Bioconductor classes for working with microarrays and similar data

Alex Sanchez

1 Introduction

Many omics data, once they have been pre-processed, can be stored as numeric data that can be represented as the typical “data matrix”. This matrix is, however, usually transposed, that is genes (variables) are in rows and samples (individuals) are in columns.

A person who is familiar with statistics and R can therefore explore an omics dataset using standard univariate and multivariate statistical methods.

In practice, omics datasets have more information than just what can be stored in a table. This can be annotation data, multiple covariates other than what is in the column names, or information about th eexperimental design or simply the experiment.

Even for a person who is proficient with software, managing simultaneously distinct objects, that contain related information, can be "tricky" and there is always a danger that the distinct components lose synchronization. For instance removing one sample from the expression matrix requires that the corresponding information is removed or updated in the covariates table. And an error at doing this can yield different problems.

In this lab we introduce the `ExpressionSet` class as an option for managing all these pieces of information simultaneously, which not only simplifies the process, but also prevents mistakes derived from lack of consistency between the parts.

The lab has three parts

1- Exploring microarray data.

2. Introducing bioconductor classes to store and access microarray data.
3. Using the `GEOquery` bioconductor package to obtain microarray data.

2 Exploring microarray data. A naïve (simple) approach

In this section we present a real microarray dataset and see how this can be explored using standard `R` functions.

For this exercise we will be using data from a small microarray study which has been deposited in the *Gene Expression Omnibus Database* with the identifier "GSE58435". You can browse all the information from this link: https://www.ncbi.nlm.nih.gov/geo/query/acc.cgi?acc=GSE58435

This study was performed using Affymetryx microarrays (type "HGU133plus2") withe the objective of identifying genes that may play a role in the pathophysiologic changes that are seen in individuals with Turner syndrome, a common sex chromosome aneuploidy, which is associated with malformations.

A preprocessed data matrix is available from the GEO web site, but given that downloading it may require using FTP software, it is provided jointly with this document.

In the following we assume that the matrix has been downloaded and extracted (it is provided as a compressed ".gz") file) in the the working directory.

2.1 Loading the data

The data matrix recovered from the web contains some general information first and the expression values for each sample after line 67.

The first thing to do is to separate both informations. This can be done using the `read.table()` command combined with `skip` and the `nrow` arguments.

Because the last line of the file is a "closing line" with no numbers in it (check it using a text editor) we also have to skip that line.

```
# setwd(" ") # Put  here your working directory
datadir <- "."
info <-readLines(file.path(datadir,"GSE58435_series_matrix.txt
rows2read <- 54743 -66 -2
x <-read.table(file.path(datadir,"GSE58435_series_matrix.txt")
```

Looking at the information contained in the header or in the GEO web site it can be seen that the first five samples correspond to Turner syndrome and the remaining 5 to control samples.

```
dim(x)
```

```
## [1] 54675    10
```

```
colnames(x) <- c(paste("Turner",1:5, sep="_"), paste("Control"
colnames(x)
```

```
##  [1] "Turner_1"  "Turner_2"  "Turner_3"  "Turner_4"  "Turne
##  [7] "Control_2" "Control_3" "Control_4" "Control_5"
```

```
head(x)
```

```
##            Turner_1 Turner_2   Turner_3  Turner_4   Turner_
## 1007_s_at 4.5066744 4.065303  5.4933161 4.2067493  4.168698
## 1053_at   2.7962339 2.208890 -0.3214509 1.3403677 -0.205507
## 117_at    0.0151638 2.248348  2.8628993 3.0157066  1.994660
## 121_at    3.6263578 4.018847  4.5990725 2.0105442  4.735246
## 1255_g_at 5.6135908 5.490236  2.8818537 2.9547538  3.363774
## 1294_at   0.2343619 0.341332  0.4199070 0.2054139  2.481445
##             Control_2  Control_3 Control_4 Control_5
## 1007_s_at  3.88586101  4.3541984 3.4440077  4.294545
## 1053_at    2.08790517 -0.4151921 0.9635010  2.285352
## 117_at     3.63037845  2.8582480 3.5146747  3.728415
## 121_at     2.03179357  2.9919757 1.9267738  3.245589
## 1255_g_at  4.02395529  2.5633916 0.4549052  4.456763
## 1294_at   -0.02923062  0.5397193 0.2182423  1.508753
```

