RNA-seq data mining & differential analysis using R

DU Bii - N. Servant / M. Deloger

24th March 2021

Contents

How to generate a count table at the gene level ?

A count table represents the number of reads mapped per gene/transcripts in an RNA-seq experiment. This is the entry point of many downstream supervised or unsupervised analysis.

There are many different ways to generate a count table, and many tools can be used.
Usually, generating such table requires two mains steps :

1. Aligning the reads on a reference genome
2. Counting how many reads can be assigned to a given gene

The mapping step aims at positioning the sequencing reads on your reference genome. Different tools such as HiSat2[1], STAR[2], etc. are still commonly used. In theory, if well configured, these tools should give close results, although their mapping strategy and computational requirements might be different. Of note, recent

[1]Kim D, Langmead B and Salzberg SL. (2015) HISAT: a fast spliced aligner with low memory requirements. Nature Methods
[2]Dobin A., Davis C.A., Schlesinger F. et al. (2013) STAR: ultrafast universal RNA-seq aligner, Bioinformatics, 29(1):15–21,

methods/tools based on pseudo-mapping approaches such as Salmon[3], Kallisto[4], Rapmap[5], etc. can also be used to quantify the gene expression from raw RNA-seq data (see *Bray et al. 2016* [6]).

Once the data are mapped on the genome, several tools can be used to count and assign reads to a given gene (exons).
Among the most popular tools, HTSeqCount[7] or FeatureCounts[8] are frequently used. Note that for this step, it is crucial to have details on the protocol used to generate the samples, and especially if the protocol was **stranded** or not.
This step also requires some gene annotations. Databases such as Ensembl, Refseq, or Gencode can be used. They all contain the most common coding genes but they also all have their own specifities.

To wrap up, here is an example of a typical RNA-seq workflow for gene expression profiling.

Before starting

Please be sure to have a recent version of R (>3.3) with the following packages from the CRAN :

- knitr
- pheatmap
- FactoMineR
- factoextra
- reshape2
- ggplot2
- RColorBrewer

From BioConductor :

- DESeq2
- edgeR
- limma
- rtracklayer
- GenomicFeatures
- [org.Hs.eg.db]
- [clusterProfiler]

In order to install these packages, use the following command :

```
## For CRAN packages
install.packages("knitr")

## For BioC packages
if (!requireNamespace("BiocManager", quietly = TRUE))
    install.packages("BiocManager")

BiocManager::install("DESeq2")
```

[3]Patro, R., Duggal, G., Love, M. I. et al. (2017). Salmon provides fast and bias-aware quantification of transcript expression. Nature Methods.

[4]Nicolas L Bray N.L., Pimentel H., Melsted P. et al. (2016) Near-optimal probabilistic RNA-seq quantification, Nature Biotechnology 34, 525–527

[5]Srivastava A., Sarkar H., Gupta N. et al. (2016) RapMap: a rapid, sensitive and accurate tool for mapping RNA-seq reads to transcriptomes, Bioinformatics. 32(12)

[6]Bray N.L. et al. (2016) Near-optimal probabilistic RNA-seq quantification. Nature Biotech., 34(5):525–527.

[7]Anders S., Pyl T.P., Huber W. (2015) HTSeq - A Python framework to work with high-throughput sequencing data. Bioinformatics 31(2):166-9

[8]Liao Y, Smyth GK and Shi W. (2014) featureCounts: an efficient general-purpose program for assigning sequence reads to genomic features. Bioinformatics, 30(7):923-30

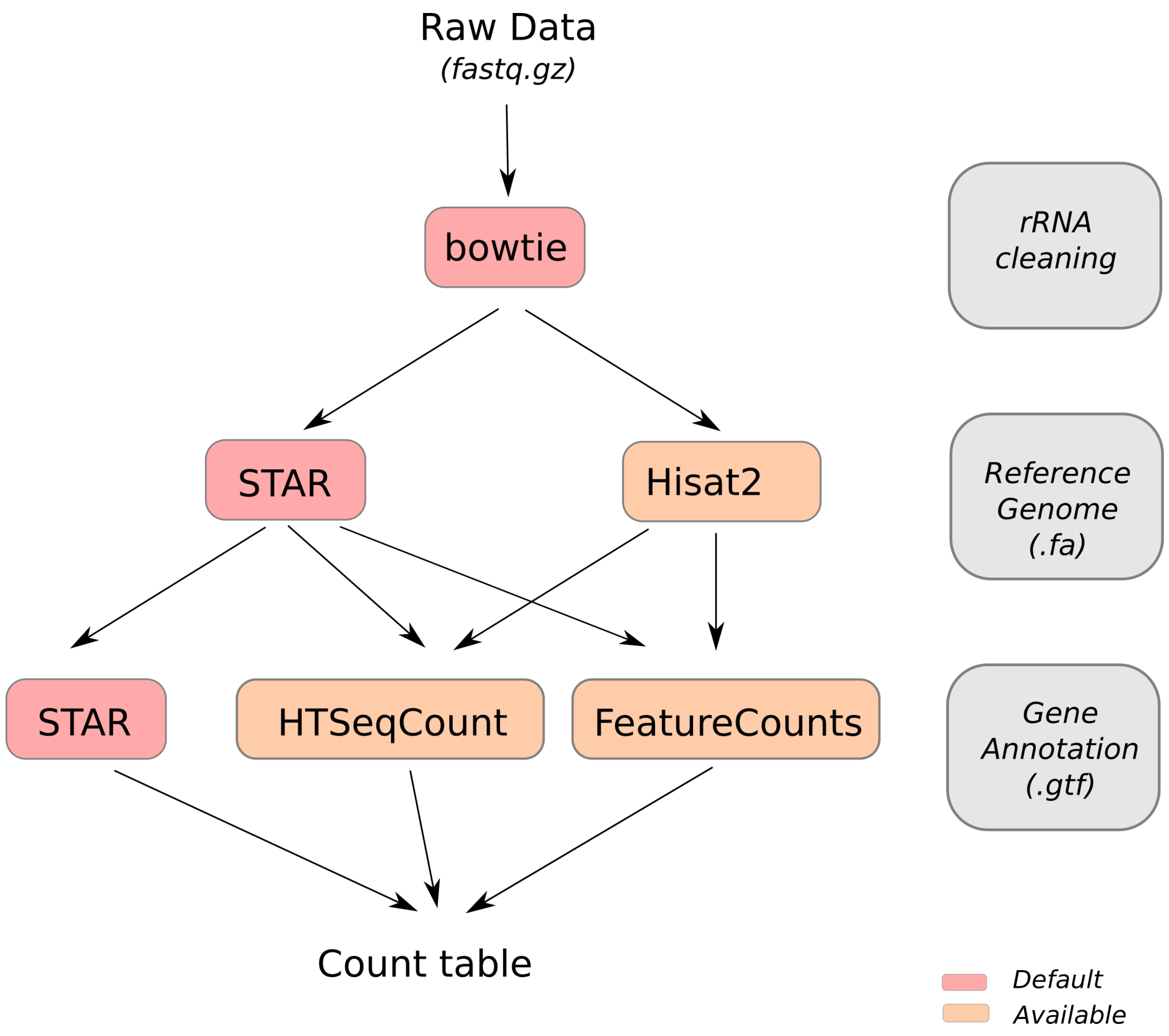

Raw Data
(fastq.gz)
bowtie
rRNA cleaning
STAR
Hisat2
Reference Genome (.fa)
STAR
HTSeqCount
FeatureCounts
Gene Annotation (.gtf)
Count table
Default
Available

Figure 1:

Reminder about R . . .

Help in R

Google is your best friend ! But a lot is already available within R . . .

```
## Example of help in R
## Use "?" followed by a function name
?read.csv
```

'Vector' in R

R is working in vectorial mode. A `vector` is a one-dimensional object accessible through its indices.

Examples

```
## 'a' is a vector from 10 to 20
## It could also be noted as
## a <- c(10, 11, 12, 14, 15, 16, 17, 18, 19, 20)
a <- 10:20
a
```

```
##  [1] 10 11 12 13 14 15 16 17 18 19 20
```

```
## Access to a value
a[3]
```

```
## [1] 12
```

```
## Sum
sum(a)
```

```
## [1] 165
```

```
## Operation of vector
a+1
```

```
##  [1] 11 12 13 14 15 16 17 18 19 20 21
```

```
## Which indices within my vector are higher that 5
which(a>5)
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11
```

```
## Therefore, what are the values corresponding to these indexes
a[which(a>5)]
```

```
##  [1] 10 11 12 13 14 15 16 17 18 19 20
```

'Matrix' in R

A table or `matrix` is a n x n array of similar data type (character or numeric). It therefore contains two coordinates [x, y] where x represents the raws and y the colums. A `matrix` is characterized by its values, but also its colnames and rownames.

Examples

```
## Let's create a simple matrix in R
## You do not know what 'rnorm' is doing ?
## ?rnorm
```

```
d <- matrix(1:50, ncol=5, nrow=10)
d
```

```
##       [,1] [,2] [,3] [,4] [,5]
##  [1,]    1   11   21   31   41
##  [2,]    2   12   22   32   42
##  [3,]    3   13   23   33   43
##  [4,]    4   14   24   34   44
##  [5,]    5   15   25   35   45
##  [6,]    6   16   26   36   46
##  [7,]    7   17   27   37   47
##  [8,]    8   18   28   38   48
##  [9,]    9   19   29   39   49
## [10,]   10   20   30   40   50
```

```
## Access to the first column
d[,1]
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```
## Access to the first row
d[1,]
```

```
## [1]  1 11 21 31 41
```

```
## Access to the value at 3 rows and 4 columns
d[3,4]
```

```
## [1] 33
```

```
## Making a sum for each raw
rowSums(d)
```

```
##  [1] 105 110 115 120 125 130 135 140 145 150
```

```
## Making a sum for each cols
colSums(d)
```

```
## [1]  55 155 255 355 455
```

```
## How many values in the matrix are higher than 5
length(which(d>1))
```

```
## [1] 49
```

```
## The matrix we just created does not have any col/rownames
rownames(d)
```

```
## NULL
```

```
colnames(d)
```

```
## NULL
```

```
## Let's add some colnames
colnames(d) <- c("A", "B", "C", "D", "E")
d
```

```
##        A  B  C  D  E
##  [1,]  1 11 21 31 41
##  [2,]  2 12 22 32 42
##  [3,]  3 13 23 33 43
##  [4,]  4 14 24 34 44
##  [5,]  5 15 25 35 45
##  [6,]  6 16 26 36 46
##  [7,]  7 17 27 37 47
##  [8,]  8 18 28 38 48
##  [9,]  9 19 29 39 49
## [10,] 10 20 30 40 50
```

```
colnames(d)
```

```
## [1] "A" "B" "C" "D" "E"
```

'dataframe' in R

A `dataframe` is used to store data tables. It is a list of vectors of equal length. A `dataframe` is more general than a matrix, in the sense that different columns can have different modes (numeric, character, factor, etc.).

