosa
3          4.7         3.2          1.3         0.2  setosa
4          4.6         3.1          1.5         0.2  setosa
5          5.0         3.6          1.4         0.2  setosa
6          5.4         3.9          1.7         0.4  setosa
```

We need to remove the last column because it is a categorical variable that we are not able to use in a PCA. We will use it later to analyze the results.

```
head(iris$Species)
[1] setosa setosa setosa setosa setosa setosa
Levels: setosa versicolor virginica

i <- iris[,-5]
```

The function that performs a PCA is `getPCA` and we have to set the method that we want to use to compute this PCA. The default method is the Block method, so if we want to compute the PCA using the Jacobi method we need to specify it. After computing a PCA with `getPCA` we can use function `print` to see the main values obtained with the PCA, the variance of each component and the percentage of variance explained.

```
gJ <- getPCA(i,method="Jacobi")
print(gJ)
**Results for the Principal Component Analysis using method 'Jacobi' **

Variance of the components
          PC1       PC2       PC3        PC4
[1,] 2.918498 0.9140305 0.1467569 0.02071484

Percentatge of variance explained for each component
          PC1      PC2      PC3       PC4
[1,] 72.96245 22.85076 3.668922 0.5178709
```

Now, we can plot the results using function `plot`. We need to specify what type of plot we want, individuals or variables. Moreover, if we do not specify anything more the plot will be made in the first two principal components.

```
par(mfrow=c(1,2))
plot(gJ, type="variables")
plot(gJ, type="individuals")
```

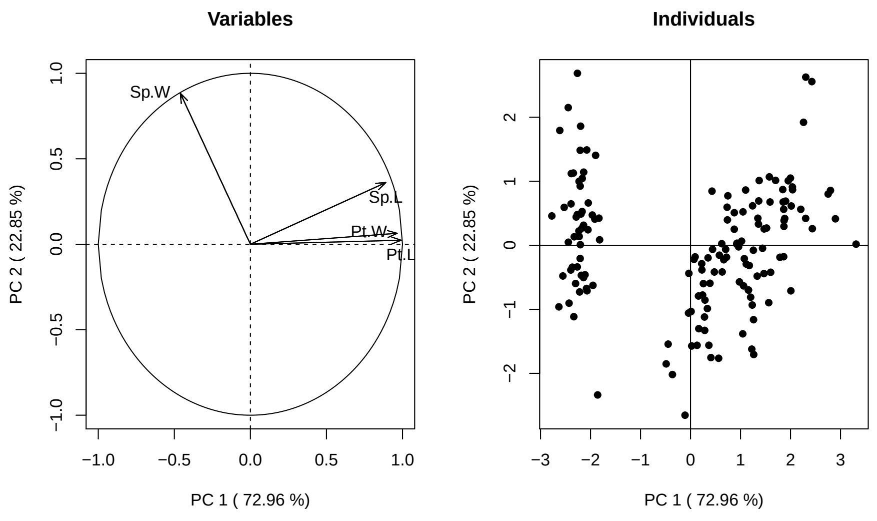

Figure 1: Variables and individuals projections from getPCA function using Jacobi method
Left plot shows the variables in to the two first principal components, while right plot corresponds to the individuals.

If we want the plots in other components we can do it by adding `comps=c(i,j)`, where i and j are the new components.

```r
par(mfrow=c(1,2))
plot(gJ, type="variables", comps=c(1,3))
plot(gJ, type="individuals", comps=c(1,3))
```

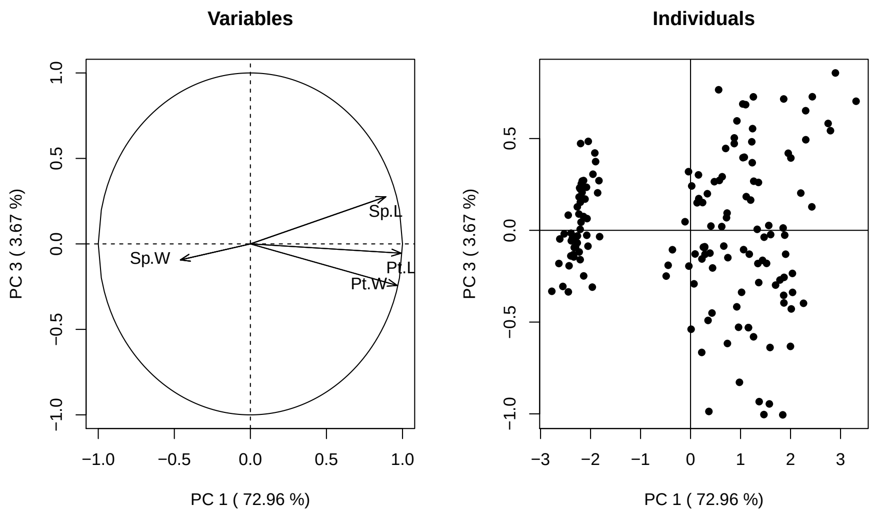

Figure 2: Variables and individuals projections from getPCA function using Jacobi method
Left plot shows the variables into the first and third first principal components, while right plot corresponds to the individuals.

The `getPCA` function returns a list of different variables that we can use to perform a deeper analysis of the data. This variables are the coordinates of the variables, the Y matrix containing the coordinates of the individuals, the variance and percentage of variance explained by each component, the V matrix of the SVD performed and the method used.

In order to access to this variables we can do