2.2 Exploratory analysis with univariate statistics

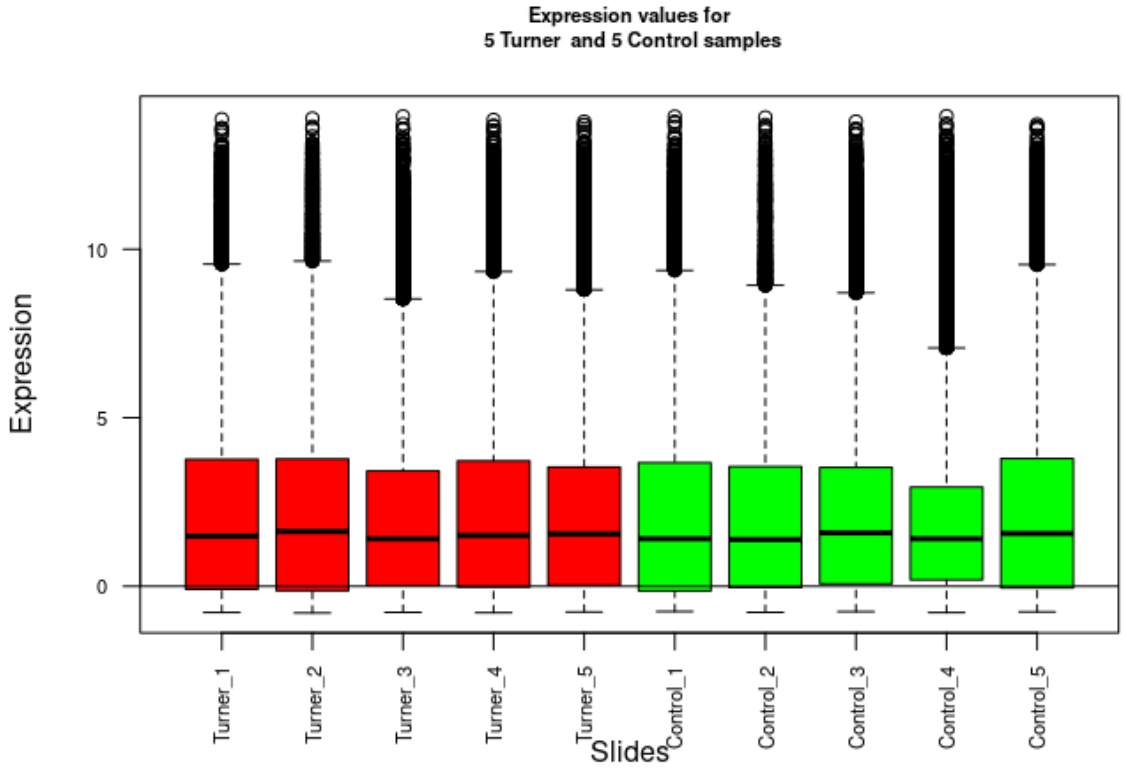
A first glimpse of the dataset can be obtained using basic summary statistics and basic plots.

```
round(apply(x,2, summary),3)  # Column-wise summary statistics
```

```
##         Turner_1 Turner_2 Turner_3 Turner_4 Turner_5 Contro
## Min.      -0.779   -0.792   -0.773   -0.785   -0.765    -0.
## 1st Qu.   -0.091   -0.135    0.018   -0.025    0.026    -0.
## Median     1.486    1.627    1.411    1.507    1.552     1.
## Mean       2.145    2.187    2.070    2.166    2.115     2.
## 3rd Qu.    3.769    3.779    3.421    3.723    3.537     3.
## Max.      13.859   13.879   13.944   13.839   13.787    13.
##         Control_3 Control_4 Control_5
```

```
## Min.       -0.753    -0.781    -0.760
## 1st Qu.     0.074     0.190    -0.051
## Median      1.584     1.410     1.566
## Mean        2.138     1.977     2.184
## 3rd Qu.     3.528     2.942     3.788
## Max.       13.791    13.951    13.707
```

A boxplot of the data shows that values are assymetrically distributed



2.3 Data visualization using unsupervised techniques (PCA, Clustering)

A very useful visualization for omics data obtained by computing "sample-wise" principal components and plotting the first two components.

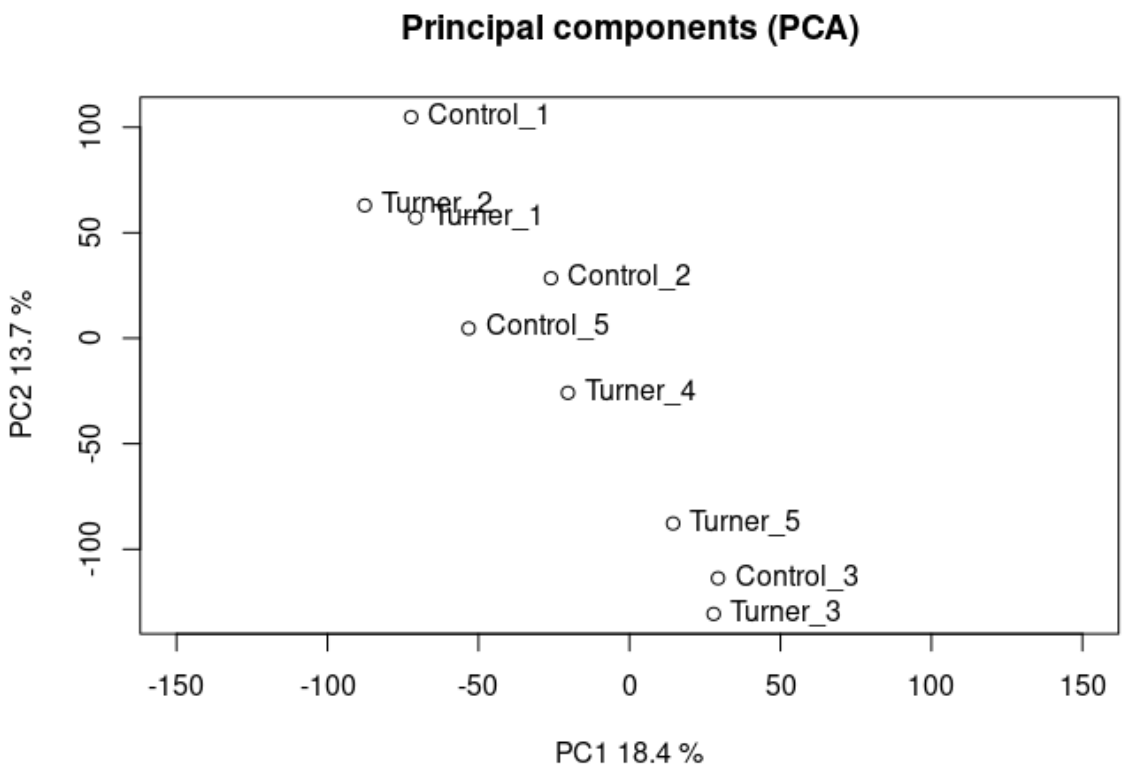
If samples are more similar within groups that beteen this is usually reflected in these plots. For the same reason they can also be useful if the goal is detect unusual samples or batch effects.