Examples

```
df <- data.frame(mycol=letters[1:nrow(d)], d)
df
```

```
##    mycol  A  B  C  D  E
## 1      a  1 11 21 31 41
## 2      b  2 12 22 32 42
## 3      c  3 13 23 33 43
## 4      d  4 14 24 34 44
## 5      e  5 15 25 35 45
## 6      f  6 16 26 36 46
## 7      g  7 17 27 37 47
## 8      h  8 18 28 38 48
## 9      i  9 19 29 39 49
## 10     j 10 20 30 40 50
```

```
class(df)
```

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

```
## The as.matrix (resp. as.dataframe) allows to convert my object from a dataframe to a matrix
## But note that in this case, all values will be convert into character
as.matrix(df)
```

```
##       mycol A    B    C    D    E
##  [1,] "a"   " 1" "11" "21" "31" "41"
##  [2,] "b"   " 2" "12" "22" "32" "42"
##  [3,] "c"   " 3" "13" "23" "33" "43"
##  [4,] "d"   " 4" "14" "24" "34" "44"
##  [5,] "e"   " 5" "15" "25" "35" "45"
##  [6,] "f"   " 6" "16" "26" "36" "46"
##  [7,] "g"   " 7" "17" "27" "37" "47"
##  [8,] "h"   " 8" "18" "28" "38" "48"
##  [9,] "i"   " 9" "19" "29" "39" "49"
## [10,] "j"   "10" "20" "30" "40" "50"
```

```
## Removing the first column allows to get only one data type and to come back to numeric values
as.matrix(df[,-1])
```

```
##        A  B  C  D  E
##  [1,]  1 11 21 31 41
##  [2,]  2 12 22 32 42
##  [3,]  3 13 23 33 43
##  [4,]  4 14 24 34 44
##  [5,]  5 15 25 35 45
##  [6,]  6 16 26 36 46
##  [7,]  7 17 27 37 47
##  [8,]  8 18 28 38 48
##  [9,]  9 19 29 39 49
## [10,] 10 20 30 40 50
```

How to check the class of my object ?

The `class` and `typeof` functions allow to get information about the nature of an object and its data type.

Examples

```r
class(d)
```

```
## [1] "matrix"
```

```r
is.matrix(d)
```

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

```r
is.data.frame(d)
```

```
## [1] FALSE
```

Functions in R

As in any programming langage, R allows the user to write functions. A function is defined by:

- A name
- One or several argument(s)
- A `return` value if needed

Using functions is strongly adviced to factorize a code as much as possible, and to limit coding errors.

Examples

```r
myfunc <- function(a, b=1){
  ## do something
  res <- a^2 + b
  ## return the results
  return(res)
}

myfunc(10)
```

```
## [1] 101
```

```r
myfunc(a=10, b=5)
```

```
## [1] 105
```

Loop in R

The 'for' loop

The `for` loop system is common to many langage. It allows a variable to be updated in an iterative way. In the following example, the variable `i` will be replaced by `1, 2, 3, 4, ..., 10`

Examples

```
for (i in c(1:10)){
    myfunc(i)
}
```

However, this looping system is usually not recommanded in R, as it is quite slow . . . Instead, R offers the `apply()` family function.

The Apply functions

The `apply()` family function is a set of functions to manipulate and loop on data structure such as matrices, dataframes, lists, etc. In practice, the family is made up of `apply()`, `lapply()`, `sapply()`, `vapply()`, `mapply()` ,`rapply()`, `tapply()`.
So far, we will just focus on the `apply()` function that can be called on matrices and dataframes where :

- X is a matrix
- MARGIN is a variable that define rows (MARGIN=1) and/or columns (MARGIN=2)
- FUN is a function
- . . . , any parameters that can be passed to the function

Examples

```
## Loop over all columns of 'd' and run the 'sum' function
apply(d, 2, sum)
```

```
##   A   B   C   D   E
##  55 155 255 355 455
```

```
## Loop over all rows of 'd' and run the 'sum' function
apply(d, 1, sum)
```

```
##  [1] 105 110 115 120 125 130 135 140 145 150
```

Matching values in R

One of the most common operation in R is to be able to match two vectors.
The `match` operation returns a vector of the positions of (first) matches of its first argument in its second.
`%in%` is built as a binary operator, and indicates if there is a match or not between the two arguments.

Examples

```
a <- c(-100, -50, -64, 20, 65, 126)

## Get indices of values > 0
which(a>0)
```

```
## [1] 4 5 6
```

```
## Get values > 0
a[which(a>0)]
```

```
## [1]  20  65 126
```

```
a <- c("A","B","C","D","E","F")
b <- c("B", "A", "E")
## match 'b' values into the 'a' vector - return the position
match(b, a)
```

```
## [1] 2 1 5
```

```
## %in% can be an alternative to which for vector comparison
a %in% b
```

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

FAIR Guiding Principles in R

It is usually recommanded to follow good programming practices when you are writting your own code. Among them, pay attention to :

- Use an editor (such as RStudio) to write and save your code
- Use appropriate variable names
- Use functions
- Add comments into your code
- Do not hesitate the write a command in multiple lines
- Make simple tests
- Never use a command line that you do not understand
- Add a `sessionInfo()` at the end of each script

If you want to go further :

- Use a versioning system (git)
- Make your code reproducible with the `renv` package.

```
library(renv)
renv::init() ## Init a new project locally
## Install your packages
renv::snapshot() ## Save the state of the project library in a lockfile
renv::restore() ## Restore your local environment defined in the lockfile
```

Loading a count table

Experimental data

As a toy dataset, we will use the data published by *Horvath et al.* [9] available on GEO (GSE52194). This dataset is composed of `20 breast samples` :

- 5 HER2+ samples (ER and PR negative)
- 6 Triple negative (ER, PR, HER2 negative)
- 6 non-TNBC (ER, PR and HER2-positive)
- 3 normal-like breast samples

All these samples were processed using the pipeline presented above to generate the count table. Data are available on the cluster in the folder `/shared/projects/dubii2020/data/rnaseq/countsdata`.

[9] Kim D, Langmead B and Salzberg SL. (2015) HISAT: a fast spliced aligner with low memory requirements. Nature Methods

Importing data

Usually, the easiest way would be to start the analysis from a raw count table. If you do not have access to such table and have the STAR gene counts output, the following functions can be used to build the raw contact matrix.

```
starload <- function(input_path, col.nb){
  message("loading STAR gene counts ...")
  exprs.in <- list.files(path=input_path,
                         pattern="ReadsPerGene.out.tab",
                         full.names=TRUE,recursive=TRUE)
  counts.exprs <- lapply(exprs.in, read.csv, sep="\t", header=FALSE,
                         row.names=1, check.names=FALSE)
  counts.exprs <- data.frame(lapply(counts.exprs, "[", col.nb))
  colnames(counts.exprs) <- basename(exprs.in)
  ## remove first 4 lines
  counts.exprs <- counts.exprs[5:nrow(counts.exprs), , drop=FALSE]
  counts.exprs
}
```

R can easily load .csv files. The CSV (Comma Separated Values) format is fully compatible with Excel. It is a plain text format, where columns are separated by a `comma`.

```
proj <- "/shared/projects/dubii2021/trainers/module5/seance5/data/counts/"
proj <- "../data/counts"
## Load a raw count table from a csv file
d <- read.csv(file.path(proj, "tablecounts_raw.csv"), row.names = 1)
d <- as.matrix(d)
## Load TPM normalized count table from a csv file
d.tpm  <- read.csv(file.path(proj, "tablecounts_tpm.csv"), row.names = 1)
d.tpm <- as.matrix(d.tpm)

## Loading sample plan
splan <- read.csv(file.path(proj, "SAMPLE_PLAN"), row.names=1, header=FALSE)
colnames(splan) <- c("sname","subtype")
splan
```

```
##                sname subtype
## SRR1027171    TNBC1    TNBC
## SRR1027172    TNBC2    TNBC
## SRR1027173    TNBC3    TNBC
## SRR1027174    TNBC4    TNBC
## SRR1027175    TNBC5    TNBC
## SRR1027176    TNBC6    TNBC
## SRR1027177 NonTNBC1 NonTNBC
## SRR1027178 NonTNBC2 NonTNBC
## SRR1027179 NonTNBC3 NonTNBC
## SRR1027180 NonTNBC4 NonTNBC
## SRR1027181 NonTNBC5 NonTNBC
## SRR1027182 NonTNBC6 NonTNBC
## SRR1027183   HER2_1    HER2
## SRR1027184   HER2_2    HER2
## SRR1027185   HER2_3    HER2
## SRR1027186   HER2_4    HER2
## SRR1027187   HER2_5    HER2
## SRR1027188     NBS1 Control
## SRR1027189     NBS2 Control
## SRR1027190     NBS3 Control
```

```
## Update colnames of my count tables
colnames(d) <- as.character(splan[colnames(d),"sname"])
colnames(d.tpm) <- as.character(splan[colnames(d.tpm),"sname"])
d[1:5,1:5]
```

```
##                                TNBC1 TNBC2 TNBC3 TNBC4 TNBC5
## ENSG00000223972.4|DDX11L1          0     0     0     0     1
## ENSG00000227232.4|WASH7P          16     5    16    26    26
## ENSG00000243485.2|MIR1302-11       1     0     0     0     0
## ENSG00000237613.2|FAM138A          0     0     0     0     0
## ENSG00000268020.2|OR4G4P           0     0     1     0     0
```

Data mining: playing with a count table

R is extremely powerfull and is a perfect tool to explore your data. Here are a few examples of questions you may want to address:

- How many samples/genes do I have in my count table ?

Show

```
## How many samples/genes ?
dim(d)
```

```
## [1] 57820    20
```

- How many reads (ie. raw counts) per sample ?

Show

```
## How many reads do I have per sample
colSums(d)
```

```
##    TNBC1    TNBC2    TNBC3    TNBC4    TNBC5    TNBC6 NonTNBC1 NonTNBC2
## 10536204  3014694  7963296  9233649  8820837  9245006 21299507 11956159
## NonTNBC3 NonTNBC4 NonTNBC5 NonTNBC6   HER2_1   HER2_2   HER2_3   HER2_4
## 12774570 12045260 13350516  9897833 11164046 15483794 11256553 12441938
##   HER2_5     NBS1     NBS2     NBS3
##  5746612 23629818 24802379 29812185
```

- How many genes have zero count in all samples ?

Show

```
## How many genes have zero counts in all samples
rs <- rowSums(d)
nbgenes_at_zeros <- length(which(rs==0))
nbgenes_at_zeros
```

```
## [1] 3689
```

- For each sample, how many genes have more than one count ?

Show

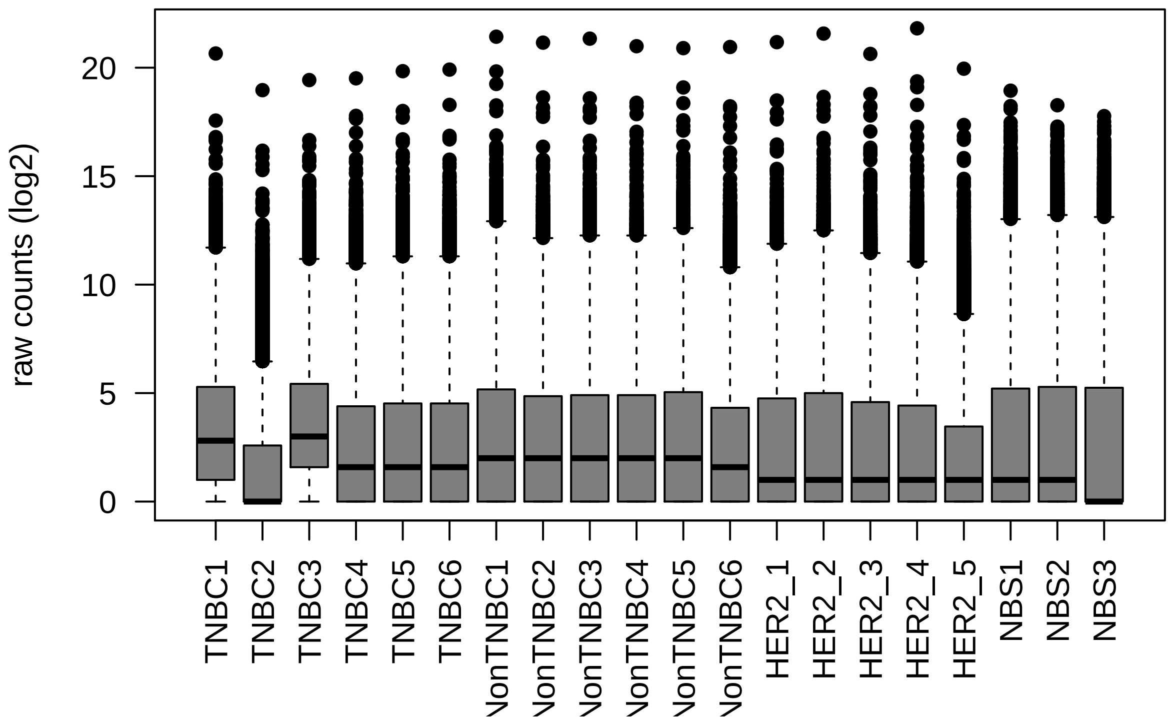
```
## For each sample, how many genes have more than one count ?
number_expressed <- function(x, mincounts=1){
    nb <- length(which(x>mincounts))
    return(nb)
}
nbgenes_per_sample <- apply(d, 2, number_expressed)
nbgenes_per_sample
```

```
##     TNBC1     TNBC2     TNBC3     TNBC4     TNBC5     TNBC6 NonTNBC1 NonTNBC2
##     42316     19338     43913     29067     31222     32807     33237     34484
## NonTNBC3 NonTNBC4 NonTNBC5 NonTNBC6    HER2_1    HER2_2    HER2_3    HER2_4
##     35270     35839     35819     29013     28711     28852     27723     26966
##    HER2_5      NBS1      NBS2      NBS3
##     22389     25490     24736     24107
```

