Introduction to Bioconductor class `ExpressionSet`

Alex Sanchez

May 19, 2019

Contents

1 Introduction. Working with Omics data

Omics data are not only high throughput, what means we require big dta matrices to store raw data. They are also complex and, besides numerical data, they often require additional information such as covariates, annotations or technical information required for preprocessing the data.

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.

We start illustrating with a toy example what would be a "standard approach" to managing an omics dataset. Next we introduce the OOP paradigm and a Bioconductor class that allows encapsulating all the informations together and show how it facilitates the process of storing, managing and analyzing omics data.

1.1 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,]  0.34369662  0.78823091 -1.44177165 -0.7266339 0.39270155 -0.5717577
## [2,]  0.08211110  0.20145515 -0.17826983 -0.1991470 0.04898769  0.3044198
## [3,]  1.26830733  0.85615633 -0.31455846 -1.1569649 0.26111225  0.3779398
## [4,] -1.18745479 -0.06958821  0.97127133 -0.1442893 2.54421016  0.7096269
## [5,]  0.09012818  2.80657915  1.33035629 -0.2506080 1.01565316  0.5385987
## [6,] -1.36883880 -0.09024312 -0.02837428  1.6989627 0.97407279  0.7977909
##          sample7    sample8    sample9   sample10
## [1,]  0.74944697  0.5271380  0.5113672  0.3994993
## [2,]  2.09004745  0.7277956 -0.4489498  0.8279574
## [3,]  1.23031729  0.0606610 -0.8256985  0.6623504
## [4,] -1.22395185  1.3703740 -0.4430325 -1.2101024
## [5,] -0.05406224 -1.8356286  0.8337383  0.1933899
## [6,] -1.71415093 -1.8292895  1.6613754  0.2356994
```

```
targets <- data.frame(sampleNames = paste0("sample",1:10), group=c(paste0("CTL",1:5),paste
head(targets, n=10)

##    sampleNames group age    sex
## 1      sample1  CTL1  34 Female
## 2      sample2  CTL2  32   Male
## 3      sample3  CTL3  26 Female
## 4      sample4  CTL4  30   Male
## 5      sample5  CTL5  36 Female
## 6      sample6   TR1  30   Male
## 7      sample7   TR2  27   Male
## 8      sample8   TR3  25 Female
## 9      sample9   TR4  29   Male
## 10    sample10   TR5  30   Male
```

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