Start by computing prncipal components and loadings.

```
pcX<-prcomp(t(x), scale=TRUE)
loads<- round(pcX$sdev^2/sum(pcX$sdev^2)*100,1)
```

Then plot the first two components.

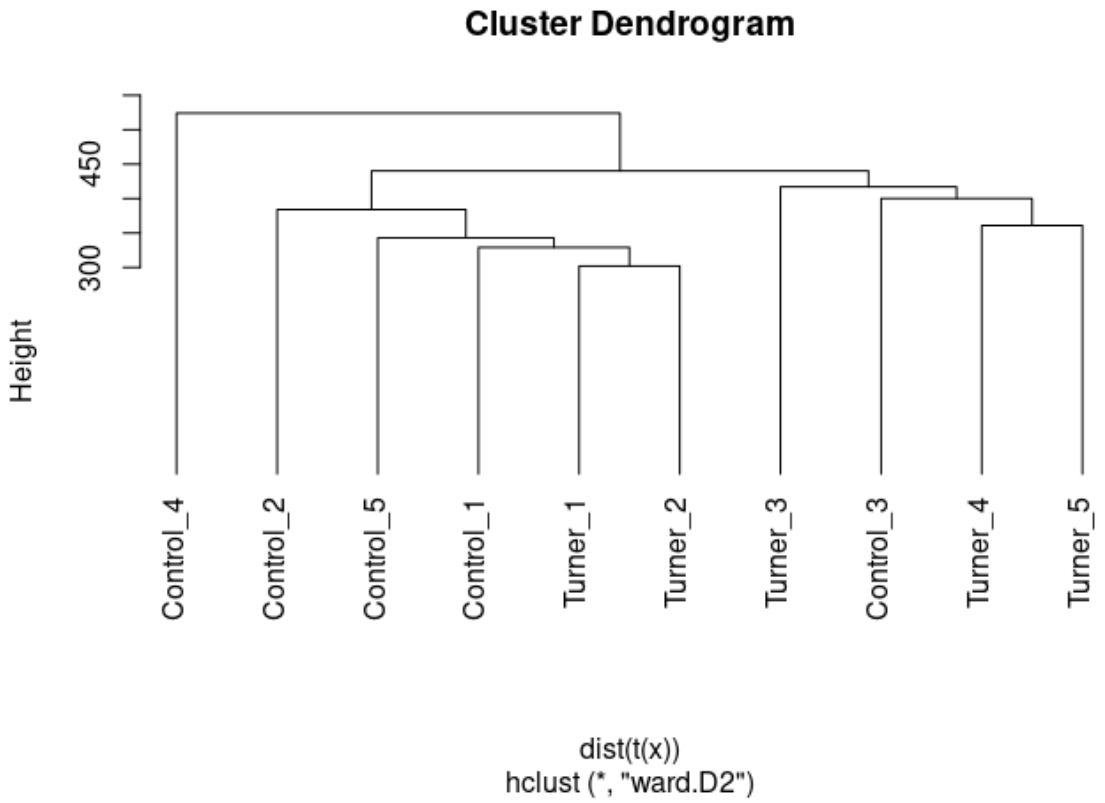
```
xlab<-c(paste("PC1",loads[1],"%"))
ylab<-c(paste("PC2",loads[2],"%"))
plot(pcX$x[,1:2],xlab=xlab,ylab=ylab, xlim=c(-150, 150))
title("Principal components (PCA)")
text(pcX$x[,1],pcX$x[,2],colnames(x), pos=4)
```



Alternatively a hierarchichal clustering can be applied to detect any expected (or unexpected grouping of the samples).

```
clust.euclid.average <- hclust(dist(t(x)),method="ward.D2")
```

```
plot(clust.euclid.average, hang=-1)
```



Both PCA and clustering suggest that the differences between the groups are not very clear which can be attributed to the fact that gene expression may not be the best surrogate for the effects of Turner Syndrome.