- Draw the Log2 raw counts distribution per sample

Show

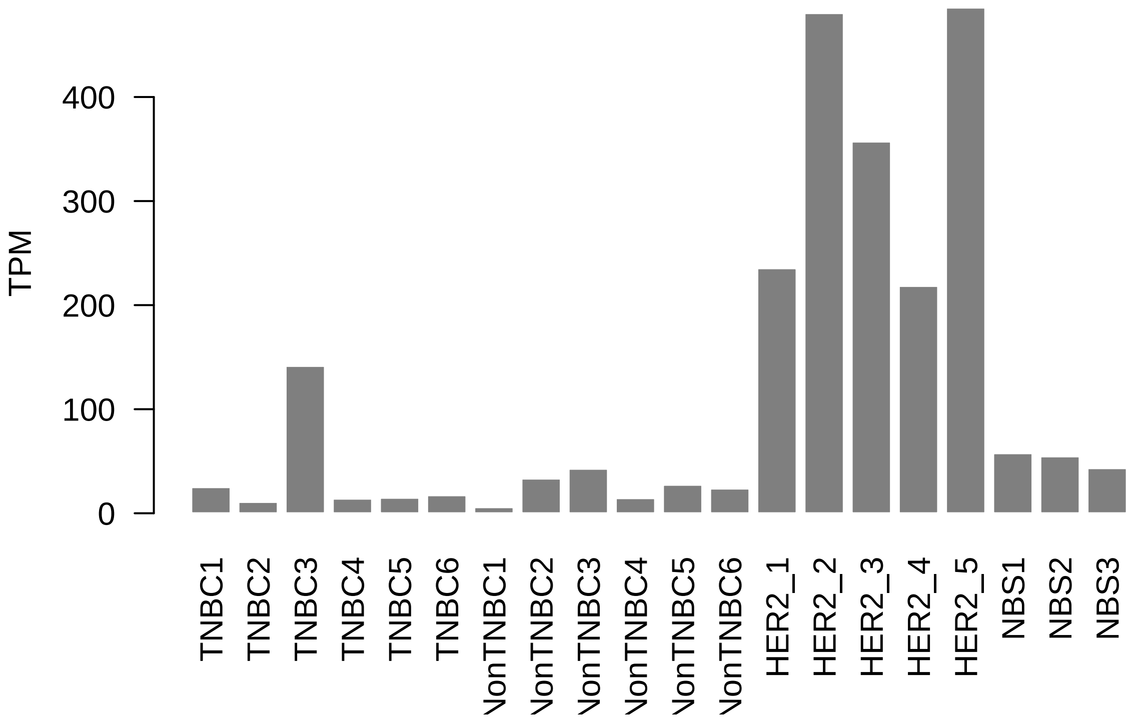
```
## Distribution of raw counts (log2)
boxplot(log2(1+d),
        las=2, ylab="raw counts (log2)", col="gray50", pch=16)
```



- Look at gene "ENSG00000141736.9|ERBB2" in my TPM normalized counts table

Show

```
## if you no want to use grep
## grep("ERBB2$", rownames(d.tpm))
erbb2_pos = which(rownames(d.tpm)=="ENSG00000141736.9|ERBB2")

## Looking at the expression level of my favorite gene
barplot(d.tpm[erbb2_pos, ], ylab="TPM", las=2, col="gray50", border="white")
```



- How many genes are expressed (TPM > 1) in my data ?

Show

```
getExpressedNumber <- function(x){
  n <- length(which(x>1))
  return(n)
}
nb_expressed_genes <- apply(d.tpm, 2, getExpressedNumber)
nb_expressed_genes
```

```
##    TNBC1    TNBC2    TNBC3    TNBC4    TNBC5    TNBC6 NonTNBC1 NonTNBC2
##    35439    18249    42477    20245    22206    24195    16873    21891
## NonTNBC3 NonTNBC4 NonTNBC5 NonTNBC6   HER2_1   HER2_2   HER2_3   HER2_4
##    22015    23389    21604    19178    19896    20099    19404    16174
##   HER2_5     NBS1     NBS2     NBS3
##    17528    18963    19101    18067
```

- Calculate the mean expression of all genes over TNBC and NonTNBC samples

Show

```
tnbc_mean <- rowMeans(d.tpm[,c("TNBC1","TNBC2", "TNBC3", "TNBC4", "TNBC5", "TNBC6")])
nontnbc_mean <- rowMeans(d.tpm[,c("NonTNBC1","NonTNBC2", "NonTNBC3", "NonTNBC4", "NonTNBC5", "NonTNBC6"
```

- Calculate the log2 fold-change of my mean TNCB / NonTNBC samples

Show

```
## Calculate logFC
fc <- log2(1 + tnbc_mean) - log2(1 + nontnbc_mean)

## Order by Fold Changes
fc.ord <- fc[order(fc, decreasing=TRUE)]
```

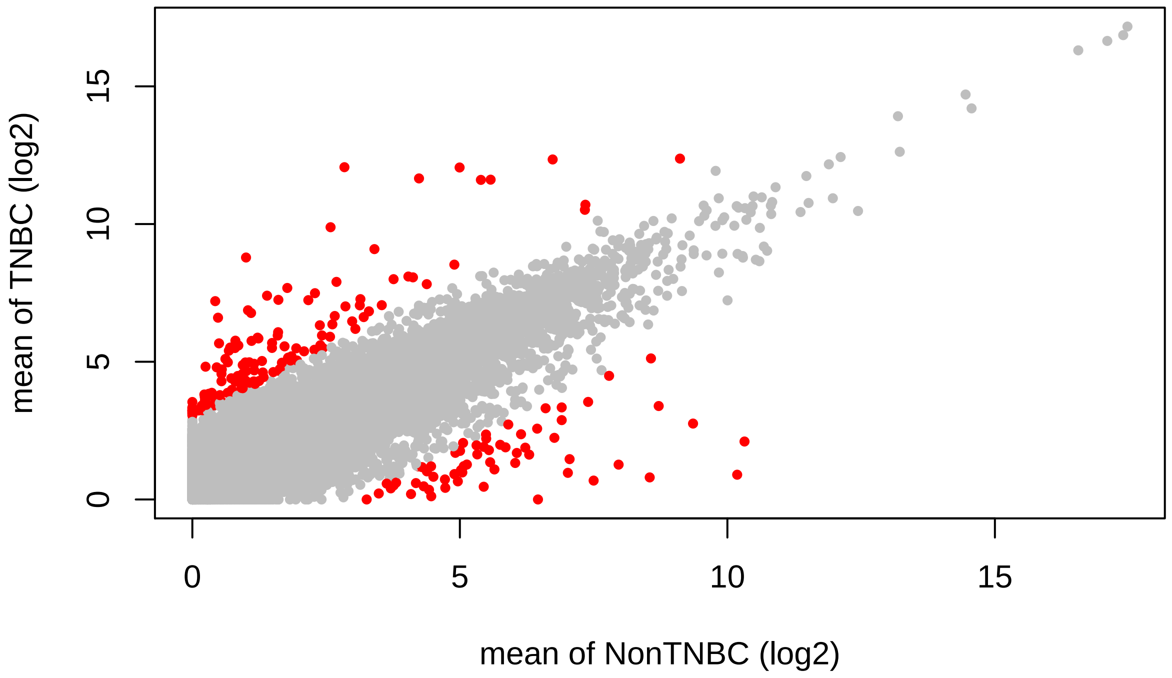
```
## Top 10 genes with higher fold change
head(fc.ord, 10)
```

```
## ENSG00000232216.1|IGHV3-43    ENSG00000181617.5|FDCSP
##                   9.221682                   7.781491
## ENSG00000240382.1|IGKV1-17 ENSG00000211967.2|IGHV3-53
##                   7.420509                   7.301344
## ENSG00000211955.2|IGHV3-33     ENSG00000171209.3|CSN3
##                   7.058050                   6.772671
## ENSG00000211663.2|IGLV3-19 ENSG00000211972.2|IGHV3-66
##                   6.210281                   6.118904
## ENSG00000211666.2|IGLV2-14     ENSG00000167754.8|KLK5
##                   6.036707                   6.002069
```

- Display the two sample types using a scatter plot with the genes with a logFC>3 in red

Show

```
## Scatter plot
plot(x=log2(nontnbc_mean + 1), y=log2(tnbc_mean + 1), col=ifelse(abs(fc)>3, "red", "gray"),
     pch=16, cex=.7,
     xlab="mean of NonTNBC (log2)", ylab="mean of TNBC (log2)")
```



- Extract the 10 first genes with the highest expression in TNBC samples

Show

```
od <- order(tnbc_mean, decreasing=TRUE)
names(tnbc_mean[od[1:10]])
```

```
##  [1] "ENSG00000210082.2|MT-RNR2"    "ENSG00000265150.1|RN7SL2"
##  [3] "ENSG00000258486.2|RN7SL1"     "ENSG00000202198.1|RN7SK"
##  [5] "ENSG00000259001.2|RPPH1"      "ENSG00000269900.2|RMRP"
##  [7] "ENSG00000239776.2|AC079949.1" "ENSG00000211459.2|MT-RNR1"
##  [9] "ENSG00000241781.2|AL161626.1" "ENSG00000211896.2|IGHG1"
```

Exploratory analysis

Exploratory analysis assesses overall similarity between samples without any prior knowledge: which samples are similar to each other? Does this fit to the experimental design? is there any outlier samples?