```
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"
```

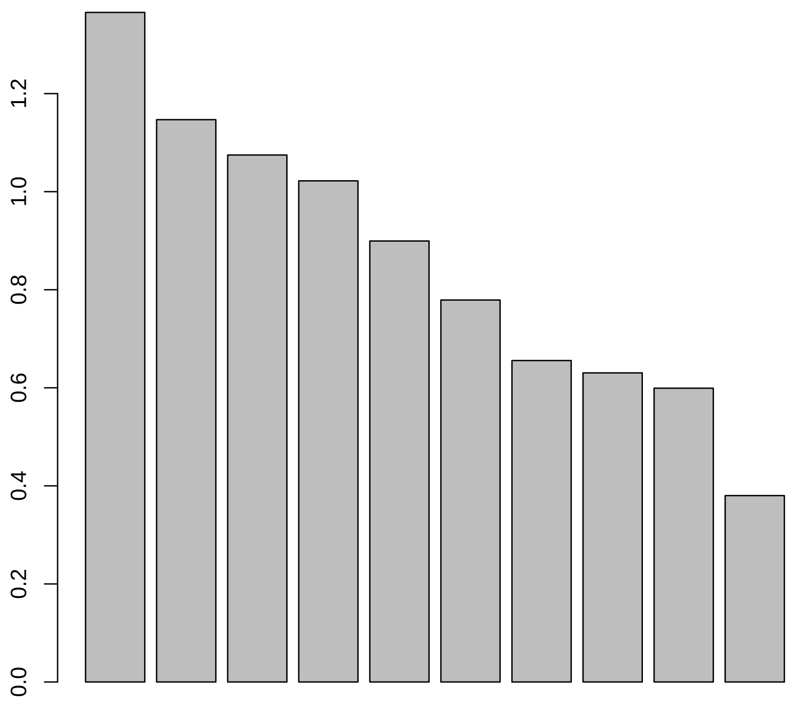
Having data stored in this way is usually enough for most of the analyes we may want to do. The only unconvenient comes from the fact that the information about the same individuals is in separate R objects so that, for certain applications, we will have to access several objects and *assume they are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

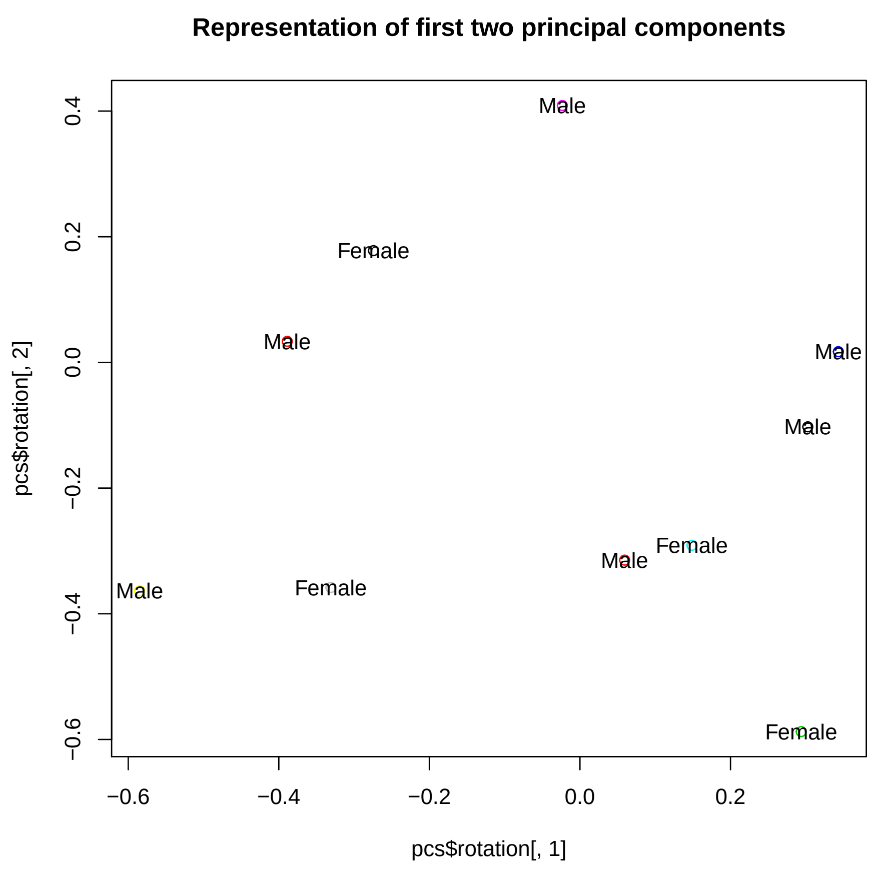
```
pcs <- prcomp(expressionValues)
names(pcs)

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

barplot(pcs$sdev)
```



```
plot(pcs$rotation[,1], pcs$rotation[,2], col=targets$group, main="Representation of first
text(pcs$rotation[,1], pcs$rotation[,2],targets$sex)
```



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] 1.309937 1.256927 1.200237 1.188556 1.139880 1.122894

head(orderedGenes)

## [1] "gene6"  "gene4"  "gene5"  "gene13" "gene17" "gene18"
```

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!

Next section introduces a data structure that allows to encapsulate all these informations together ensuring that the links assumed are true.

2 Bioconductor classes for omics 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 (`http://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 *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 1 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.

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.

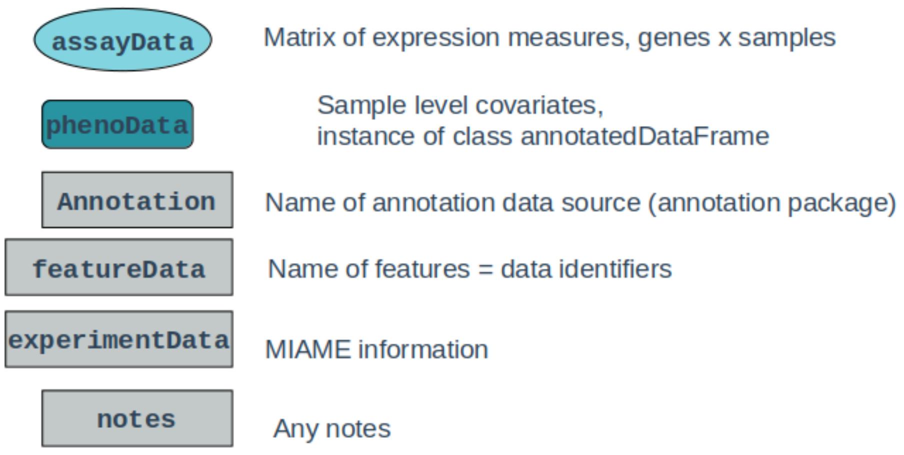

assayData
Matrix of expression measures, genes x samples
phenoData
Sample level covariates,
instance of class annotatedDataFrame
Annotation
Name of annotation data source (annotation package)
featureData
Name of features = data identifiers
experimentData
MIAME information
notes
Any notes

Figure 1: Structure of the `ExpressionSet` class, showing its slots and their meaning

2.4 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 our 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.