2.4 Exercises

This exercises are intended for people who is starting to work with Bioconductor.

1. Go to the website of the Gene Expression Omnibus and Look for a comparative experiment that uses a small number of arrays and try to understand how the information is organized.

2. Download the expressions and the covariate information (both stored in the "Series Matrix File(s)"). Notice that **you need a ftp program such as filezilla to download the file**

3. Reproduce the exploration using the dataset you have downloaded. Feel free to complement it with any additional plot or summary which you fiond interesting.

3 Bioconductor classes to manage micrarray and similar data

3.1 The OOP paradigm

Object-oriented design provides a convenient way to represent data structures and actions performed on them.

- A *class* can be tought of as a template, a description of what constitutes each instance of the class.
- An *instance* of a class is a realization of what describes the class.
- Attributes of a class are data components, and methods of a class are functions, or actions the instance/class is capable of.

The {} language has several implementations of the OO paradigm but, in spite of its success in other languages, it is relatively minoritary.

3.2 Bioconductor Classes

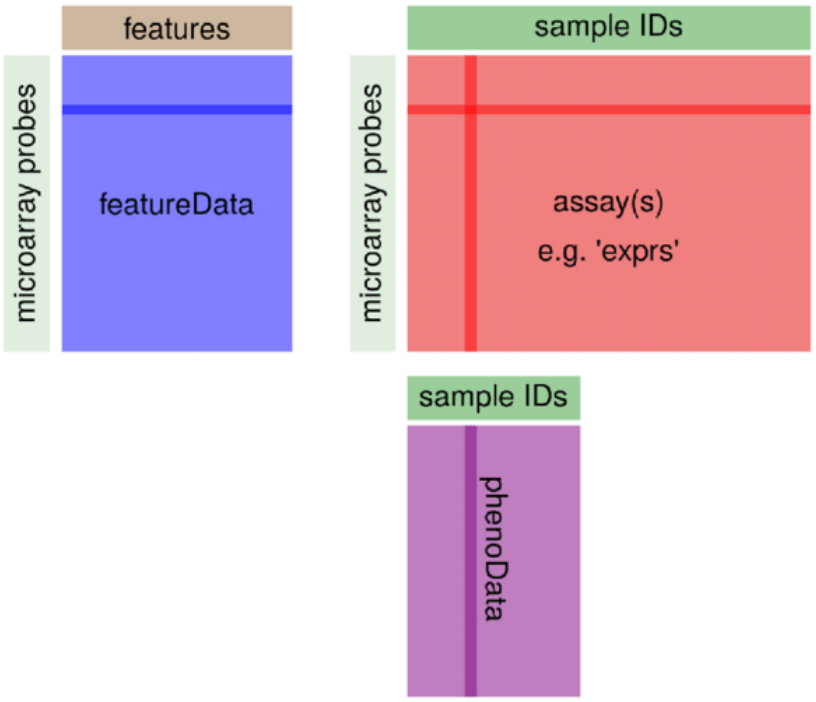
One case where OOP has succeeded in R or, at least, is more used than in others is in the Bioconductor Project (bioconductor.org). In Bioconductor we have to deal with complex data structures such as the results of a microarray experiment, a genome and its annotation or a complex multi-omics dataset. These are situations where using OOP to create classes to manage those complex types of data is clearly appropriate.

3.3 The `Biobase` package

The `R` package{Biobase} package implements one of the best known Bioconductor classes: `ExpressionSet`. It was originally intended to contain microarray data and information on the study that generated them and it has become a standard for similar data structures.

```
require(Biobase)
```

Figure @ref(ExpressionSet) shows the structure of this class. It is essentially a *container* that has distinct slots to store some of the most usual components in an omics dataset.



Structure of the `ExpressionSet` class, showing its slots and their meaning. Reproduced from Klaus, B., & Reisenauer, S. (2018)

The advantage of the OOP approach is that, if a new type of omics data needs a similar but different structure it can be created using inheritance, which means much less work than and better consistency than creating it from scratch.

3.4 A toy dataset

For the purpose of this lab we are going to simulate a toy (fake) dataset that consists of the following:

- **Expression values** A matrix of 30 rows and 10 columns containing expression values from a gene expression experiment. Matrix column names are sample identifiers
- **Covariates** A table of ten rows and four columns containing the sample identifiers, the treatment groups and the age and sex of individuals. {Genes} Information about the features contained in the data. May be the gene names, the probeset identifiers etc. Usually stored in a character vector but may also be a table with distinct annotations per feature.
- **Information about the experiment** Additional information about the study, such as the authors and their contact details or the title and url of the study that originated them.

```
expressionValues <- matrix (rnorm (300), nrow=30)
colnames(expressionValues) <- paste0("sample",1:10)
head(expressionValues)
```

```
##          sample1    sample2    sample3     sample4      samp
## [1,] -1.1350396  0.6398255  0.4829373  0.01680618  1.051014
## [2,]  0.6492049  0.5016141 -0.6600928  1.14319021 -1.386051
## [3,] -1.4497335  0.3790781  0.7308945  0.92903135  0.257408
## [4,]  0.4302514 -1.4317677  2.3946338  0.43610244  0.005777
## [5,] -0.2915217  0.8552503  1.0580434 -1.31528811  0.575793
## [6,]  0.7939579  1.7598164  2.0006651  0.15246185  0.092659
##          sample7   sample8     sample9   sample10
## [1,]  0.1126905 0.2191192  0.99981211  0.6248215
## [2,]  1.7622667 0.5456288 -0.28497541  1.2116382
## [3,]  1.0142685 0.9207880 -0.03968217 -1.5801708
## [4,]  0.2274577 1.8962846 -0.38660431 -1.3688258
## [5,] -0.2589717 0.1060780  2.17899238 -0.2770203
## [6,] -0.7406773 1.2858179  1.30290319  1.0484484
```

