Bioconductor classes for working with microarrays or similar data

Alex Sanchez-Pla

2024-02-26

Table of contents

1 Introduction 2

1.1 Availability . . . 2

2 Bioconductor classes to manage micrarray and similar data 2

2.1 The OOP paradigm . . . 2

2.2 Bioconductor Classes . . . 3

2.3 The `Biobase` package . . . 3

2.4 A toy dataset . . . 4

2.5 Creating and using objects of class ExpressionSet . . . 9

2.5.1 Slot AssayData . . . 9

2.5.2 Information about covariates . . . 10

2.5.3 Adding information about features . . . 11

2.5.4 Storing information about the experiment . . . 11

2.6 Using objects of class ExpressionSet . . . 12

2.6.1 Accessing Slot values . . . 12

2.6.2 Subsetting `ExpressionSets` . . . 14

2.7 Exercises . . . 15

3 The `GEOquery` package to download data from GEO 15

3.1 Downloading a dataset in GSE format . . . 16

3.2 Downloading a dataset in GSD format . . . 19

4 Exercises 21

5 References 22

6 Additional info 22

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 two parts

1. Introduces bioconductor classes to store and access microarray data.
2. Shows how to use the `GEOquery` bioconductor package to download microarray data into an analysis-ready form.

1.1 Availability

This document can be re-created using the repository

2 Bioconductor classes to manage micrarray and similar data

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

2.2 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.

2.3 The `Biobase` package

The `Rpackage`{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.

```r
library(Biobase)
```

```
Loading required package: BiocGenerics


Attaching package: 'BiocGenerics'

The following objects are masked from 'package:stats':

    IQR, mad, sd, var, xtabs

The following objects are masked from 'package:base':

    anyDuplicated, aperm, append, as.data.frame, basename, cbind,
    colnames, dirname, do.call, duplicated, eval, evalq, Filter, Find,
    get, grep, grepl, intersect, is.unsorted, lapply, Map, mapply,
    match, mget, order, paste, pmax, pmax.int, pmin, pmin.int,
    Position, rank, rbind, Reduce, rownames, sapply, setdiff, sort,
    table, tapply, union, unique, unsplit, which.max, which.min
```

```
Welcome to Bioconductor

    Vignettes contain introductory material; view with
    'browseVignettes()'. To cite Bioconductor, see
    'citation("Biobase")', and for packages 'citation("pkgname")'.
```

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.

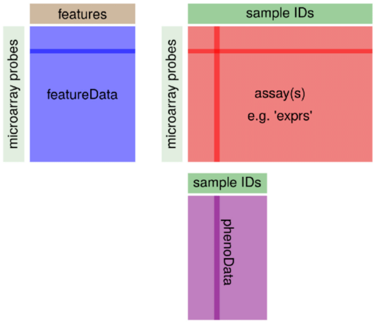


Figure 1: Structure of the ExpressionSet class, showing its slots and their meaning. Reproduced from Klaus, B., & Reisenauer, S. (2018)

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.

2.4 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     sample5    sample6
[1,]  0.46916094  0.3903435  0.4357728  0.470767648  1.17744987  0.5287082
[2,]  0.05107398 -0.1434228  0.2297242  1.342357235  0.09091712 -0.1798971
[3,]  1.52827288 -0.3463219  0.2508739 -0.604185590  0.07860402 -1.0445360
[4,]  0.39494043 -0.3227243 -0.4913008 -0.009780294 -0.66069818  2.0878878
[5,] -0.02984719  0.7494648  1.0872786  0.092532099  1.64245541 -0.5900602
[6,] -0.56134721  0.3410694  0.3892843  0.081827549  0.85580619  1.9978552
         sample7     sample8     sample9    sample10
[1,]  0.95977990 -1.81781295  0.99774455  0.76727411
[2,]  1.50877670 -0.69993087 -0.03040642 -1.42816307
[3,]  0.04855134 -0.06790493  0.63223253  0.04140840
[4,]  1.68167923 -0.58783279  2.27166753  2.34513056
[5,]  1.12586061  0.02476171  0.35220536  1.16030548
[6,] -0.33879569  1.39494530 -0.03770080  0.04096671
```

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",1:5)),
                      age = rpois(10, 30),
                      sex=as.factor(sample(c("Male", "Female"),10,replace=TRUE)),
                      row.names=1)
head(targets, n=10)
```

```
         group age    sex
sample1   CTL1  23 Female
sample2   CTL2  27 Female
sample3   CTL3  34   Male
sample4   CTL4  34 Female
```

```
sample5   CTL5  36 Female
sample6    TR1  22 Female
sample7    TR2  25 Female
sample8    TR3  27   Male
sample9    TR4  36   Male
sample10   TR5  20 Female
```