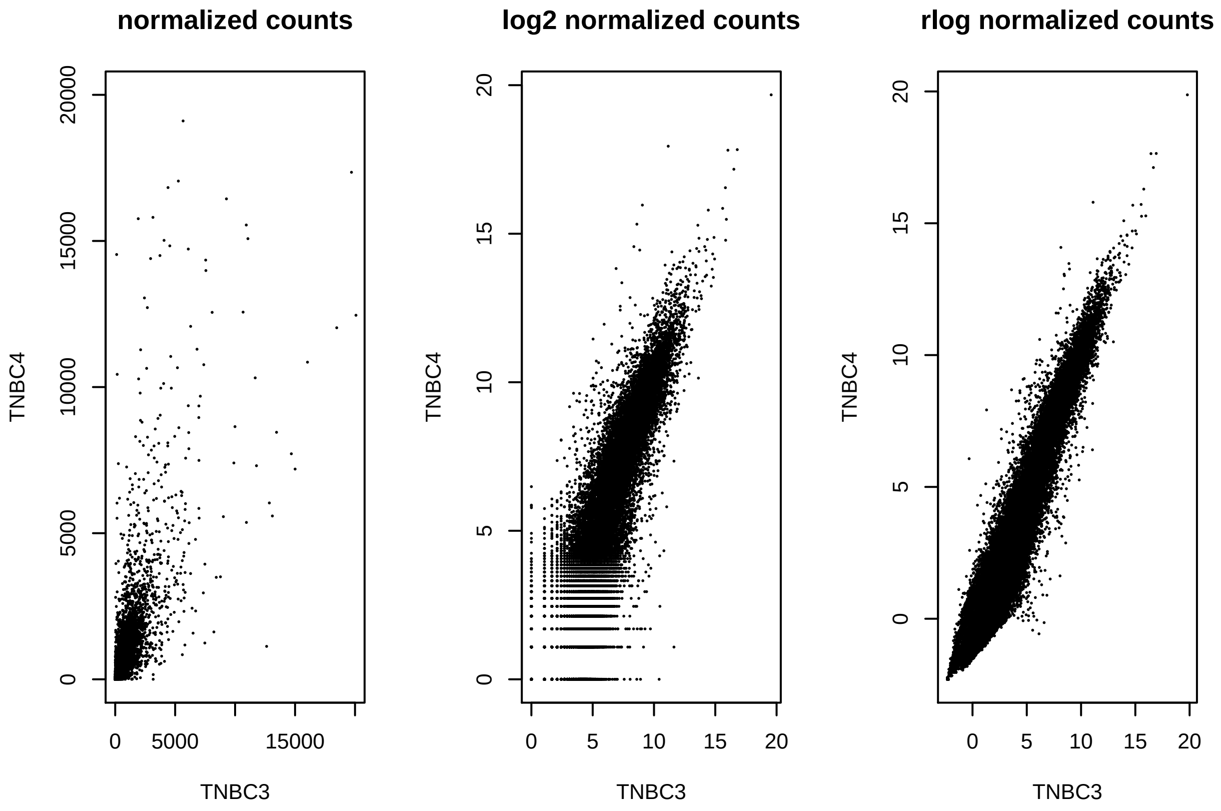
TPM and RPKM are `units` that can be used to look at transcript abundance. However, they do not perform robust cross-sample normalization. Methods as implemented in the limma, DESeq2 or edgeR packages propose to normalize by sequencing depth by calculing scaling factors using more sophisticated approaches. DESeq2 defines a virtual reference sample by taking the median of each gene's values across samples and then computes size factors as the median of ratios of each sample to the reference sample. For details about how DESeq2 calculates its size factors, see here.

Variance Stabilization

To avoid that the distance measure between samples was dominated by a few highly variable genes, and have a roughly equal contribution from all genes, it is recommanded to use a normalization approach that `stabilize the variance` accross expression level, leading to near-homoskedastic data (i.e. the variance of the gene expression does not depend on the mean).

In RNA–Seq data, the variance grows with the mean. For example, if we run the PCA directly on a matrix of counts, the result will mainly depends on the few most strongly expressed genes because they show the largest absolute differences between samples. A simple strategy to avoid this is to take the log of the counts. However, now the genes with low counts tend to dominate the results because they show the strongest relative differences between samples.
Therefore, transformations that stabilize the variance over the mean are advised.
Here, we will use the `rlog` method from the DESeq2 package.

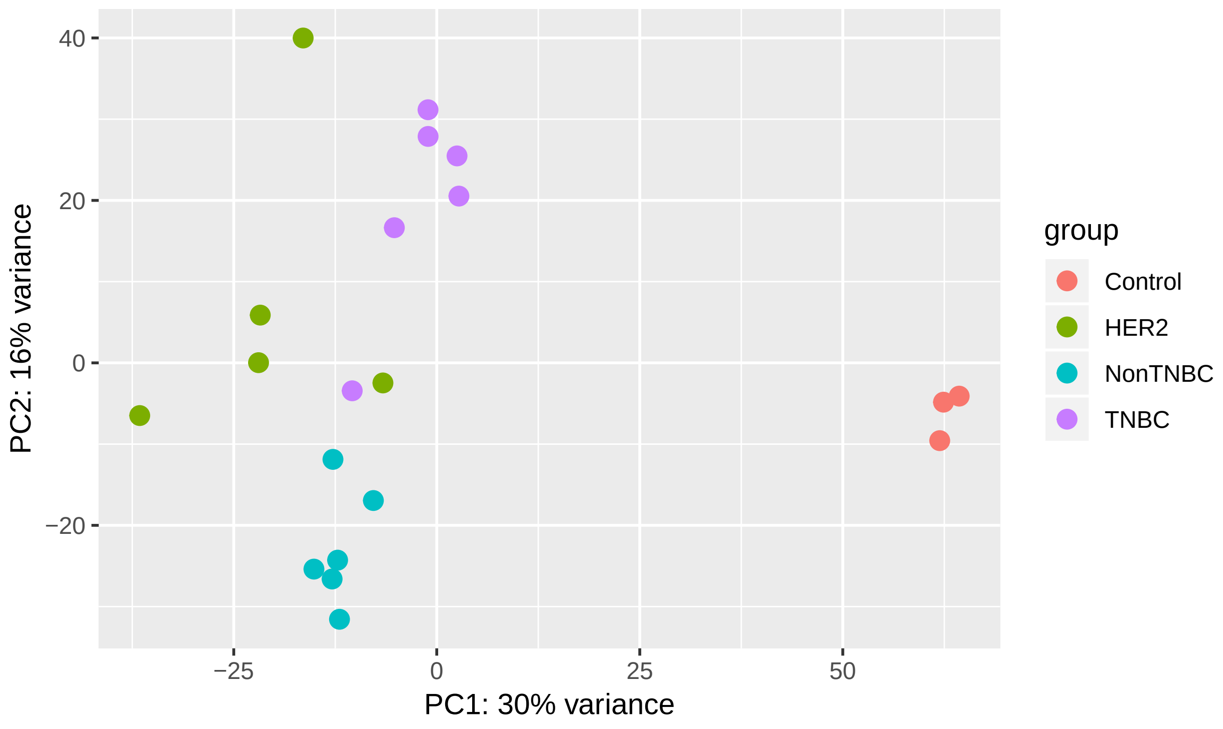
```r
library(DESeq2)
## Load data
dds <- DESeqDataSetFromMatrix(countData=d, DataFrame(condition=splan$subtype), ~ condition)

## Estimate size factors
dds <- estimateSizeFactors(dds)

## Remove lines with only zeros
dds <- dds[ rowSums(counts(dds)) > 0, ]

## Run the rlog normalization
rld <- rlog(dds, blind=TRUE)

## r rlog
par(mfrow=c(1,3))
plot(counts(dds, normalized=TRUE)[,3:4],
     pch=16, cex=0.3, xlim=c(0,20e3), ylim=c(0,20e3), main="normalized counts")
plot(log2(counts(dds, normalized=TRUE)[,3:4] + 1),
     pch=16, cex=0.3, main="log2 normalized counts")
plot(assay(rld)[,3:4],
     pch=16, cex=0.3, main="rlog normalized counts")
```



Principal Component Analysis

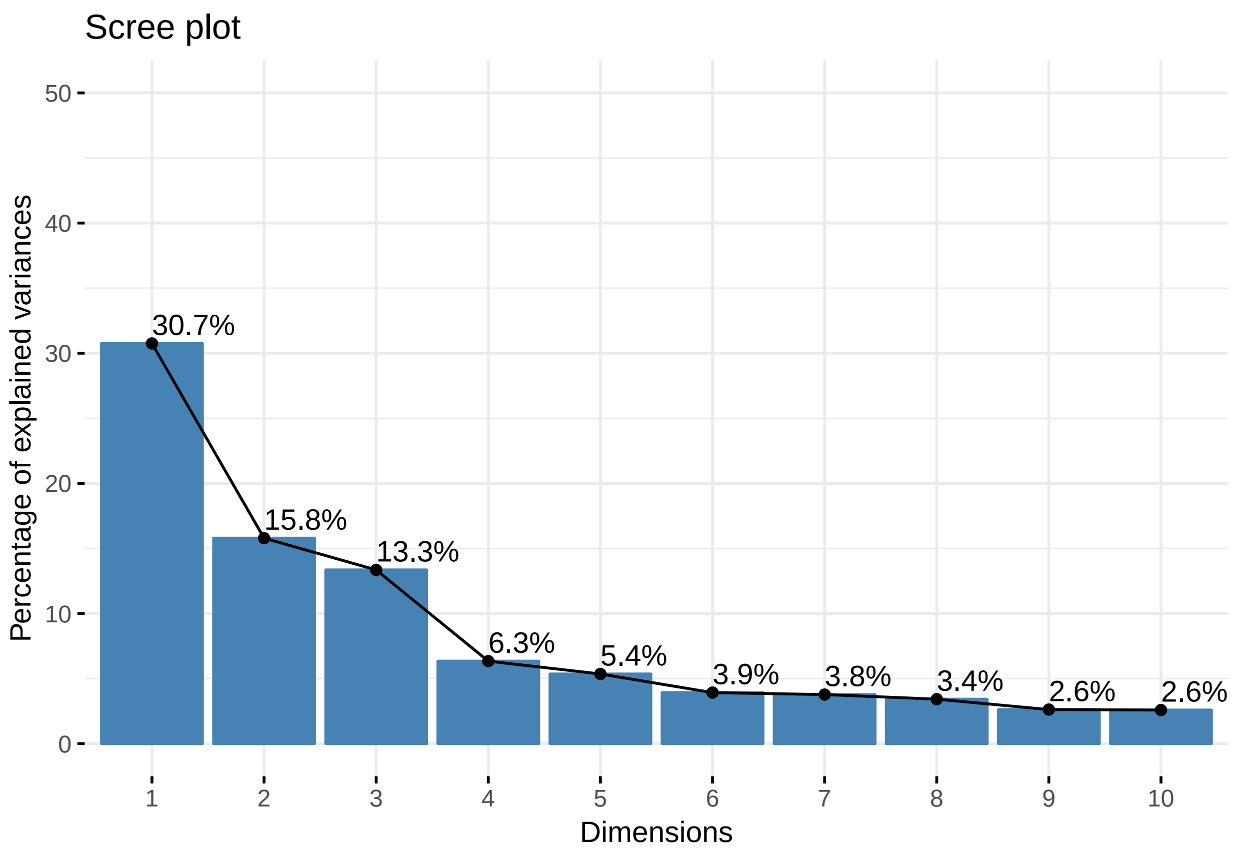
A first way to visualize distances between samples is to project the samples onto a 2-dimension plane such that they spread out optimally. This can be achieved using PCA analysis. Several PCA packages are available in R. Here, we simply use the function provided by DESeq2.

```
## short and easy version
DESeq2::plotPCA(rld, intgroup = c("condition"), ntop=1000)
```