VERY IMPORTANT: To create the ExpressionSet the following has to be verified:

- The names of the columns of the object that contains the expressions, that will be stored in `assayData`
- must match the names of the rows of the object that contains the covariates, that will be stored in `phenoData`.

In this example it is saved in the variable `sampleNames` but this field will be used as the *name of the rows*, not as another column

```
targets <- data.frame(sampleNames = paste0("sample",1:10),
                      group=c(paste0("CTL",1:5),paste0("TR",1:
                      age = rpois(10, 30),
                      sex=as.factor(sample(c("Male", "Female")
                      row.names=1)
head(targets, n=10)
```

```
##          group age    sex
## sample1   CTL1  24 Female
## sample2   CTL2  32   Male
## sample3   CTL3  30   Male
```

```
## sample4   CTL4  41 Female
## sample5   CTL5  25   Male
## sample6    TR1  25   Male
## sample7    TR2  36 Female
## sample8    TR3  31   Male
## sample9    TR4  34   Male
## sample10   TR5  33   Male
```

```
myGenes <-  paste0("gene",1:30)
```

```
myInfo=list(myName="Alex Sanchez", myLab="Bioinformatics Lab",
          myContact="alex@somemail.com", myTitle="Practical Ex
show(myInfo)
```

```
## $myName
## [1] "Alex Sanchez"
##
## $myLab
## [1] "Bioinformatics Lab"
##
## $myContact
## [1] "alex@somemail.com"
##
## $myTitle
## [1] "Practical Exercise on ExpressionSets"
```

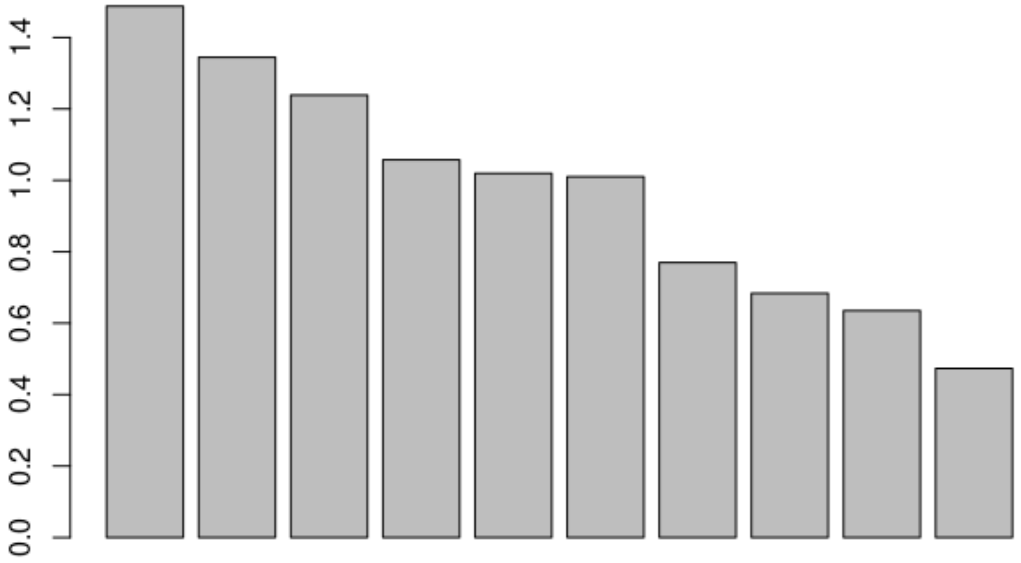
Having data stored in this way is usually enough for most of the analyes we may want to do. The only unconvenient comes from the fact that the information about the same individuals is in separate R objects so that, for certain applications, we will have to access several objects and *assume they are well related*.

For example if we want to make a principal components analysis and plot the groups by treatment we need to use both `expressionValues" and` targets."