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

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

```
$myName
[1] "Alex Sanchez"

$myLab
[1] "Bioinformatics Lab"

$myContact
[1] "alex@somemail.com"

$myTitle
[1] "Practical Exercise on ExpressionSets"
```

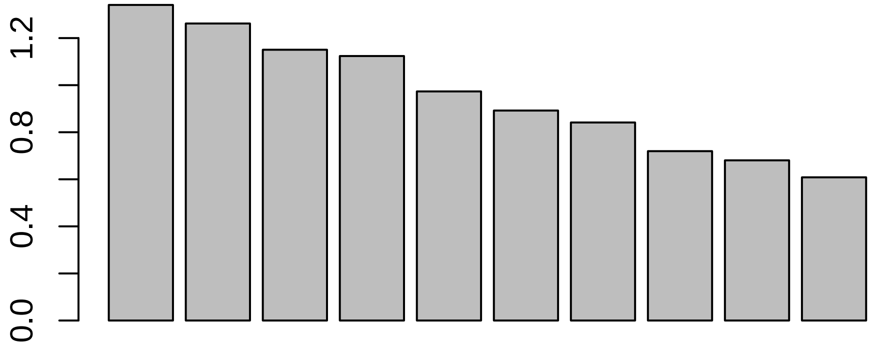
Having data stored in this way is usually enough for most of the analyes we may want to do. The only unconvenient comes from the fact that the information about the same individuals is in separate R objects so that, for certain applications, we will have to access several objects and assume they are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

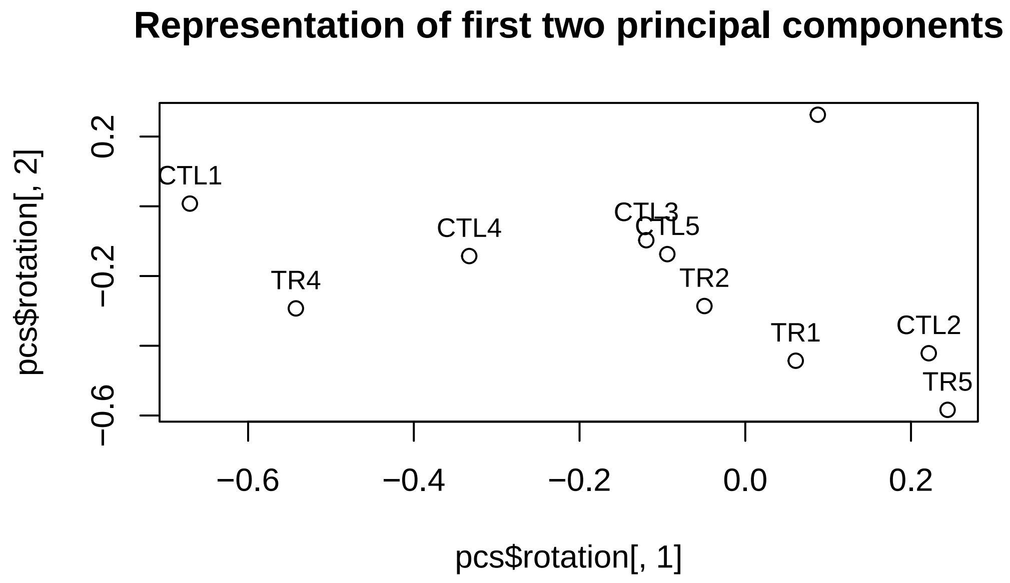
```r
pcs <- prcomp(expressionValues)
names(pcs)
```

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

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



```
plot(pcs$rotation[,1], pcs$rotation[,2],
     main="Representation of first two principal components")
text(pcs$rotation[,1], pcs$rotation[,2], targets$group, cex=0.8, pos=3)
```



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" andmyGenes” assuming they are well linked:

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

```
[1] 1.359100 1.325969 1.274976 1.214838 1.196113 1.144608
```

```
head(orderedGenes)
```

```
[1] "gene10" "gene7"  "gene4"  "gene11" "gene18" "gene9"
```

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,]
```

It is relatively easy to make an unnoticeable mistake in removing unrelated values from the data matrix and the targets table. If instead of removing individual 9 we remove individual 10 it may be difficult to realize what has happened unless it causes a clear unconsistency!

2.5 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.

2.5.1 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:
```

2.5.2 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 Rclass{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/Control",
                                                "Age at disease onset",
                                                "Sex of patient (Male/Female"))
myAnnotDF <- new("AnnotatedDataFrame", data=targets, varMetadata= columnDesc)
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 theAnnotatedDataFrame 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=myAnnotDF)
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:
```

2.5.3 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.

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

2.5.4 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)
```

2.6 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.