```
gJ$varcoord
                    PC1        PC2         PC3         PC4
Sepal.Length  0.8901688 0.36082989  0.27565767 -0.03760602
Sepal.Width  -0.4601427 0.88271627 -0.09361987  0.01777631
Petal.Length  0.9915552 0.02341519 -0.05444699  0.11534978
Petal.Width   0.9649790 0.06399985 -0.24298265 -0.07535950
```

For instance, we can use the Y matrix with the coordinates of the individuals to represent them according to the species they belong:

```
setosa <- iris$Species == "setosa"
versicolor <- iris$Species == "versicolor"
virginica <- iris$Species == "virginica"
plot(gJ$Y[setosa,c(1,2)], main="Individuals",
     type="p", col="blue",xlab="PC1 (72.96%)",
     ylab="PC2 (22.85%)", xlim=c(-4,5.5))
points(gJ$Y[versicolor,c(1,2)], col="red")
points(gJ$Y[virginica,c(1,2)], col="green")
legend("bottomright",
       c("Setosa", "Versicolor", "Virginica"),
       col=c("blue", "red", "green"), pch=1,
       cex=0.75)
```

As we can see, the individuals are classified according to the species.

we can perform the same calculations using `JacobiR` method. This method is not parallelizable and it is faster than `Jacobi`. We would recommend using `JacobiR` when we do not deal with big matrices and we do not need to parallelize. However, in small matrices the difference is not significant.

The functions that contain the Jacobi algorithm are `Jacobi`, `JacobiR` and `parJacobi` and they compute the eigenvalues and eigenvectores of a real symmetric matrix using the two-sided Jacobi algorithm. Functions `svdJacobi` and `svdJacobiR` compute a SVD of a real rectangular matrix and returns the left and right singular vectors and the singular values.

```
svdJ <- svdJacobi(i)
svdJ$d
Sepal.Length  Sepal.Width Petal.Length  Petal.Width
   95.959914    17.761034     3.460931     1.884826
```

We have obtained the singular values of the iris dataset.

As these functions use an iterative method we need to set a tolerance. The value set by default is the tolerance of the machine, but it can be changed setting the `tol` parameter.

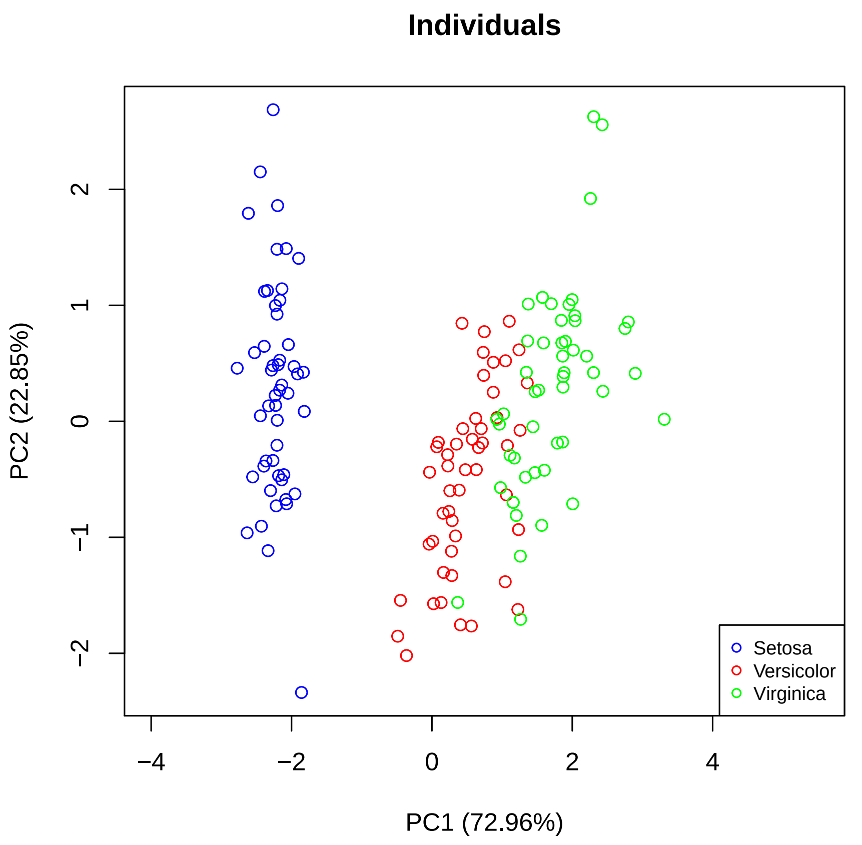

Figure 3: Principal components of iris dataset
The plot shows the samples according to the species they belong to.

4 Computing a PCA using Block method for DataSHIELD.

In this case we will also use `getPCA` function but we will specify a different method. If we want to simulate that we are in a DataSHIELD experiment and we cannot join the different datasets we can use a list containing the datasets. To do so, we will use `breast` dataset. We need to remove the categorical variables to be able to work with the numerical ones.