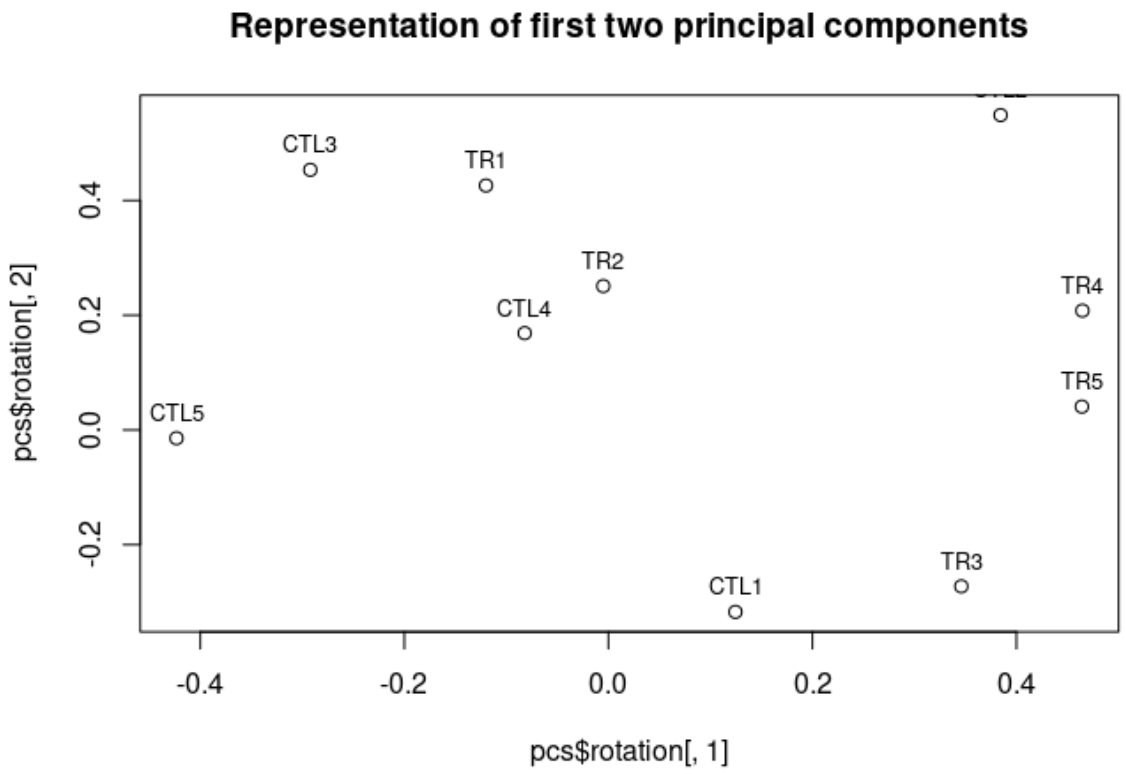
```
pcs <- prcomp(expressionValues)
names(pcs)
```

```
## [1] "sdev"     "rotation" "center"   "scale"    "x"
```

```r
barplot(pcs$sdev)
```




```r
plot(pcs$rotation[,1], pcs$rotation[,2], main="Representation
text(pcs$rotation[,1], pcs$rotation[,2], targets$group, cex=0.
```



Or, if we sort the genes from most to least variable and whant to see which are the top variable genes. We need to use both objects `expressionValues" and` myGenes" assuming they are well linked:

```
variab <- apply(expressionValues, 1, sd)
orderedGenes <- myGenes[order(variab, decreasing=TRUE)]
head(variab[order(variab, decreasing=TRUE)])
```

```
## [1] 1.344437 1.297292 1.296048 1.269957 1.223432 1.197980
```

```
head(orderedGenes)
```

```
## [1] "gene22" "gene4"  "gene9"  "gene27" "gene8"  "gene14"
```

Imagine we are informed that individual has to be removed. We have to do it in "expressionValues" and "targets".

```
newExpress<- expressionValues[,-9]
newTargets <- targets[-9,]
wrongNewTargets <- targets [-10,]
```

3.5 Creating and using objects of class ExpressionSet

In order to use a class we need to *instantiate* it, that is we need to create an object of this class.

This can be done using the generic constructor `new` or with the function `ExpressionSet`.

Both the constructor or the function require a series of parameters which roughly correspond to the slots of the class (type `? ExpressionSet` to see a list of compulsory and optional arguments).

In the following subsections we describe how to create an `ExpressionSet` using the components of the toy dataset. Some of the elements will directly be the element in the toy dataset, such as the expression matrix. For others such as the covariates or the experiment information, specific classes have been introduced so that we have to instantiate these classes first and then use the the objects created to create the `ExpressionSet` object.

3.5.1 Slot `AssayData`

The main element, and indeed the only one to be provided to create an `ExpressionSet`, is `AssayData`. For our practical purposes it can be seen as a matrix with as many rows as genes or generically "features" and as many columns as samples or individuals.

```
myEset <- ExpressionSet(expressionValues)
class(myEset)
```

```
## [1] "ExpressionSet"
## attr(,"package")
## [1] "Biobase"
```

```
show(myEset)
```

```
## ExpressionSet (storageMode: lockedEnvironment)
## assayData: 30 features, 10 samples
##   element names: exprs
## protocolData: none
## phenoData: none
## featureData: none
## experimentData: use 'experimentData(object)'
## Annotation:
```

3.5.2 Information about covariates

Covariates, such as those contained in the "targets" data frame are not included in the "ExpressionSet" "as.is". Instead we have first to create an intermediate object of class `AnnotatedDataFrame`.