2.6.1 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'sphenoData 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  23 Female
sample2  CTL2  27 Female
sample3  CTL3  34   Male
sample4  CTL4  34 Female
sample5  CTL5  36 Female
sample6   TR1  22 Female
```

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

```
        group age    sex
sample1  CTL1  23 Female
sample2  CTL2  27 Female
sample3  CTL3  34   Male
sample4  CTL4  34 Female
sample5  CTL5  36 Female
sample6   TR1  22 Female
```

2.6.2 Subsetting ExpressionSets

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  23 Female
sample2  CTL2  27 Female
sample3  CTL3  34   Male
sample6   TR1  22 Female
sample7   TR2  25 Female
sample8   TR3  27   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  6
```

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

```
         group age    sex
sample1   CTL1  23 Female
sample2   CTL2  27 Female
sample6    TR1  22 Female
sample7    TR2  25 Female
sample8    TR3  27   Male
sample10   TR5  20 Female
```

2.7 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.

3 The `GEOquery` package to download data from 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. More information is available in the GEO site and in the document Analisis_de_datos_omicos-Ejemplo_0-Microarrays available in github.

Data can be downloaded from GEO in a wide variety of formats and using a variety of mechanisms. See the download page in this link.

Here we focus on an alternative based on Bioconductor, the `GEOquery` package (http://bioconductor.org/packages/release/bioc/html/GEOquery.html)

This package has been developed **to facilitate downloading data from GEO and turning them into objects of Bioconductor classes such as `expressionSets`**

The best way to learn how to use this package is following its vignette, available at the package site.

Here we only describe how to download a datset using either its series ("GSExxx") or its Dataset ("GDSxxx") identifier.

In the following lines we illustrate how to get the data for this example using the dataset used in the case study Analisis_de_datos_omicos-Ejemplo_0-Microarrays, avilable from github.

As can be seen there the the dataset has the following identifiers:

- Series accesion ID for : GSE27174
- Dataset accesion ID for : GDS4155
- Plattform accession ID : GPL6246

3.1 Downloading a dataset in GSE format

Getting a series dataset from GEO is straightforward. There is only one command that is needed: `getGEO`.

This function interprets its input (depending on the data format) to determine how to get the data from GEO and then parse the data into useful R data structures.

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

```
Loading required package: GEOquery
```

```
Setting options('download.file.method.GEOquery'='auto')
```

```
Setting options('GEOquery.inmemory.gpl'=FALSE)
```

```r
require(GEOquery)
gse <- getGEO("GSE27174", GSEMatrix=TRUE, AnnotGPL=TRUE)
```

```
Found 1 file(s)
```

```
GSE27174_series_matrix.txt.gz
```

If the data format required is a “Series” (GSExxxx) the function returns a list, each of which elements is an expressionSet (this is so because sometimes a Series may have several collections of samples).

```
class(gse)
```

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

```
names(gse)
```

```
[1] "GSE27174_series_matrix.txt.gz"
```

```
length(gse)
```

```
[1] 1
```

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

```
ExpressionSet (storageMode: lockedEnvironment)
assayData: 35557 features, 8 samples
  element names: exprs
protocolData: none
phenoData
  sampleNames: GSM671653 GSM671654 ... GSM671660 (8 total)
  varLabels: title geo_accession ... strain:ch1 (40 total)
  varMetadata: labelDescription
featureData
  featureNames: 10338001 10338002 ... 10608724 (35557 total)
  fvarLabels: ID Gene title ... GO:Component ID (21 total)
  fvarMetadata: Column Description labelDescription
experimentData: use 'experimentData(object)'
  pubMedIds: 21725324
Annotation: GPL6246
```

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

By creating the expressionSet automatically the slow process of creating the object step by step, as in the previous section, can be avoided.