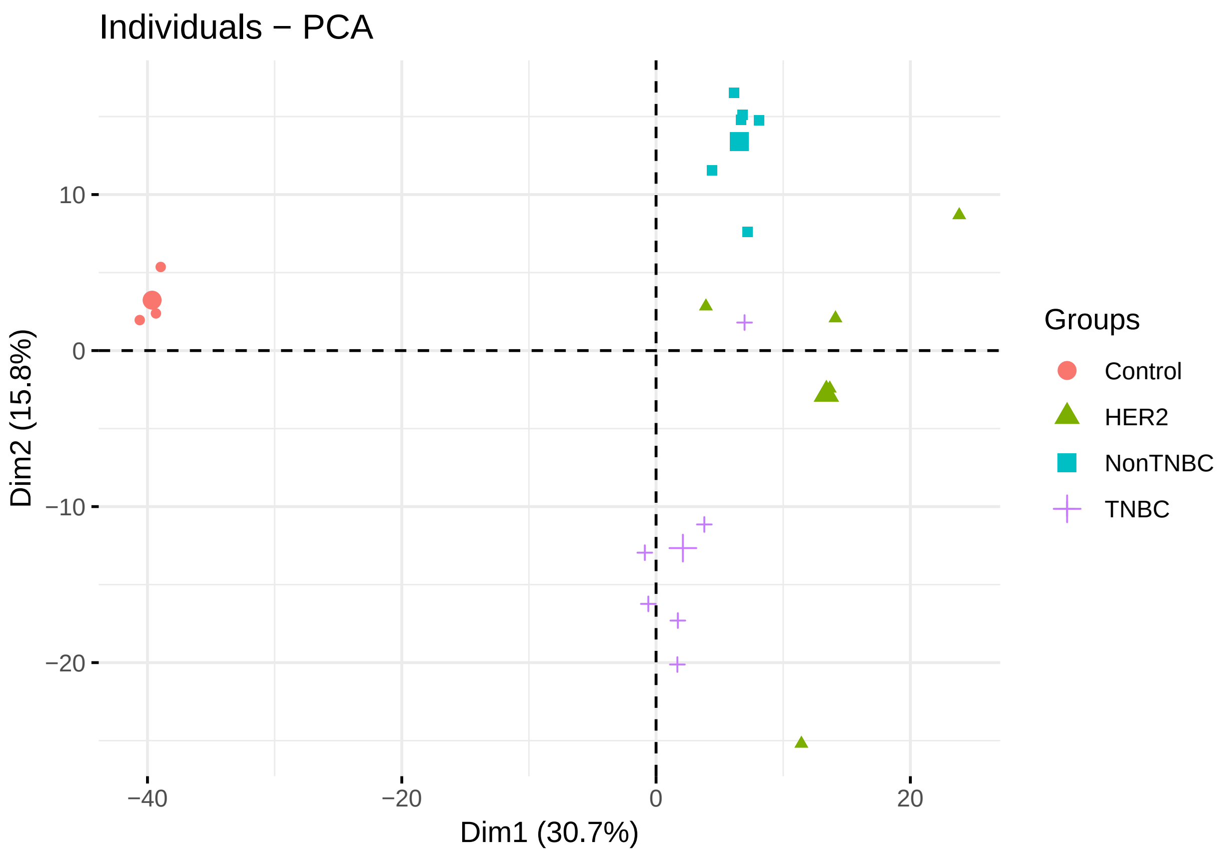


Other (nicer) visualization functions with `FactoMineR` . . .

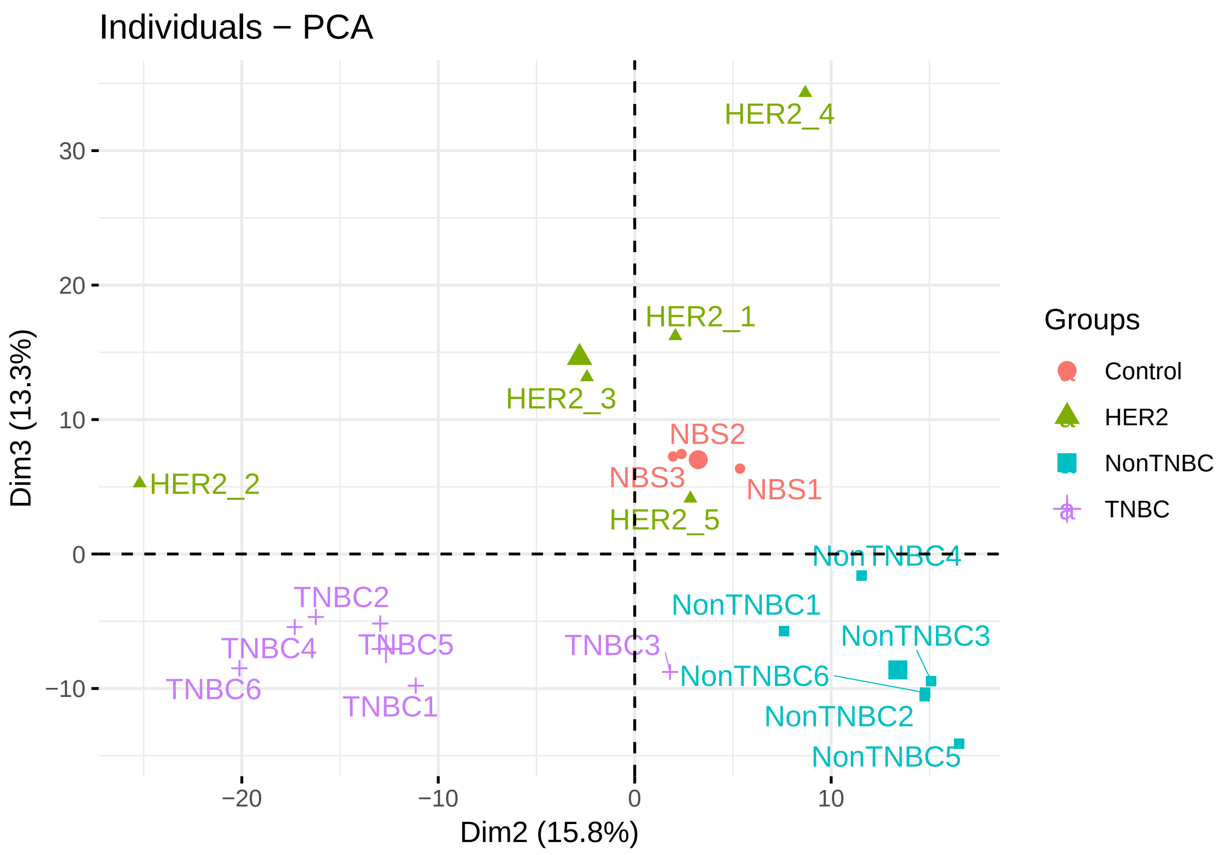
```
library(FactoMineR) ## for PCA
library(factoextra) ## for visualisation functions

## extract rlog matrix
d.rlog <- assay(rld)

## Select most variable genes
gvar <- apply(d.rlog, 1, var)
mostvargenes <- order(gvar, decreasing=TRUE)[1:1000]

## Run PCA
res_pca <- PCA(t(d.rlog[mostvargenes,]), ncp=3, graph=FALSE)

## Make beautiful plots
fviz_eig(res_pca, addlabels = TRUE, ylim = c(0, 50))
```



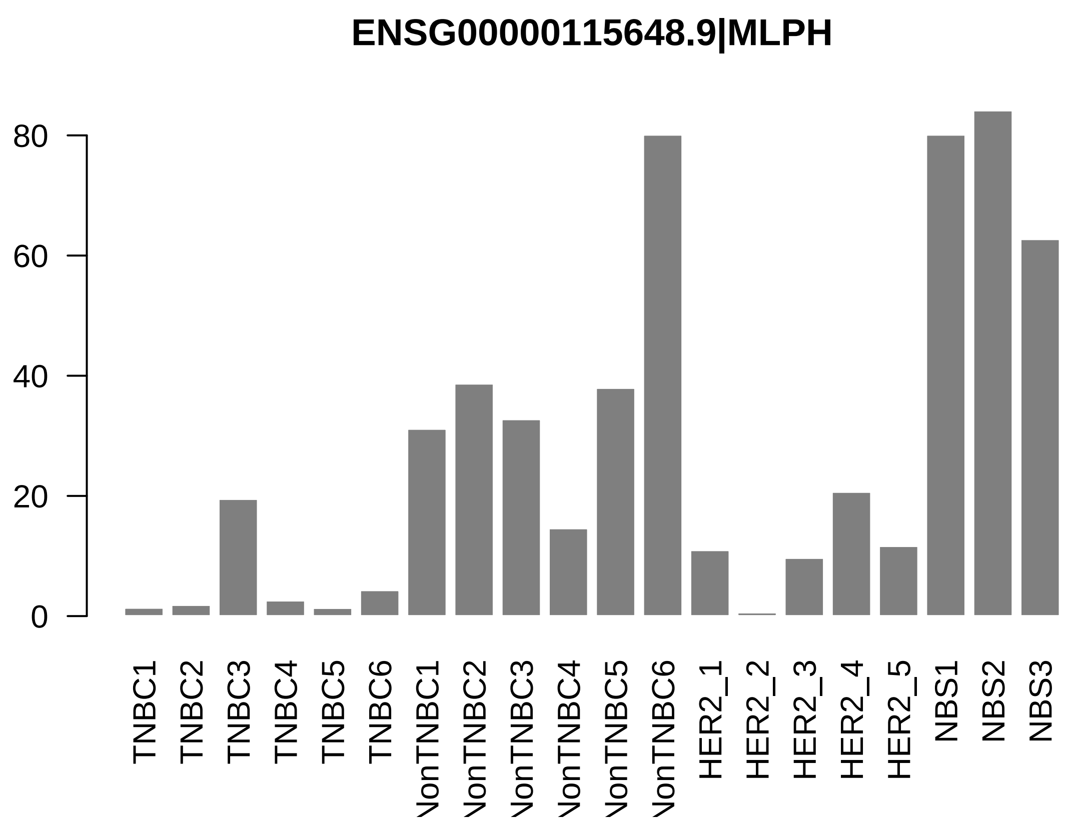
```
fviz_pca_ind(res_pca, label="none", habillage=splan$subtype)
```



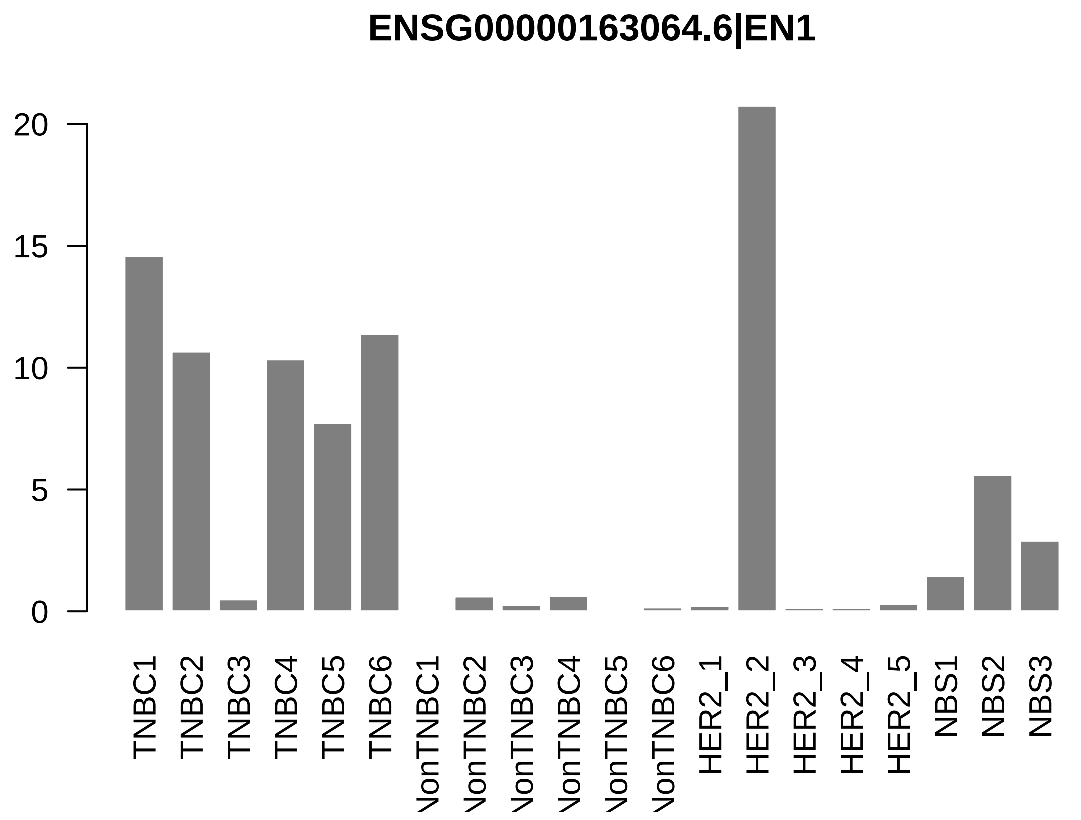
```
fviz_pca_ind(res_pca, axes=c(2,3), label="ind", habillage=splan$subtype, repel = TRUE)
```



```
## Look at the variable that contribute the most to the PC2
best.contrib <- names(sort(res_pca$var$contrib[,"Dim.2"], decreasing=TRUE))

## Let's check the best contributors
barplot(d.tpm[best.contrib[5],], col="gray50",
        border="white", las=2, main=best.contrib[5])
```