Class `R` class{AnnotatedDataFrame} is intended to contain a data frame where we may want to provide enhanced information for columns, i.e. besides the short column names, longer labels to describe them better.

The information about covariates, contained in an instance of class `AnnotatedDataFrame`, is stored in the slot `phenoData`.

```
columnDesc <-  data.frame(labelDescription= c("Treatment/Contr
                                              "Age at diseas
                                              "Sex of patien
myAnnotDF <- new("AnnotatedDataFrame", data=targets, varMetada
show(myAnnotDF)
```

```
## An object of class 'AnnotatedDataFrame'
##   rowNames: sample1 sample2 ... sample10 (10 total)
##   varLabels: group age sex
##   varMetadata: labelDescription
```

Notice that we have not included a label for sample names because this information is not a column of the `phenoData` object.

Once we have an `AnnotatedDataFrame` we can add it to the `ExpressionSet`

```
phenoData(myEset) <- myAnnotDF
```

Alternatively we could have created the`AnnotatedDataFrame` object first and then create the `ExpressionSet` object with both the expression values and the covariates. In this case it would be required that the expression matrix colum names are the same as the targets row names.

```
myEset <- ExpressionSet(assayData=expressionValues, phenoData=
show(myEset)
```

```
## ExpressionSet (storageMode: lockedEnvironment)
## assayData: 30 features, 10 samples
##   element names: exprs
## protocolData: none
## phenoData
##   sampleNames: sample1 sample2 ... sample10 (10 total)
##   varLabels: group age sex
##   varMetadata: labelDescription
## featureData: none
## experimentData: use 'experimentData(object)'
## Annotation:
```

3.5.3 Adding information about features

Similarly to what we do to store information about covariates, information about genes (or generically "features") may be stored in the optional slot `featureData` as an `AnnotatedDataFrame`.

The number of rows in `featureData` must match the number of rows in `assayData`. Row names of `featureData` must match row names of the matrix / matrices in assayData.

This slot is good if one has an annotations table that one wishes to store and manage jointly with the other values. ALternatively we can simple store the names of the features using a character vector in the slot `featureNames`.

```
myEset <- ExpressionSet(assayData=expressionValues,
                        phenoData=myAnnotDF,
                        featureNames =myGenes)
# show(myEset)
```

3.5.4 Storing information about the experiment

In a similar way to what happens with the `AnnotatedDataFrame` class there has been developed a class to store information about the experiment. The structure of the class, called `MIAME` follows the structur of what has been described as the "Minimum Information About a Microarray Experiment" see www.ncbi.nlm.nih.gov/pubmed/11726920

This is useful information but it is clearly optional for data analysis.

```
myDesc <- new("MIAME", name= myInfo[["myName"]],
              lab= myInfo[["myLab"]],
              contact= myInfo[["myContact"]] ,
              title=myInfo[["myTitle"]])
print(myDesc)
```

```
## Experiment data
##   Experimenter name: Alex Sanchez
##   Laboratory: Bioinformatics Lab
##   Contact information: alex@somemail.com
##   Title: Practical Exercise on ExpressionSets
##   URL:
##   PMIDs:
##   No abstract available.
```

Again we could add this object to the `ExpressionSet` or use it when creating it from scratch.

```
myEset <- ExpressionSet(assayData=expressionValues,
                        phenoData=myAnnotDF,
                        fetureNames =myGenes,
                        experimentData = myDesc)
# show(myEset)
```

3.6 Using objects of class `ExpressionSet`

The advantage of working with `ExpressionSets` lies in the fact that action on the objects are done in such a way that its consistency is ensured. That means for instance that if we subset the `ExpressionSet` it is automatically done on the columns of the expressions and on the rows of the covariates and it is no possible that a distinct row/column are removed.

The following lines illustrate some management of data in an `ExpressionSet`.

3.6.1 Accessing Slot values

Notice that to access the values we use special functions called "accessors" instead of the dollar symbol (which would not work for classes) or the @ symbol that does substitute the $ symbol.

Notice also that, in order to access the data frame contained in the `phenoData` slot, which is an `AnnotatedDataFrame`, we need to use two accessors: `phenoData` to access the `ExpressionSet`'s`phenoData` slot and `pData` to access the `data` slot in it. It is strange until you get used to it!

```
dim(exprs(myEset))
```

```
## [1] 30 10
```

```
class(phenoData(myEset))
```

```
## [1] "AnnotatedDataFrame"
## attr(,"package")
## [1] "Biobase"
```

```
class(pData(phenoData(myEset)))
```

```
## [1] "data.frame"
```

```
head(pData(phenoData(myEset)))
```

```
##         group age    sex
## sample1  CTL1  24 Female
## sample2  CTL2  32   Male
## sample3  CTL3  30   Male
## sample4  CTL4  41 Female
## sample5  CTL5  25   Male
## sample6   TR1  25   Male
```

```
head(pData(myEset))
```

```
##         group age    sex
## sample1  CTL1  24 Female
## sample2  CTL2  32   Male
## sample3  CTL3  30   Male
## sample4  CTL4  41 Female
## sample5  CTL5  25   Male
## sample6   TR1  25   Male
```

3.6.2 Subsetting ExpressionSet

This is where the interest of using ExpressionSets is most clearly realized.