The expressioSet can now be used as usual:

```
head(exprs(esetFromGEO))
```

```
         GSM671653 GSM671654 GSM671655 GSM671656 GSM671657 GSM671658 GSM671659
10338001  13.12027  12.97898  12.99977  12.93720  13.07715  13.06317  12.87192
10338002   6.47144   6.29206   6.60156   6.09510   6.79910   5.77111   5.73771
10338003  11.50182  11.23240  11.11705  11.03028  11.35657  11.42738  10.91709
10338004  10.37514  10.21853  10.29204  10.17732  10.55388  10.43201  10.07903
10338005   1.78245   1.76433   2.77200   1.95012   2.11798   1.65184   2.10928
10338006   2.72243   2.20203   1.60098   2.70849   3.06379   2.31079   1.93041
         GSM671660
10338001  12.91035
10338002   7.02775
10338003  11.09959
10338004  10.43057
10338005   1.70317
10338006   1.78245
```

We can look at the covariates information, but the phenoData object created automatically contains lot of repeated information. Eventually we can explore it and decide which columns we keep and whichs may be removed. For instance we keep the last two columns and see that column 39 contains the information that defines the groups.

```
colnames(pData(esetFromGEO))
```

```
 [1] "title"                     "geo_accession"
 [3] "status"                    "submission_date"
 [5] "last_update_date"          "type"
 [7] "channel_count"             "source_name_ch1"
 [9] "organism_ch1"              "characteristics_ch1"
[11] "characteristics_ch1.1"     "characteristics_ch1.2"
[13] "characteristics_ch1.3"     "characteristics_ch1.4"
[15] "treatment_protocol_ch1"    "growth_protocol_ch1"
[17] "molecule_ch1"              "extract_protocol_ch1"
[19] "label_ch1"                 "label_protocol_ch1"
[21] "taxid_ch1"                 "hyb_protocol"
[23] "scan_protocol"             "data_processing"
```

```
[25] "platform_id"             "contact_name"
[27] "contact_department"      "contact_institute"
[29] "contact_address"         "contact_city"
[31] "contact_state"           "contact_zip/postal_code"
[33] "contact_country"         "supplementary_file"
[35] "data_row_count"          "cell type:ch1"
[37] "developmental stage:ch1" "genotype/variation:ch1"
[39] "infection status:ch1"    "strain:ch1"
```

```
pData(esetFromGEO)[,39:40]
```

```
GSM671653 infected with lentiviruses expressing the three dopaminergic transcription factors
GSM671654 infected with lentiviruses expressing the three dopaminergic transcription factors
GSM671655 infected with lentiviruses expressing the three dopaminergic transcription factors
GSM671656 infected with lentiviruses expressing the three dopaminergic transcription factors
GSM671657
GSM671658
GSM671659
GSM671660
          strain:ch1
GSM671653   C57BL/6J
GSM671654   C57BL/6J
GSM671655   C57BL/6J
GSM671656   C57BL/6J
GSM671657   C57BL/6J
GSM671658   C57BL/6J
GSM671659   C57BL/6J
GSM671660   C57BL/6J
```

3.2 Downloading a dataset in GSD format

Eventually, we may prefer to download the data in GSD format.

```
gds <- getGEO("GDS4155")
```

The object that has been created now is not a list but it is of a special class “GDS”

```
class(gds)
```

```
[1] "GDS"
attr(,"package")
[1] "GEOquery"
```

```
slotNames(gds)
```

```
[1] "gpl"       "dataTable" "header"
```

Class 'GDS' is comprised of a metadata header (taken nearly verbatim from the SOFT format header) and a GEODataTable. The GEODataTable has two simple parts, a Columns part which describes the column headers on the Table part. There is also a show method ("Meta") for the class.

```
head(Meta(gds))
```

```
$channel_count
[1] "1"

$dataset_id
[1] "GDS4155" "GDS4155"

$description
[1] "Analysis of induced dopaminergic (iDA) neurons generated from  E14.5 mouse embryonic fil
[2] "dopaminergic-induced"
[3] "control"

$email
[1] "geo@ncbi.nlm.nih.gov"

$feature_count
[1] "35557"

$institute
[1] "NCBI NLM NIH"
```