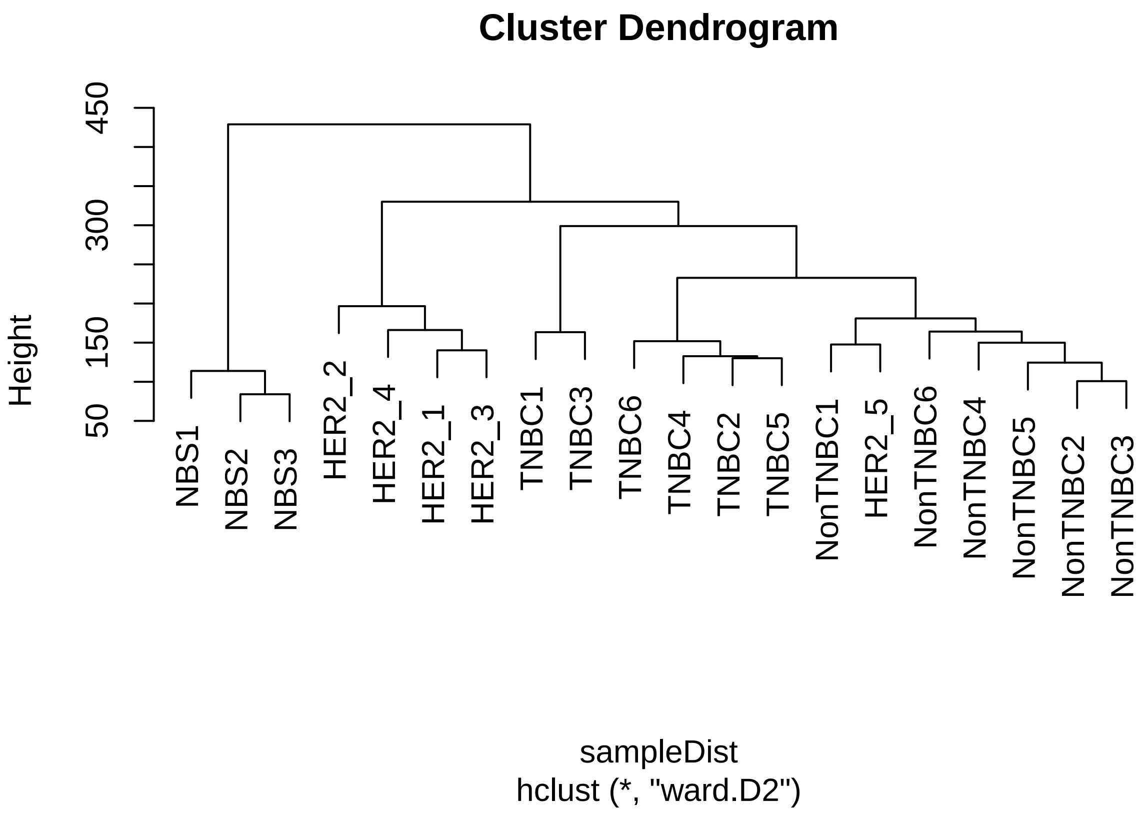
```
barplot(d.tpm[best.contrib[7],], col="gray50",
        border="white", las=2, main=best.contrib[7])
```



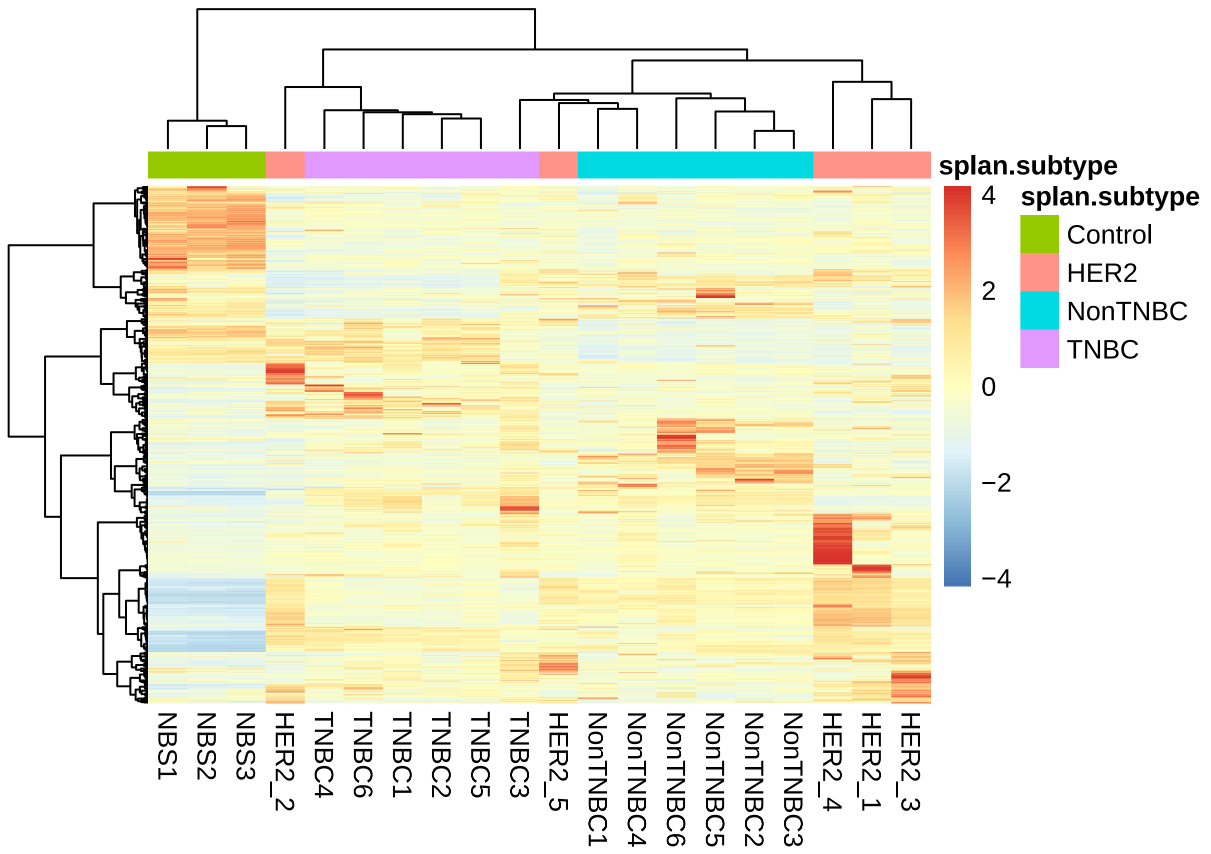
Hierarchical Clustering

In addition to PCA analysis, hierarchical clustering is another way to represent the similarity between samples using different metrics. Hierarchical clustering requires a `distance` between sample and a `method` to agglomerate clusters.

```
require(pheatmap)
require(RColorBrewer)

## matrix of rlog normalized data
d.rlog <- assay(rld)

## Clustering of samples on all genes
sampleDist <- dist(t(d.rlog))
hc <- hclust(sampleDist, method="ward.D2")
plot(hc)
```



```
## Clustering of samples on the most variable 100 genes
mostvargenes100 <- order(gvar, decreasing=TRUE)[1:100]
annot <- data.frame(splan$subtype, row.names=splan$sname)
pheatmap(d.rlog[mostvargenes,],
          clustering_distance_rows = "correlation",
          clustering_distance_cols = "euclidean",
          clustering_method = "ward.D2",
          annotation_col = annot,
          show_rownames = FALSE,
          scale = "row")
```



The first steps of the analysis presented above are dedicated to count data manipulation and exploration. The main goal is to validate the overlall quality of the dataset, and to see how similar are the different samples. If any outlier is detected, they will most likely increase the overall variability and thus decrease the statistical power later when testing for differential expression. It is therefore advised to discard these samples. Importantly, these analyses rely on appropriate normalization methods that stabilize the variance over the mean expression levels, and are therefore well designed for exploratory analysis.

Differential Analysis

Then, the next step of the analysis is usually to perform differential expression analysis between groups of samples. In this context, different normalization methods can be used, mainly based on Negative Binomial model. In RNA-seq count data, the variance is usually greater than the mean, requiring the estimation of an overdispersion parameter for each gene. Packages as DESeq2 or edgeR propose methods to normalize and detect differentially expressed genes based on these assumptions.
In addition, the limma package proposes an alternative approach based on a Gaussian model, through the `voom` transformation. This will compute additional precision weights to account for the change in the mean-variance relationship between small and large counts[10].

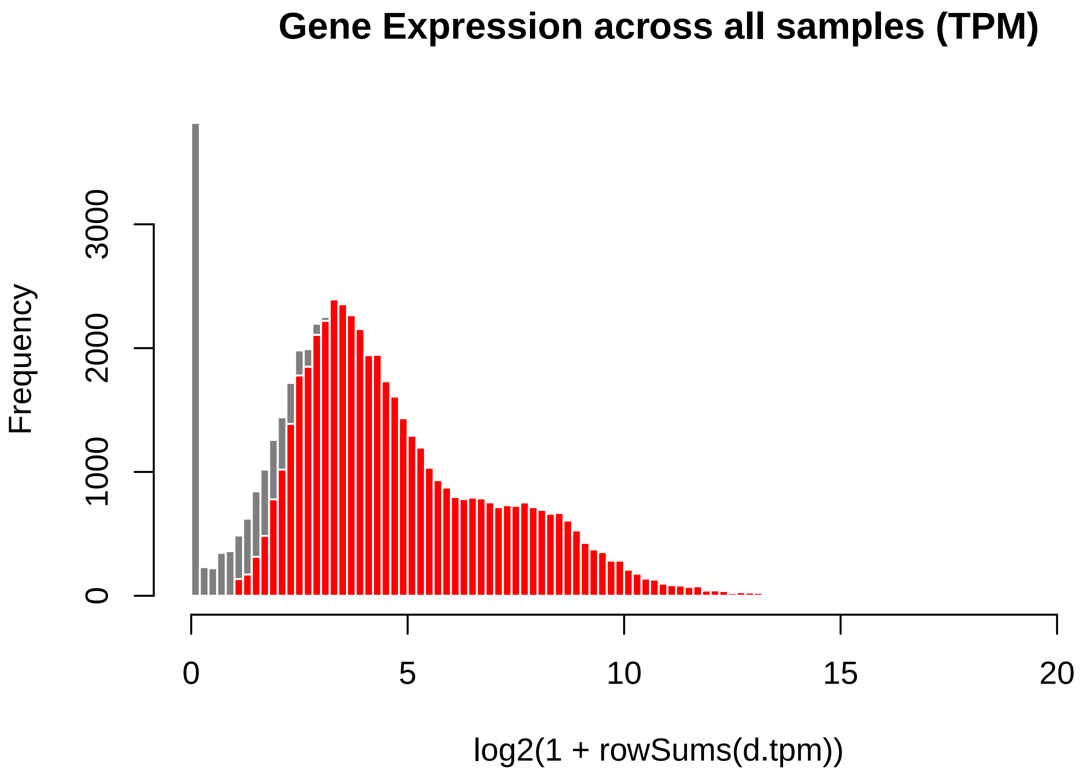
Genes filtering

Genes filtering is usually required before differential analysis. The idea is simply to restrict the analysis to expressed genes, and to remove any non-relevant information.