```
breast[1:5,1:3]
        id diagnosis radius_mean
1   842302         M       17.99
2   842517         M       20.57
3 84300903         M       19.69
4 84348301         M       11.42
5 84358402         M       20.29
num_breast <- breast[,3:32]
dim(num_breast)
[1] 569  30
```

As we can see, the dimension of the dataset is 569 x 29. We can create three datasets taking the first 200 individuals, the individuals in rows between 201 and 400 and the last individuals from 401 to 569. These datasets have the same variables and have more individuals than variables. We can use the Block method to compute the SVD. This method computes a SVD of each of the subsets and then joins the results to compute a SVD again. In the following figure we can see a flowchart of the procedure:

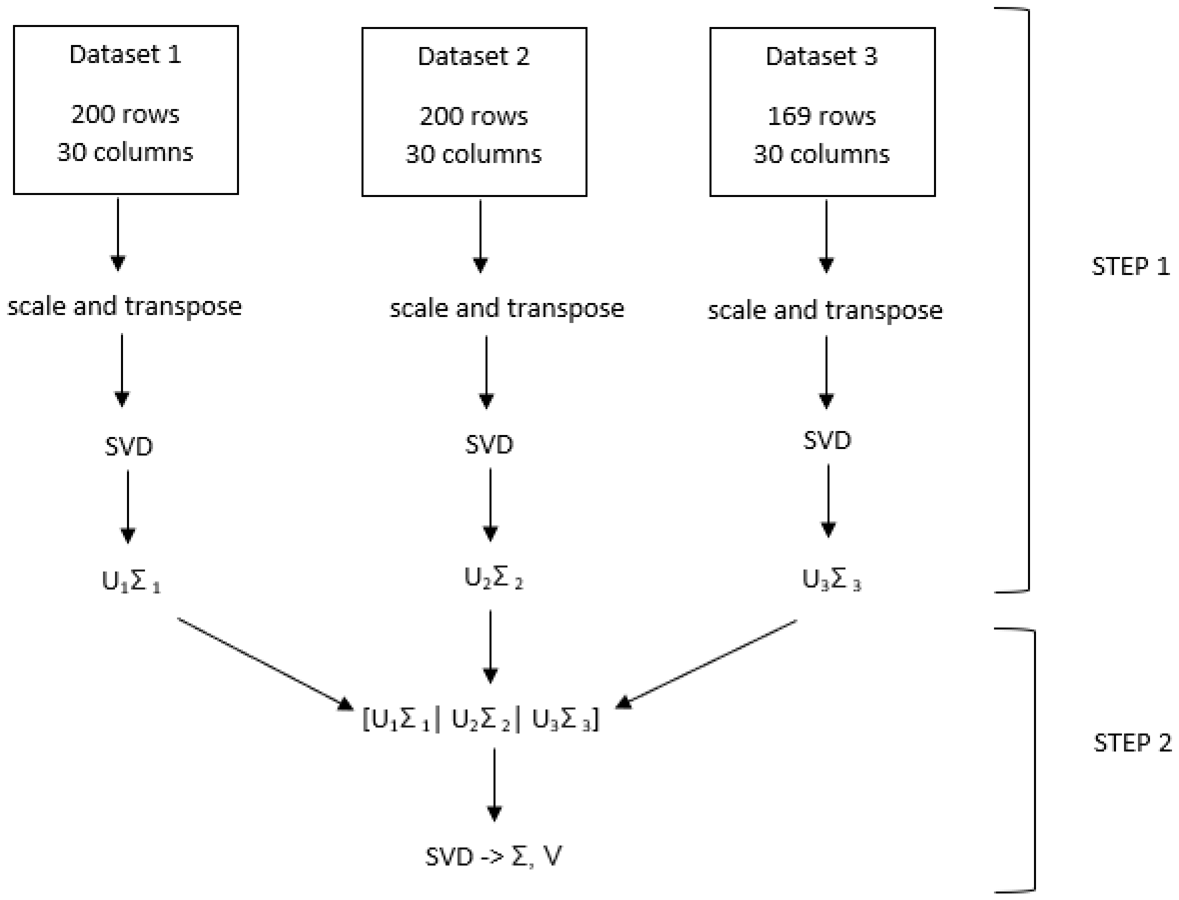

Figure 4: Flowchart of the Block method procedure
The SVD of each of the subdatasets is computed and after that, the results are joined to compute a SVD again.

To do so, we create a list containing these three datasets and we use `getPCA` function. In this case, we do not need to specify the method because the Block method is by default. It is also important that the `getPCA` function has the option to center and scale the data. By default the function will do it, but we can choose not do so.

```r
data1 <- num_breast[1:200,]
data2 <- num_breast[201:400,]
data3 <- num_breast[401:569,]

data_l <- list(data1,data2,data3)
```

```
gpcaL <- getPCA(data_l)
```

Now we can plot the variables and the individuals again

```
par(mfrow=c(1,2))
plot(gpcaL)
plot(gpcaL, type="individuals")
```

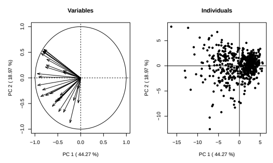

Figure 5: Variables and individuals projections from getPCA function using Block method
Left plot shows the variables into the two first principal components, while right plot corresponds to the individuals.

We can compare the results obtained with function getPCA with other functions in other R packages. IN the following plots we can see the results using `PCA` function in FactoMineR package.

```
pcaR <- PCA(num_breast, graph=FALSE)
par(mfrow=c(1,2))
plot(pcaR, choix="var", label="none")
plot(pcaR, choix="ind", label="none")
```

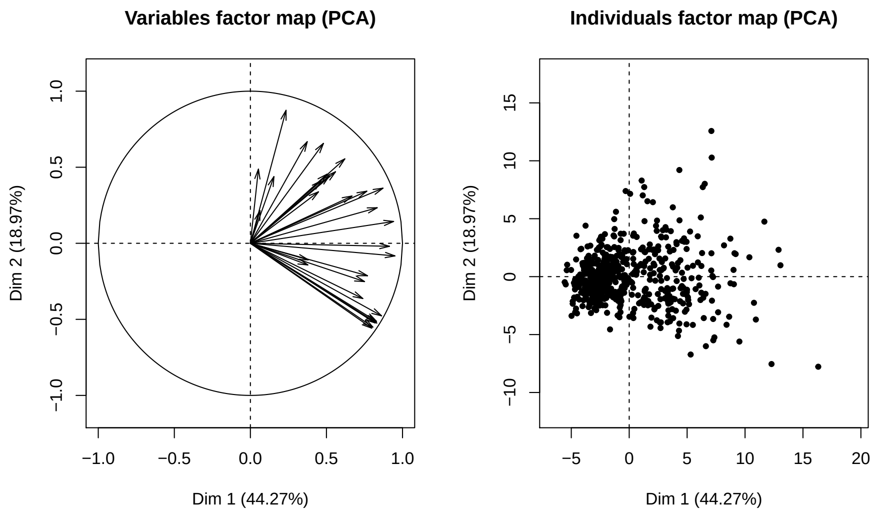

Figure 6: Variables and individuals projections from PCA function
Left plot shows the variables into the two first principal components, while right plot corresponds to the individuals.

The results do not match exactly. However, we can appreciate that these two plots are the two previous ones rotated 180º. This means that they are the same plots but using the opposite direction of the principal components. As we know, the principal components are not uniquely defined as we can take `u` or `-u` and they both would be a principal component. Therefore, the results are coherent.

Now we can also classify them according to the diagnosis of the tumor: benign or malignant.