The ExpressionSet object has been cleverly-designed to make data manipulation consistent with other basic R object types. For example, creating a subset of an ExpressionsSet will subset the expression matrix, sample information and feature annotation (if available) simultaneously in an appropriate manner. The user does not need to know how the object is represented "under-the-hood". In effect, we can treat the ExpressionSet as if it is a standard R data frame

```
smallEset <- myEset[1:15,c(1:3,6:8)]
dim(exprs(smallEset))
```

```
## [1] 15  6
```

```
dim(pData(smallEset))
```

```
## [1] 6 3
```

```
head(pData(smallEset))
```

```
##         group age    sex
## sample1  CTL1  24 Female
```

```
## sample2  CTL2  32   Male
## sample3  CTL3  30   Male
## sample6   TR1  25   Male
## sample7   TR2  36 Female
## sample8   TR3  31   Male
```

```
all(colnames(exprs(smallEset))==rownames(pData(smallEset)))
```

```
## [1] TRUE
```

We can for instance create a new dataset for all individuals younger than 30 or for all females without having to worry about doing it in every component.

```
youngEset <- myEset[,pData(myEset)$age<30]
dim(exprs(youngEset))
```

```
## [1] 30  3
```

```
head(pData(youngEset))
```

```
##         group age    sex
## sample1  CTL1  24 Female
## sample5  CTL5  25   Male
## sample6   TR1  25   Male
```

3.7 Exercises

4. Create an `ExpressionSet` object to contain the data for the example study using the data you have downloaded and used in the first section. That is, adapt the steps taken to creat the ExpressionSet with the toy dataset to create one with the data from the study.

5. Do some subsetting and check the consistency of the results obtained. For example remove some sample from

the covariates slot (the `phenoData`) and see if it is automatically removed from the expression matrix`.

6. Check that you are able to reproduce the analysis in the first part accessing the components of the object created.

4 Using the `GEOquery` bioconductor package to obtain microarray data

4.1 Overview of GEO

The NCBI Gene Expression Omnibus (GEO) serves as a public repository for a wide range of high-throughput experimental data. These data include single and dual channel microarray-based experiments measuring mRNA, genomic DNA, and protein abundance, as well as non-array techniques such as serial analysis of gene expression (SAGE), mass spectrometry proteomic data, and high-throughput sequencing data.

At the most basic level of organization of GEO, there are four basic entity types. The first three (Sample, Platform, and Series) are supplied by users; the fourth, the dataset, is compiled and curated by GEO staff from the user-submitted data. See the GEO home page for more information.

4.2 Getting data from GEO

Getting data from GEO is really quite easy. There is only one command that is needed, getGEO.

This one function interprets its input to determine how to get the data from GEO and then parse the data into useful R data structures. Usage is quite simple.

```
if (!require(GEOquery)) {
  BiocManager::install("GEOquery")
}
require(GEOquery)
```

```
gse <- getGEO("GSE58435")
class(gse)
```

```
## [1] "list"
```

```
names(gse)
```

```
## [1] "GSE58435_series_matrix.txt.gz"
```

```
gse[[1]]
```

```
## ExpressionSet (storageMode: lockedEnvironment)
## assayData: 54675 features, 10 samples
##   element names: exprs
## protocolData: none
## phenoData
##   sampleNames: GSM1411021 GSM1411022 ... GSM1411030 (10 tota
##   varLabels: title geo_accession ... karyotype:ch1 (34 tota
##   varMetadata: labelDescription
## featureData
##   featureNames: 1007_s_at 1053_at ... AFFX-TrpnX-M_at (5467
##   fvarLabels: ID GB_ACC ... Gene Ontology Molecular Functio
##   fvarMetadata: Column Description labelDescription
## experimentData: use 'experimentData(object)'
##   pubMedIds: 24850140
## Annotation: GPL570
```

```
esetFromGEO <- gse[[1]]
```

The downloaded object is an ExpressionSet stored in a list. This means that instead of doing the painful process of creating the object step by step one can simply download it from GEO and start using it as in the previous section.

4.3 Exercises

7. Last, create an `ExpressionSet` object to contain the data for this study but *instead of creating it from Scratch* use the `getGEO` function of the Bioconductor package `GEOquery` as described in the second part of the document

5 References

- Davis, S., & Meltzer, P. (2007). GEOquery: a bridge between the Gene Expression Omnibus (GEO) and BioConductor. Bioinformatics, 14, 1846–1847.
- Klaus, B., & Reisenauer, S. (2018). An end to end workflow for differential gene expression using Affymetrix microarrays [version 2; referees: 2 approved]. F1000Research, 5, 1384. https://doi.org/10.12688/f1000research.8967.2
- Clough, E., & Barrett, T. (2016). The Gene Expression Omnibus Database. In Methods in molecular biology (Clifton, N.J.) (Vol. 1418, pp. 93–110). https://doi.org/10.1007/978-1-4939-3578-9_5