[10]Law C.W., Chen Y., Shi W., Smyth G.K. voom: precision weights unlowk linear model analysis tools for RNA-seq read counts. Genome Biol. 2014; 15(2): R29

Here, we defined the set of expressed genes as those with a TPM (transcripts per million) normalized counts >= 1 in at least one sample.

```
## Expressed genes
nbexpr <- apply(d.tpm, 1, function(x){length(which(x>=1))})
isexpr <- which(nbexpr>=1)
d.f <- d[isexpr,]

hist(log2(1+rowSums(d.tpm)), breaks=100, main="Gene Expression across all samples (TPM)", col="gray50",
hist(log2(1+rowSums(d.tpm[isexpr,])), breaks=100, add=TRUE, col="red", border="white")
```



Library size normalization

In the following example, we will use the trimmed mean of M-values method (TMM) from the `edgeR` package [11] that calculate a normalization factor that can be applied to the library size of each patient for normalization. Together with the `DESeq` method, TMM has been found to perform well in comparative studies.
It is important to keep in mind that such normalization is based on the assumption that most genes are invariant.
Of note, it is good to know that DESeq and edgeR methods do not relies on the same type of normalization factor. Each normalization method should therefore be used with its appropriate statistical modeling. Methods are therefore 'not' exchangeable.

```
## Scaling factor normalization based on the TMM method
require(edgeR)
require(limma)
y <- DGEList(counts=d.f)
y <- calcNormFactors(y, method="TMM")
```

[11] Robinson MD and Oshlack A. A scaling normalization method for differential analysis of RNA-seq data. Genome Biol. 2010; 11(3): R25

```
d.norm <- cpm(y, log=TRUE)
```

Differential analysis using limma

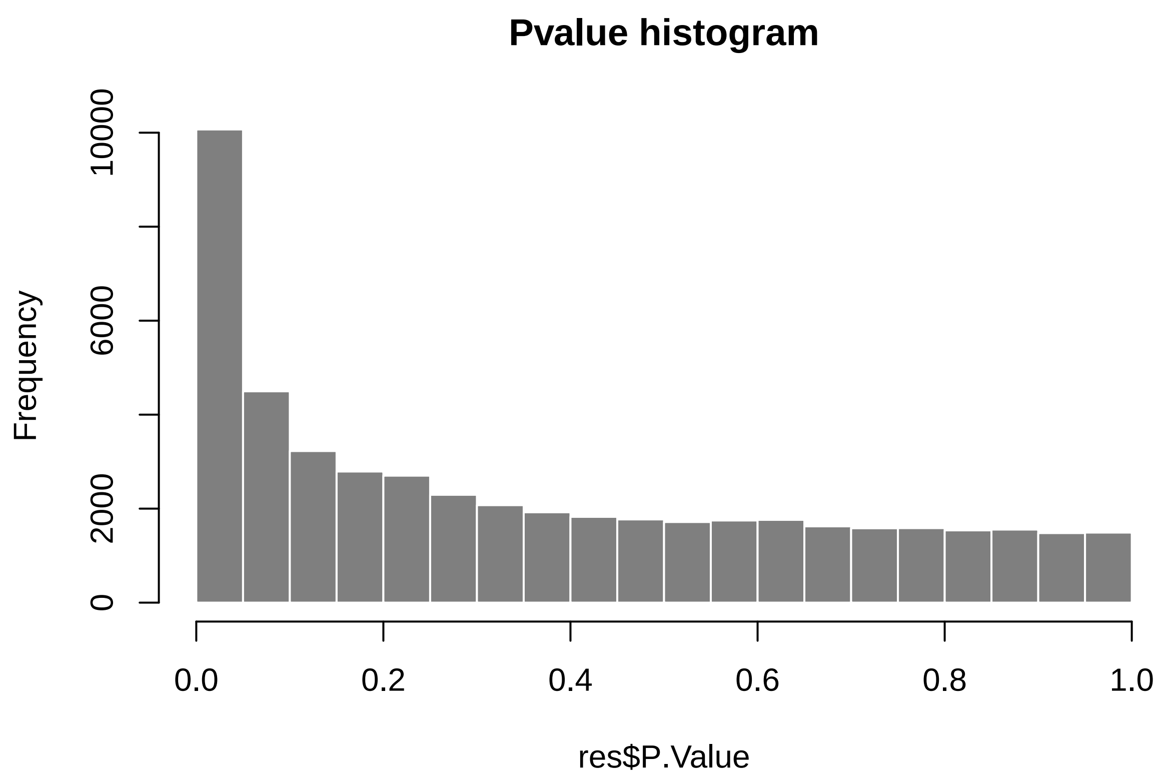
Once our data are normalized, we can now test for differential expression between groups.
Here again, several approaches like `DESeq`, `edgeR` or `limma` can be used.
Broadly speaking, the three methods should give close results, although from our experience `limma` is a bit more conservative.

When the library sizes are quite variable between samples, `limma` proposed an additional transformation (the `voom` approach) applied to the normalized and filtered `DGEList` object.

```
y <- DGEList(counts=d.f)
y <- calcNormFactors(y, method="TMM")

## Voom transformation of normalized data to apply the limma statistical framework
design <- model.matrix(~ 0 + subtype, data=splan)
v <- voom(y, design, plot=FALSE)
```

After this, the usual `limma` pipeline can be applied to comparer groups of samples (here the TNBC vs non TNBC samples).
Each gene is tested and has its own p-value. It is therefore important to correct for multiple testing in order to control for False Discovery Rate (FDR).

```
## Differential analysis based on 'limma'
fit <- lmFit(v, design)

## Compare TNBC vs NonTNBC
contrast <-makeContrasts(subtypeTNBC - subtypeNonTNBC, levels=design)

## Run test
fit2 <- contrasts.fit(fit, contrast)
fit2 <- eBayes(fit2)

## Extract the results
res <- topTable(fit2, number=1e6, adjust.method="BH")

## Pvalue distribution
hist(res$P.Value, main="Pvalue histogram", col="grey50", border="white")
```



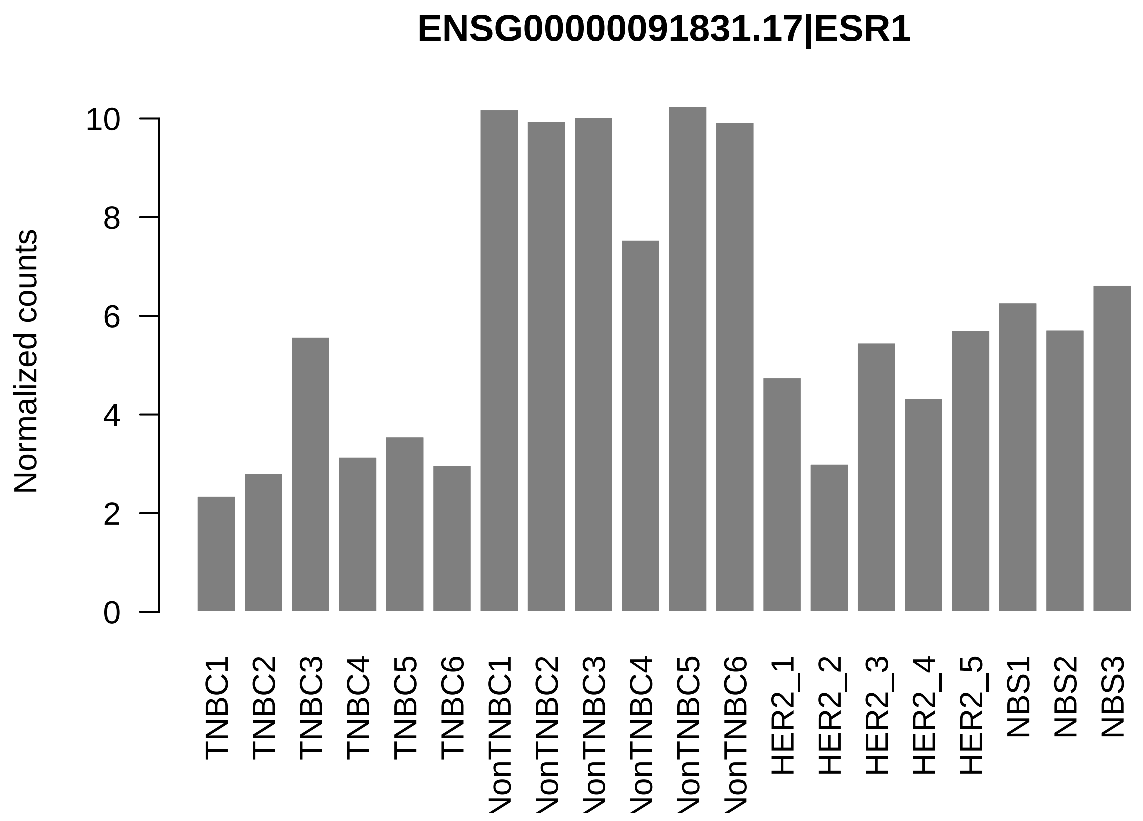
```
## Extract list of DEG
idx.sign <- which(res$adj.P.Val < 0.05 &
                  abs(res$logFC) > 1)
deg <- rownames(res[idx.sign,])
```

Plotting results

After differential analysis, it is highly recommanded to make simple plots to double check the results, such as:

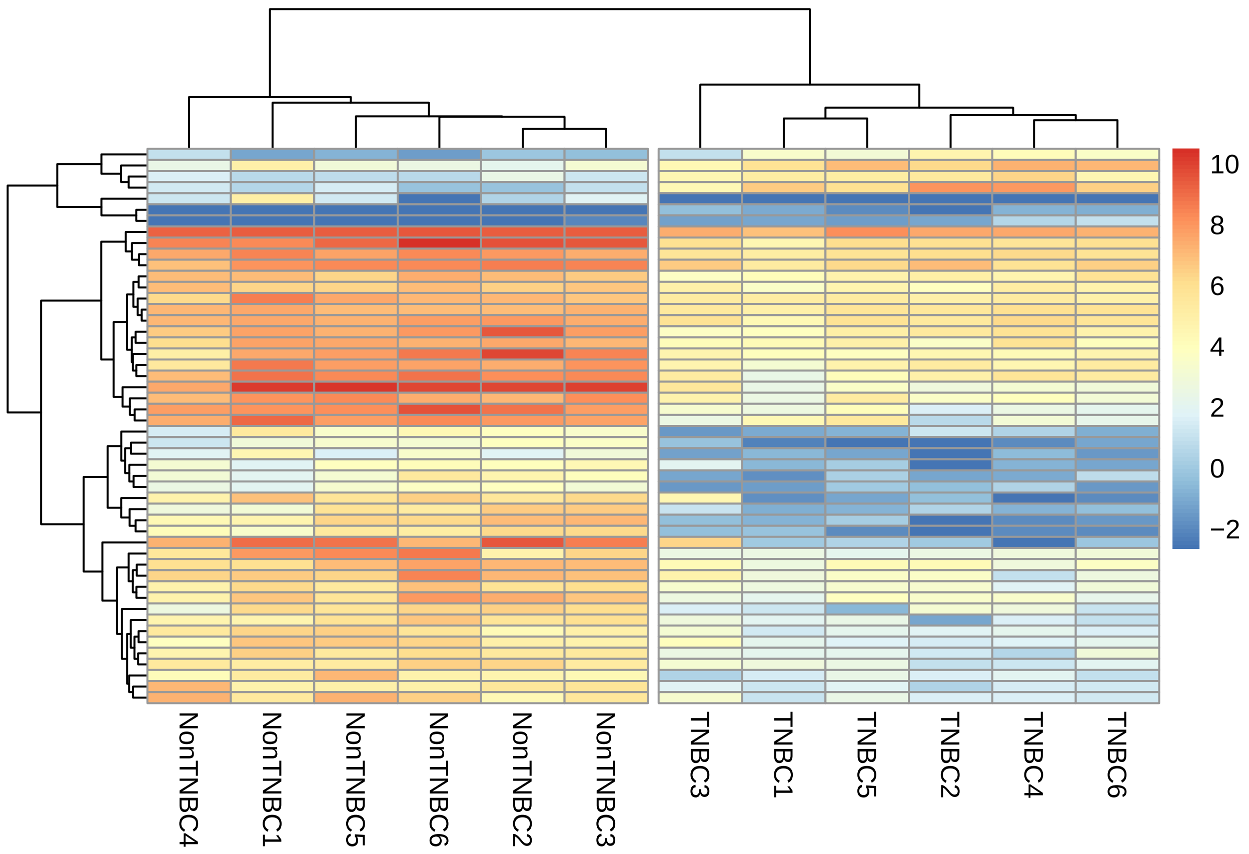
- Gene based plot, in order to check that differentially expressed genes match the expected pattern

```
mygene <- rownames(res)[1]
barplot(d.norm[mygene,], las=2, main=mygene, ylab="Normalized counts",
        col="grey50", border="white")
```