```
m <- breast[,2] =="M"
b <- breast[,2] =="B"
plot(gpcaL$Y[m,c(1,2)], main="Individuals",
     type="p", col="blue",xlab="PC1 (44.27%)",
     ylab="PC2(18.97%)", xlim=c(-18,12),
     ylim=c(-14,9))
points(gpcaL$Y[b,c(1,2)], col="red")
abline(a=0,b=0, lty=3)
abline(v=0, lty=3)
legend("topright", c("Malignant", "Benign"),
       col=c("blue", "red"), pch=1, cex=0.75)
```

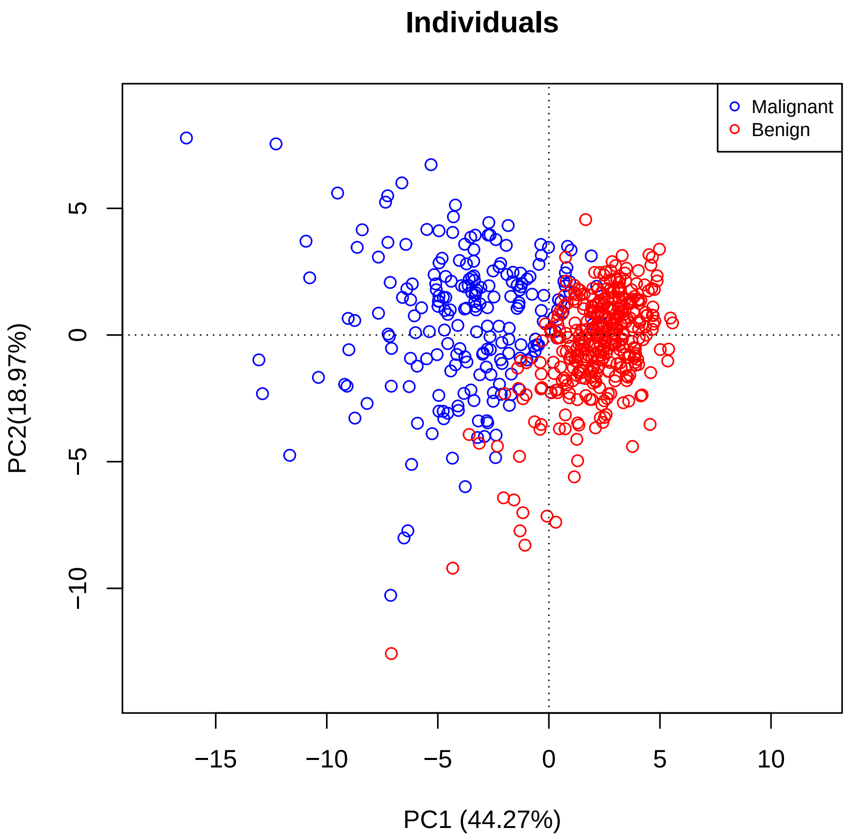

Figure 7: Principal components of breast dataset
The plot shows the samples according to the group they belong (benign or malignant cancer)

We have been able to compute a PCA of three datasets together without merging them. This means that this function would be applicable to DataSHIELD experiments.

5 Computing a PCA using Block method for omic data.

In this case we will use a larger dataset with omic data. We will use a SummarizedExperiment called `breastMulti` that contains a RangeSummarizedExperiment called `breastTCGA`.

```
breastTCGA <- breastMulti[["expression"]]
breastTCGA
class: RangedSummarizedExperiment
dim: 8362 79
metadata(0):
assays(1): genexpr
rownames(8362): ABCA6 ABCC12 ... ZNF569 ZNF610
rowData names(0):
colnames(79): TCGA-C8-A12V TCGA-A2-A0ST ... TCGA-B6-A0X0
  TCGA-A2-A04Y
colData names(30): Gender Age.at.Initial.Pathologic.Diagnosis ...
  Integrated.Clusters..unsup.exp. id
```

We can extract the dataset containing the genes information using `assay` function. There are 8362 variables and 79 individuals.