The gds object can be turned into an expressionSet that contains the same information as in the previous case:

```
eset <- GDS2eSet(gds,do.log2=FALSE)
```

```
Using locally cached version of GPL6246 found here:
C:\Users\asanchez\AppData\Local\Temp\Rtmp6lg5ag/GPL6246.annot.gz
```

```
eset
```

```
ExpressionSet (storageMode: lockedEnvironment)
assayData: 35557 features, 8 samples
  element names: exprs
protocolData: none
phenoData
  sampleNames: GSM671653 GSM671654 ... GSM671660 (8 total)
  varLabels: sample genotype/variation description
  varMetadata: labelDescription
featureData
  featureNames: 10344614 10344616 ... 10344613 (35557 total)
  fvarLabels: ID Gene title ... GO:Component ID (21 total)
  fvarMetadata: Column labelDescription
experimentData: use 'experimentData(object)'
  pubMedIds: 21725324
Annotation:
```

4 Exercises

1. Select a *GEO (*Gene Expression Omnibus) datasetfrom the list presented in the "GEO-datasets_enhanced.xls" document available in the resources of the activity.
2. Read the data from GEO using the GEOquery package. This will provide you with an expressionSet class object with the normalized data and an additional table with information about the study.
3. Determine the structure of the data (rows, columns) and the design of the study (groups of samples or individuals, treatments if any, etc.) that generated them.
   - The information of the experiment can also be downloaded from GEO, either with GEOquery if you provide the dataset identifier GDSxxxx or by accessing the study page.

5 References

- Cui, Dapeng, K. J. Dougherty, DW Machacek, S. Hochman, and D. J Baro. 2006. “Divergence Between Motoneurons: Gene Expression Profiling Provides a Molecular Characterization of Functionally Discrete Somatic and Autonomic Motoneurons.” Physiol Genomics 24 (3): 276–89. https://doi.org/ 10.1152/physiolgenomics.00109.2005.

- 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

- Davis, S., & Meltzer, P. (2007). GEOquery: a bridge between the Gene Expression Omnibus (GEO) and BioConductor. Bioinformatics, 14, 1846–1847.

- W. Huber, V.J. Carey, R. Gentleman, …, M. Morgan. Orchestrating high-throughput genomic analysis with Bioconductor. Nature Methods, 2015:12, 115.

- 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

6 Additional info

```
sessionInfo()
```

```
R version 4.3.0 (2023-04-21 ucrt)
Platform: x86_64-w64-mingw32/x64 (64-bit)
Running under: Windows 11 x64 (build 22631)

Matrix products: default


locale:
[1] LC_COLLATE=Spanish_Spain.utf8  LC_CTYPE=Spanish_Spain.utf8
[3] LC_MONETARY=Spanish_Spain.utf8 LC_NUMERIC=C
[5] LC_TIME=Spanish_Spain.utf8

time zone: Europe/Madrid
tzcode source: internal
```

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

other attached packages:
[1] GEOquery_2.68.0     Biobase_2.60.0      BiocGenerics_0.46.0

loaded via a namespace (and not attached):
 [1] limma_3.56.2      jsonlite_1.8.7    dplyr_1.1.3       compiler_4.3.0
 [5] tidyselect_1.2.0  xml2_1.3.5        tidyr_1.3.0       png_0.1-8
 [9] yaml_2.3.7        fastmap_1.1.1     readr_2.1.4       R6_2.5.1
[13] generics_0.1.3    curl_5.0.2        knitr_1.43        tibble_3.2.1
[17] pillar_1.9.0      tzdb_0.4.0        R.utils_2.12.2    rlang_1.1.1
[21] utf8_1.2.3        xfun_0.40         cli_3.6.1         withr_2.5.0
[25] magrittr_2.0.3    digest_0.6.33     rstudioapi_0.15.0 hms_1.1.3
[29] lifecycle_1.0.3   R.methodsS3_1.8.2 R.oo_1.25.0       vctrs_0.6.3
[33] evaluate_0.21     glue_1.6.2        data.table_1.14.8 fansi_1.0.4
[37] rmarkdown_2.24    purrr_1.0.2       tools_4.3.0       pkgconfig_2.0.3
[41] htmltools_0.5.6
```

```
# An "index.html" file is created to allow visualitzation in the web using github pages
file.copy(from="Introduction_2_Bioc_classes_4_tabular_data.html", to="index.html", overwrite=
```

```
[1] TRUE
```

```
# The R code for the document can be extracted from the document with the
# knitr::purl() command
# knitr::purl("Introduction_2_Bioc_classes_4_tabular_data.qmd")
```