2.4.1 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:
```

2.4.2 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 *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("Sample Names", "Treatment/Control", "Age at
myAnnotDF <- new("AnnotatedDataFrame", data=targets, varMetadata= columnDesc)
show(myAnnotDF)

## An object of class AnnotatedDataFrame
##   rowNames: 1 2 ... 10 (10 total)
##   varLabels: sampleNames group age sex
##   varMetadata: labelDescription
```

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=myAnnotDF)
# Error in validObject(.Object) :
#   invalid class ExpressionSet object: 1: sampleNames differ between assayData and phenoD
# invalid class ExpressionSet object: 2: sampleNames differ between phenoData and protocol
```

```
rownames(pData(myAnnotDF))<-pData(myAnnotDF)$sampleNames
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: sampleNames group age sex
##   varMetadata: labelDescription
## featureData: none
## experimentData: use experimentData(object)
## Annotation:
```

2.4.3 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)
```

2.4.4 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 `https://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)
```

2.4.5 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`.

Access Slot values Notice that to access the values we use special fucntions called "accessors" instead of the dollar symbol (which would not work for classe) 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)))
```

```
##          sampleNames group age    sex
## sample1     sample1  CTL1  34 Female
## sample2     sample2  CTL2  32   Male
## sample3     sample3  CTL3  26 Female
## sample4     sample4  CTL4  30   Male
## sample5     sample5  CTL5  36 Female
## sample6     sample6   TR1  30   Male

head(pData(myEset))

##          sampleNames group age    sex
## sample1     sample1  CTL1  34 Female
## sample2     sample2  CTL2  32   Male
## sample3     sample3  CTL3  26 Female
## sample4     sample4  CTL4  30   Male
## sample5     sample5  CTL5  36 Female
## sample6     sample6   TR1  30   Male
```

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 4

head(pData(smallEset))

##          sampleNames group age    sex
## sample1     sample1  CTL1  34 Female
## sample2     sample2  CTL2  32   Male
## sample3     sample3  CTL3  26 Female
## sample6     sample6   TR1  30   Male
## sample7     sample7   TR2  27   Male
## sample8     sample8   TR3  25 Female

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  4

head(pData(youngEset))

##         sampleNames group age    sex
## sample3     sample3  CTL3  26 Female
## sample7     sample7   TR2  27   Male
## sample8     sample8   TR3  25 Female
## sample9     sample9   TR4  29   Male
```

2.4.6 Exercise

1. Select a GEO dataset and prepare, from it, the components we have seen in the sections above, that is: The expression values, in a matrix or data.frame, the targets in a data frame, the experiment description, and information about annotations and gene names (you may obtain these from the matrix rownames).

2. Proceed as above and create first the pieces needed to create the `ExpressionSet` and then an object of class `ExpressionSet` with all the data and its information.

3. Reproduce the data exploration done in the first exercise accessing the data through the `ExpressionSet`.

4. Do some subsetting and check the consistency of the results obtained.

5. Add these steps to a new section in your "Exercise 1" document. Render the new document and when you are satisfied with it update your giyhub repository.

3 The `GEOquery` package

3.1 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.

3.2 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("GSE507")
class(gse)

## [1] "list"

names(gse)

## [1] "GSE507_series_matrix.txt.gz"

gse[[1]]

## ExpressionSet (storageMode: lockedEnvironment)
## assayData: 7700 features, 3 samples
##   element names: exprs
## protocolData: none
## phenoData
##   sampleNames: GSM1128 GSM1129 GSM1130
##   varLabels: title geo_accession ... data_row_count (29 total)
##   varMetadata: labelDescription
## featureData
##   featureNames: AAAAAAAAAA AAAAAAAAAC ... TTTTTGTGAA (7700 total)
##   fvarLabels: TAG GI
##   fvarMetadata: Column Description labelDescription
## experimentData: use experimentData(object)
##   pubMedIds: 11850811
## Annotation: GPL4

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.

3.2.1 Exercise

1. Use the `getGEO` command to create an `ExpressionSet` for the dataset you used in the previous exercise. Notice that the object needed is within a list so you need to access to it using the operator.

2. Once you have created it reproduce what you did there with your data.

3. Again, render the document and update your Exercise 1 repository.