```
genes <- assay(breastTCGA)
dim(genes)
[1] 8362   79
genes[1:5,1:12]
        TCGA-C8-A12V TCGA-A2-A0ST TCGA-E2-A159 TCGA-BH-A0BW TCGA-A2-A0SX
ABCA6       0.804100     1.411900     0.434300    -1.437600     0.907800
ABCC12     -0.653467    -0.411000    -0.499267    -0.939917    -0.622000
ABCC5       0.062833    -0.837000    -0.531000    -0.758667     0.034833
ABCG8       0.125750    -0.058250    -0.503250    -1.636250     0.003750
ABHD14B    -0.915313    -0.183188    -0.533312    -0.846313    -0.195687
        TCGA-AR-A1AI TCGA-AR-A1AQ TCGA-A1-A0SO TCGA-AO-A0J6 TCGA-BH-A0B3
ABCA6      -0.280100    -0.084200    -0.893300    -1.996800    -0.563900
ABCC12      3.557667    -0.359833    -0.471167    -0.179267    -0.758500
ABCC5       0.731417    -1.078000    -0.435167     0.645667    -0.285167
ABCG8       0.343750     0.559750     0.240750    -0.891750    -0.852750
ABHD14B    -0.816063    -0.178312    -1.000063    -0.383188    -0.440437
        TCGA-C8-A134 TCGA-A2-A0T2
ABCA6      -0.075100    -1.545900
ABCC12     -0.321333    -0.616667
ABCC5      -0.591000    -0.384500
ABCG8      -0.456750     0.088750
ABHD14B    -0.717438    -0.059437
```

As this dataset contains omic data the individuals are in columns and the variables are in rows. This is not the normal way to build datasets and our functions require the individuals to be in the rows and the variables in the columns. This means that we need to apply `getPCA` to the transpose of the `genes` dataset. Moreover, the Block method has a parameter called `parts`. This parameter sets the number of subdatasets to be created from the dataset under study. The default value is 2. If n is the number of rows and p is the number of columns of our dataset, this number `parts` cannot be greater than max(n,p). We can write to set `parts=5`.

```
gpca <- getPCA(t(genes), parts = 5, center=T, scale=T)
plot(gpca, type="individuals")
```

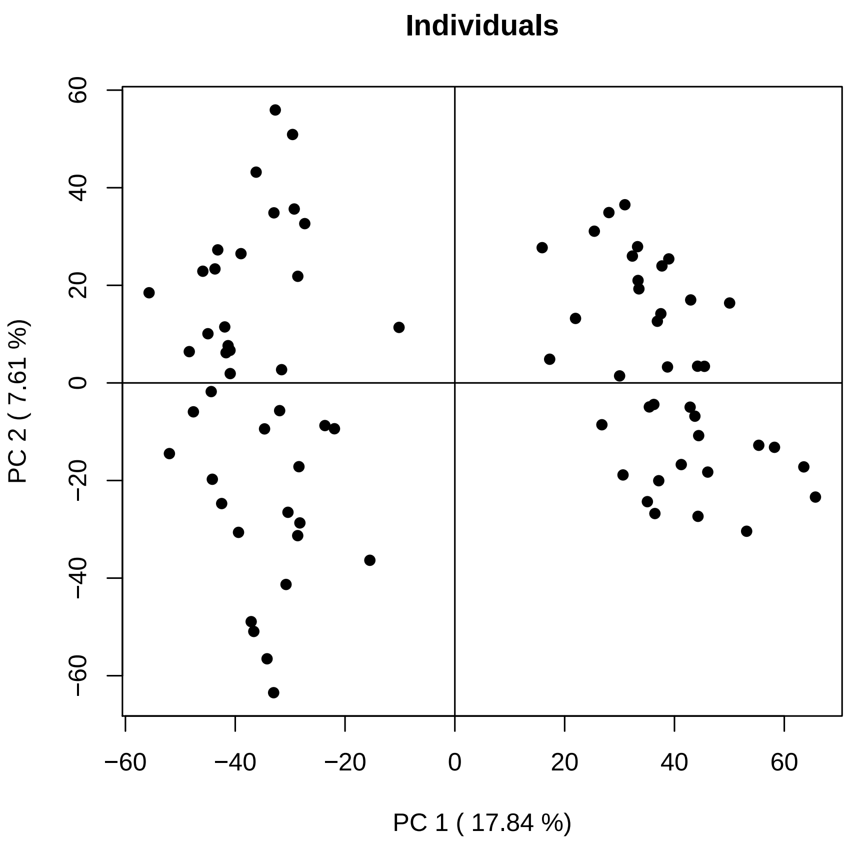

Figure 8: Individuals projections from getPCA function using Block method
The plot shows the individuals into the two first principal components

In this case we will not plot the variables because there are too many and the resulting plot is not interpretable.

In the individuals plot we can see two differentiated groups. These groups correspond to the estrongen status, that can be positive or negative.

```
group<-as.factor(breastTCGA$ER.Status)
plot(gpca$Y[group=="Positive",c(1,2)],
     main="Individuals", type="p",
     col="blue",xlab="PC1 (17.84%)",
     ylab="PC2 (7.61%)", xlim=c(-60,70),
     ylim=c(-70,60))
points(gpca$Y[group=="Negative",c(1,2)], col="red")
```

```
abline(a=0,b=0, lty=3)
abline(v=0, lty=3)
legend("topright", c("Positve", "Negative"),
       col=c("blue", "red"), pch=1, cex=0.75)
```

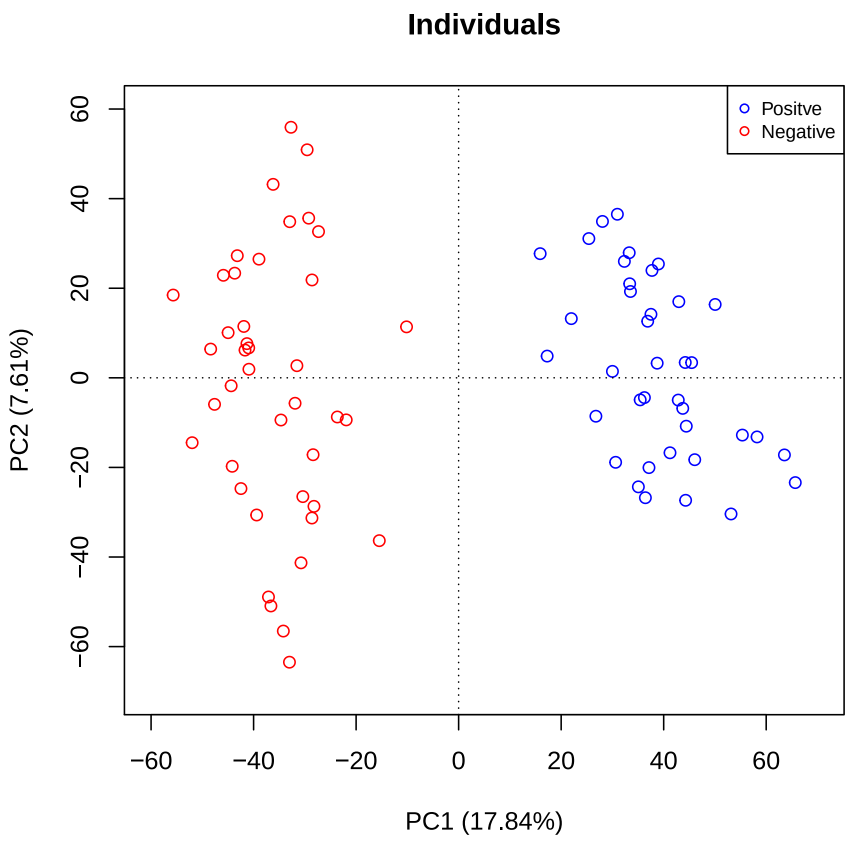

Figure 9: Principal components of breastTCGA dataset
The plot shows the samples according to their estrongen status.

There exist specific functions to compute a PCA on gene expression data. This is the case of function `ord`. We can see that we obtain similar results.

```
group<-as.factor(breastTCGA$ER.Status)
out <- ord(genes, classvec=group, type = "pca")
plot(out, nlab=3)
plotarrays(out)
par(mfrow=c(1,1))
```

Apart from method `blockSVD` there is another method called `generalBlockSVD`. This method is the generalization from the previous one in a multi level algorithm. In order to use it we can set the parameters `k` and `q`. Their default values are 2 and 1, respectively. Parameter `q`

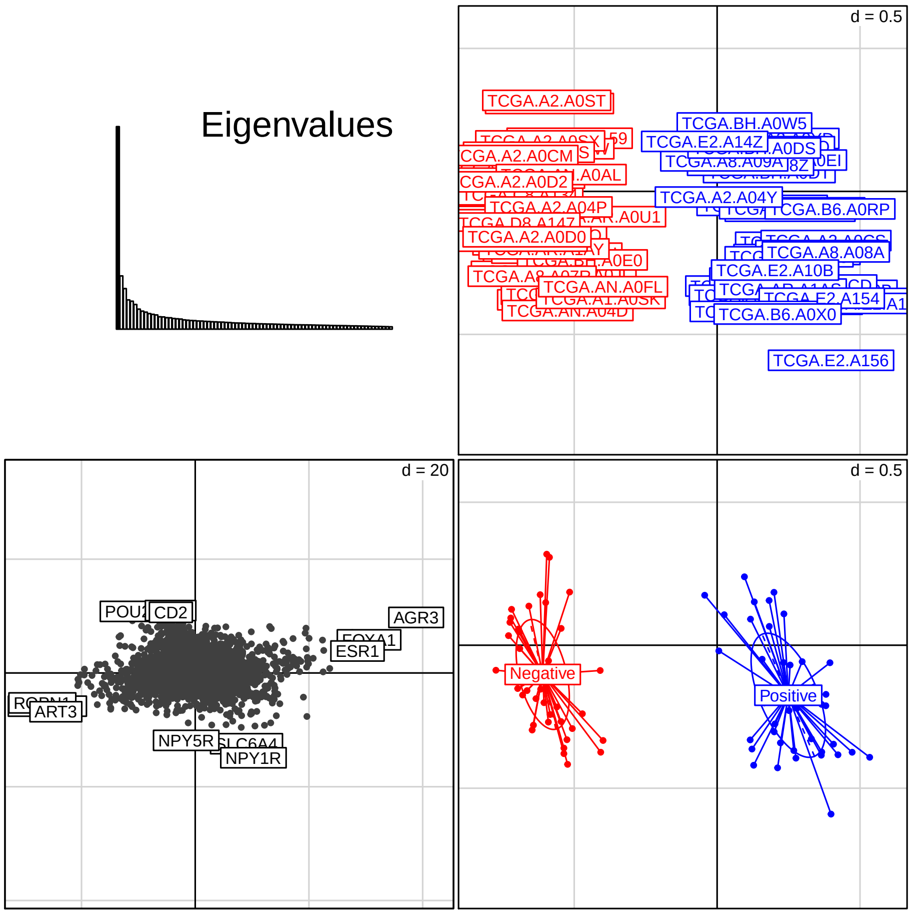

Figure 10: Variables and individuals projections from ord function
First, there is the plot of the eigenvalues. Then, the plot of the variables (top right) and the individuals (bottom left) in the first two principal components, and the individuals classified in groups.

sets the number of levels of the algorithm (how many times we will repeat the process), and `k` sets the number of subdatasets to be concatenated at each level. More information about the method can be found at (Iwen and Ong 2016).

We can see that the same results can be obtained using this method:

```r
gpca <- getPCA(t(genes),  method="generalBlockSVD", k=2, q=2)
plot(gpca$Y[group=="Positive",c(1,2)],
     main="Individuals", type="p", col="blue",
     xlab="PC1 (17.84%)", ylab="PC2 (7.61%)",
     xlim=c(-60,70), ylim=c(-70,60))
points(gpca$Y[group=="Negative",c(1,2)], col="red")
abline(a=0,b=0, lty=3)
abline(v=0, lty=3)
legend("topright", c("Positve", "Negative"),
       col=c("blue", "red"), pch=1, cex=0.75)
```

Finally we can compare the different functions performance using the function `microbenchmark`. It computes the computation time of the functions and returns a table with the results.

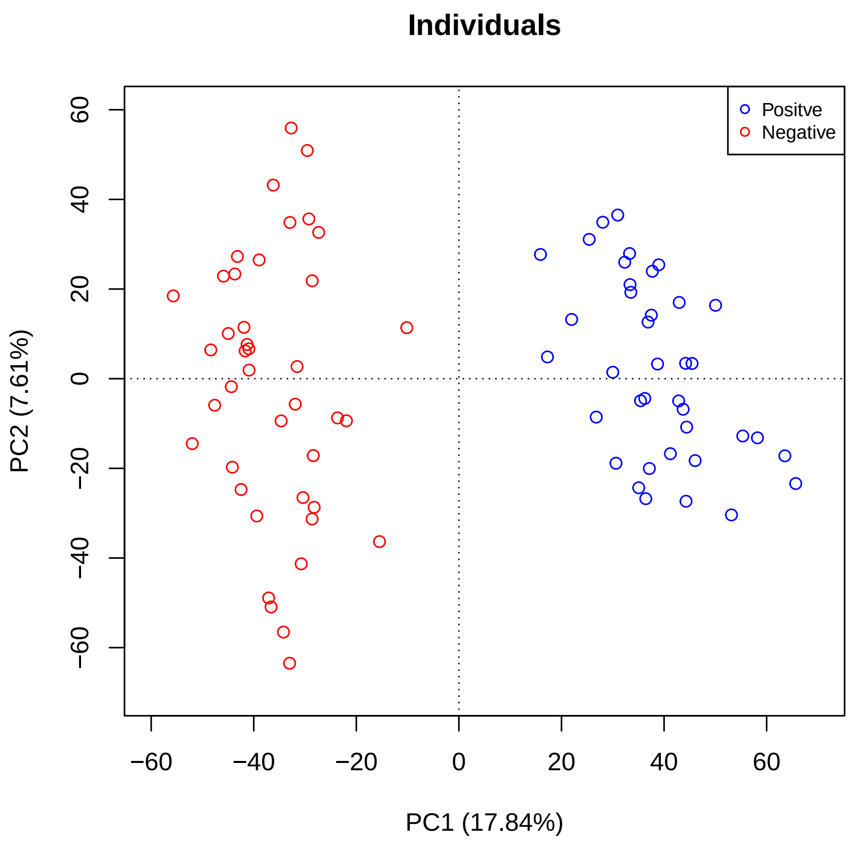

Figure 11: Individuals projections from getPCA function using General Block method
The plot shows the individuals into the two first principal components classified according to their estrongen status.

```
mb <- microbenchmark("Block method (k=2)"= getPCA(t(genes),
                       parts = 2, center=F, scale=F),
                     "Block method (k=5)"=getPCA(t(genes),
                       parts = 5, center=F, scale=F),
                     "General Block method (k=2, q=2)"
                     = getPCA(t(genes), method="generalBlockSVD",
                              k=2, q=2, center=F, scale=F),
                     "General Block method (k=3, q=2)"
                     = getPCA(t(genes), method="generalBlockSVD",
                              k=3, q=2, center=F, scale=F),
                     "PCA function"=PCA(t(genes), graph=FALSE),
                     "ord function" = ord(genes, classvec=group,
                                          type = "pca"), times=10)

autoplot.microbenchmark(mb)
```

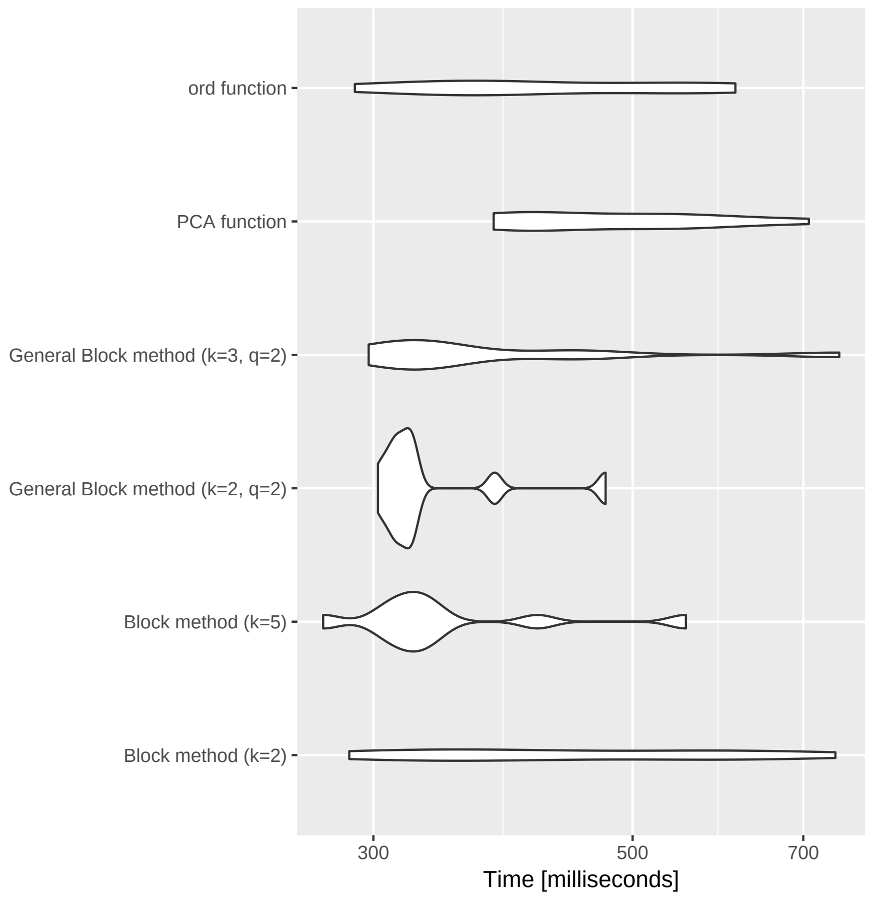

Figure 12: Violinplot of the computational times for different functions
The two in the bottom are done using Block method and the following two are using General Block method. The second has been done using PCA function and the first one is using ord function.

If we look at the mean column we can see that the `PCA` function is the one that last more in average. This can be explained because this function computes a lot of different values a part from the necessary ones to compute a basic PCA. This also happens with function `ord`. These functions compute other scores and it makes it difficult to compare them with the others. The best way to analyze our functions' performance is to compare it to the function `getPCA0`. This function computes a basic PCA using only one SVD of the whole matrix. Our `getPCA` functions are faster in this case than `PCA` and `ord`, but it does not necessarily mean that they are more efficient.

```
mb2 <- microbenchmark("Block method (k=2)"= getPCA(t(genes),
                        parts = 2, center=F, scale=F),
                      "Block method (k=5)"= getPCA(t(genes),
                        parts = 5, center=F, scale=F),
                       "General Block method (k=2,q=2)"
                       = getPCA(t(genes),
                      method="generalBlockSVD", k=2, q=2,
                      center=F, scale=F),
                      "General Block method (k=3, q=2)"
```

```
                              = getPCA(t(genes),
                              method="generalBlockSVD", k=3, q=2,
                              center=F, scale=F),
                              "PCA0 no method" = getPCA0(t(genes),
                              center=F, scale=F),  times=10)

autoplot.microbenchmark(mb2)
```

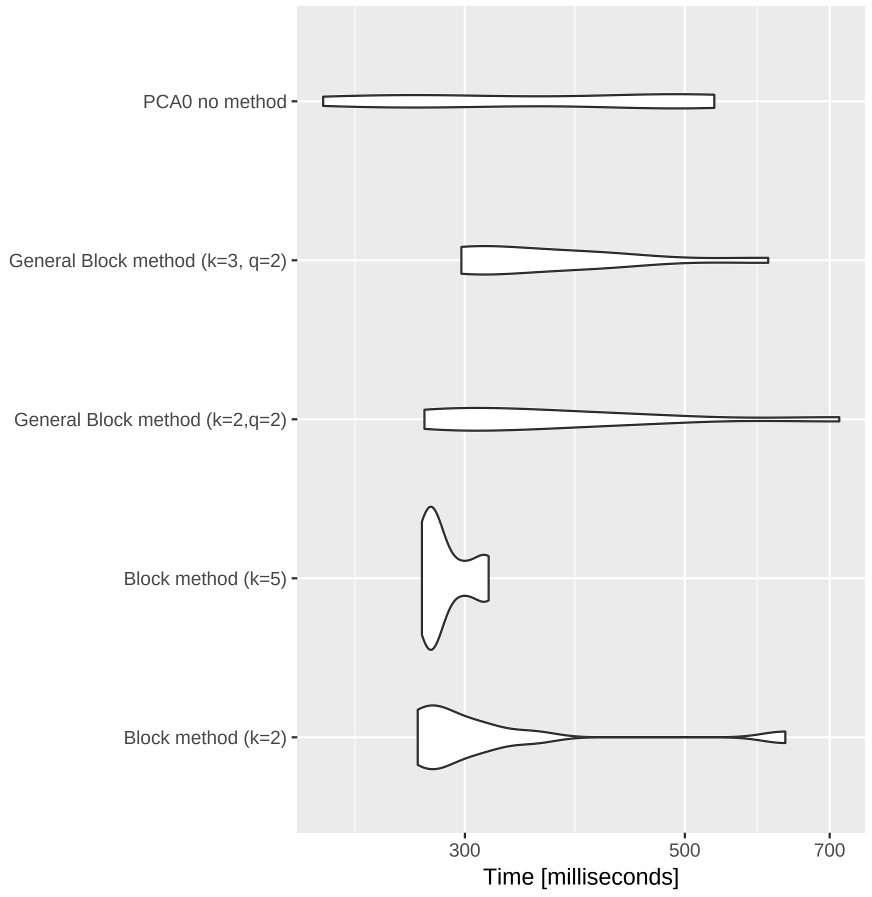

Figure 13: Violinplot of the computational times for different functions
The two in the bottom are done using Block method and the following two have been done using General Block method. The one in the top is using PCA0, it does not use any method.

The size of the `genes` dataset is not very big so the differences between the different functions are not significant. The performance of the Block method gets better when the size of the dataset increases.

When the size of the dataset is really large, we should use the parallel version of the two methods: `parBlockSVD` and `parGeneralBlockSVD`. They work as `blockSVD` and `general BlokSVD`, respectively.

References

Iwen, Mark, and Benjamin Ong. 2016. "A Distributed and Incremental Svd Algorithm for Agglomerative Data Analysis on Large Networks."