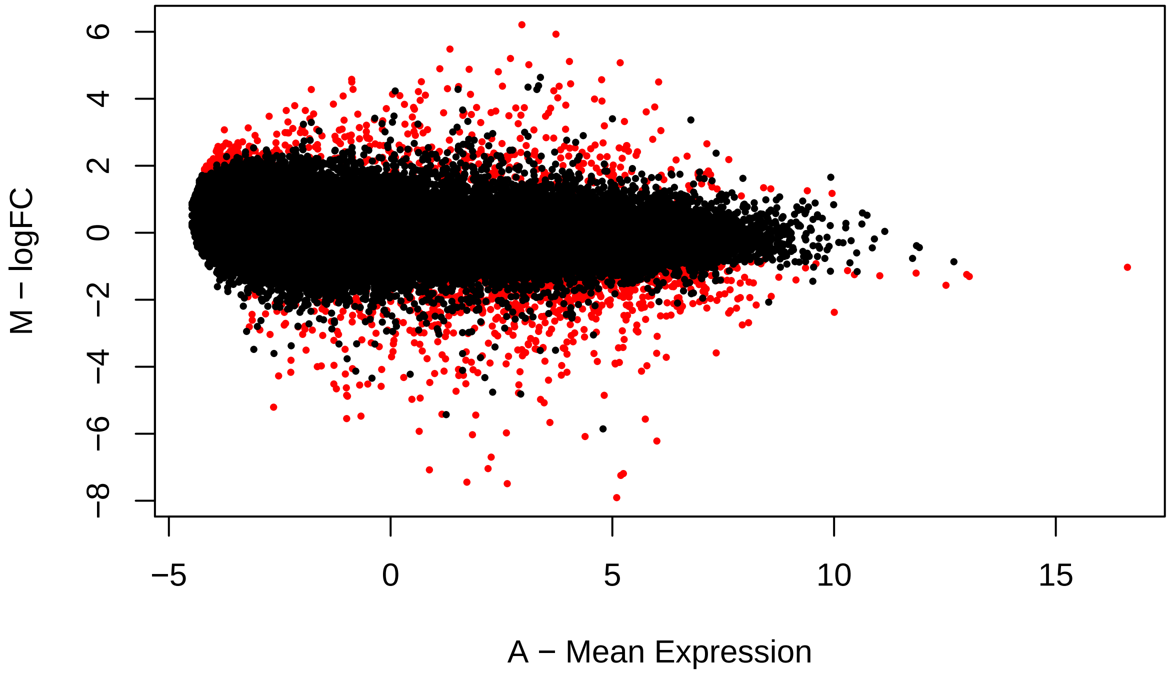
- Heatmap of 50 most differentially expressed genes

```
idx.sub <- which(splan$subtype=="TNBC" | splan$subtype=="NonTNBC")
data.sub <- d.norm[deg[1:50],idx.sub]
pheatmap(data.sub,
        cutree_cols = 2,
        show_rownames=FALSE)
```



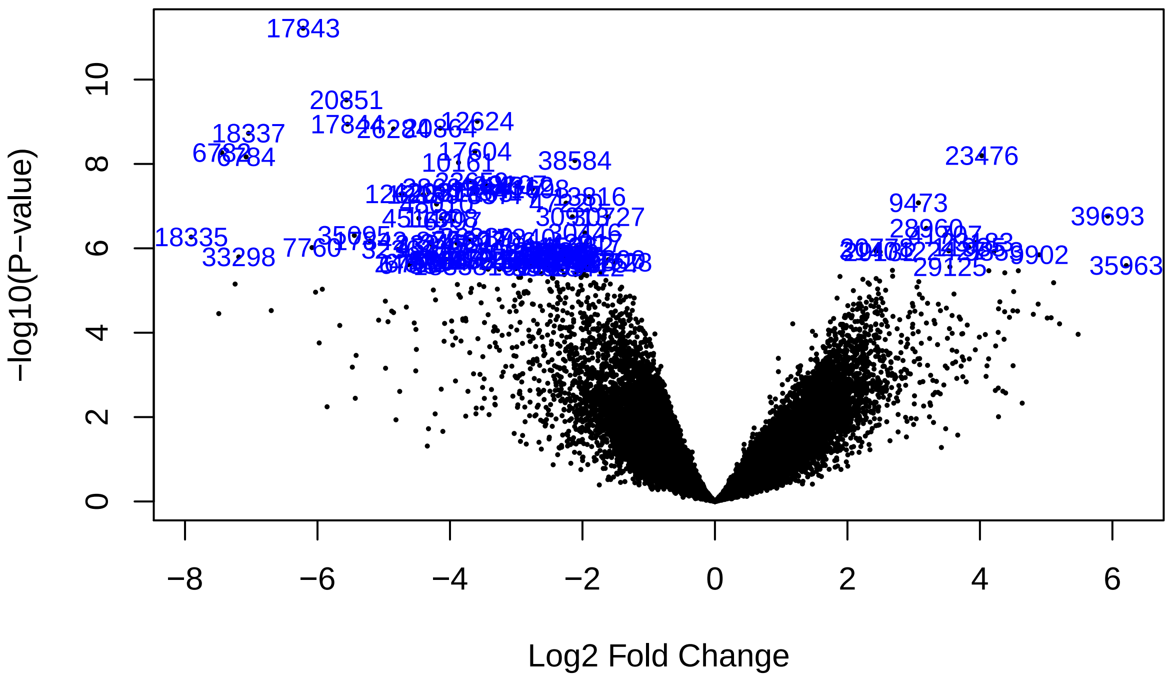
- MA plot (mean / average) where 'M' stands for 'fold-change', and 'A' for 'mean expression'.

```
plot(res$AveExpr, res$logFC, xlab="A - Mean Expression", ylab="M - logFC",
     col=ifelse(res$adj.P.Val<0.05, "red", "black"), pch=16, cex=.5)
```



- A volcano plot to represent both the gene p-value and the fold-change information

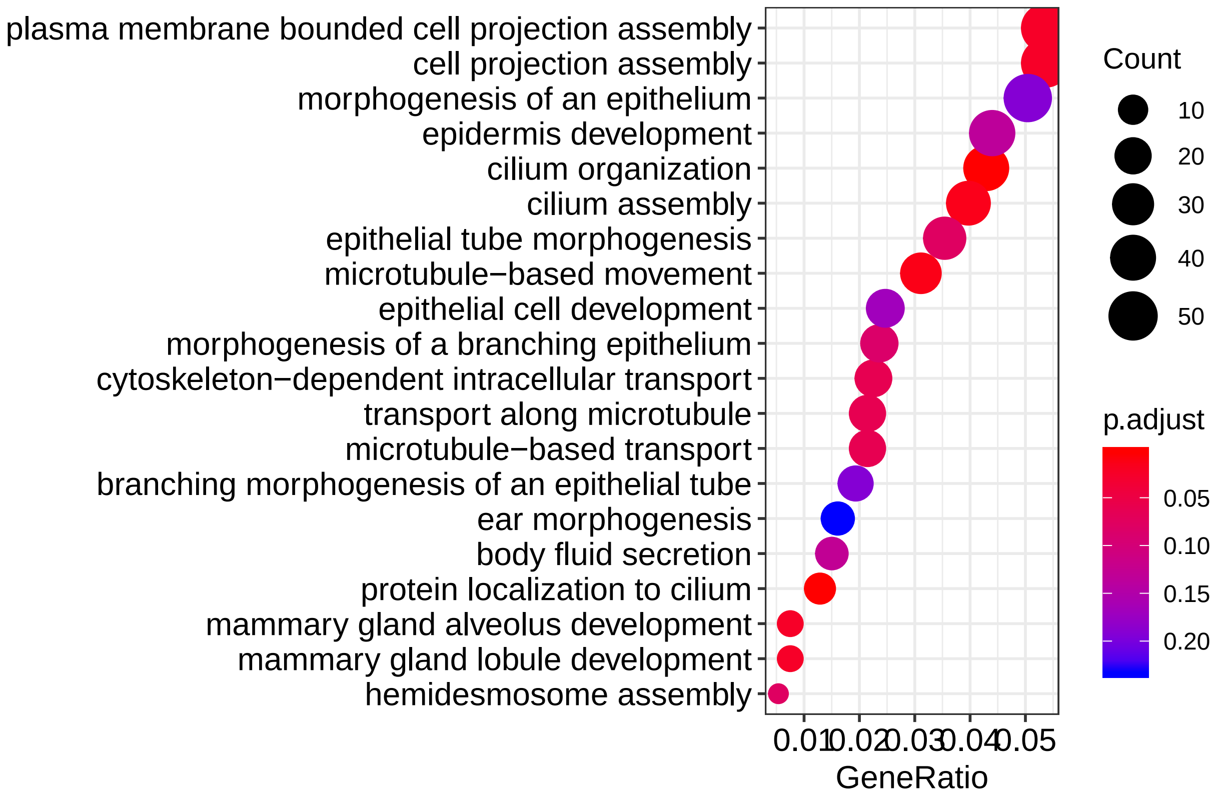
```
volcanoplot(fit2, highlight=100)
```



Functional analysis

Here is a short example of Gene Ontology analysis with R, and the `clusterProfiler` package. Many other packages are available for functional analysis. You can also use online tools such as PANTHER or DAVID.

```
library(org.Hs.eg.db)
library(clusterProfiler)

## convert to Entrez Id
symbol.sign <- sapply(strsplit(deg, "\\|"), "[", 2)
entrez.sign <- bitr(symbol.sign, fromType = "SYMBOL",
                    toType = c("ENTREZID"), OrgDb = org.Hs.eg.db)
symbol.all <- sapply(strsplit(rownames(res), "\\|"), "[", 2)
universe <- bitr(symbol.all, fromType = "SYMBOL",
                 toType = c("ENTREZID"), OrgDb = org.Hs.eg.db)

## Enrich GO
ego.BP <- enrichGO(gene=entrez.sign$ENTREZID, universe=universe$ENTREZID, OrgDb=org.Hs.eg.db, ont="BP",
dotplot(dropGO(ego.BP, level=c(1:3)), showCategory=20)
```



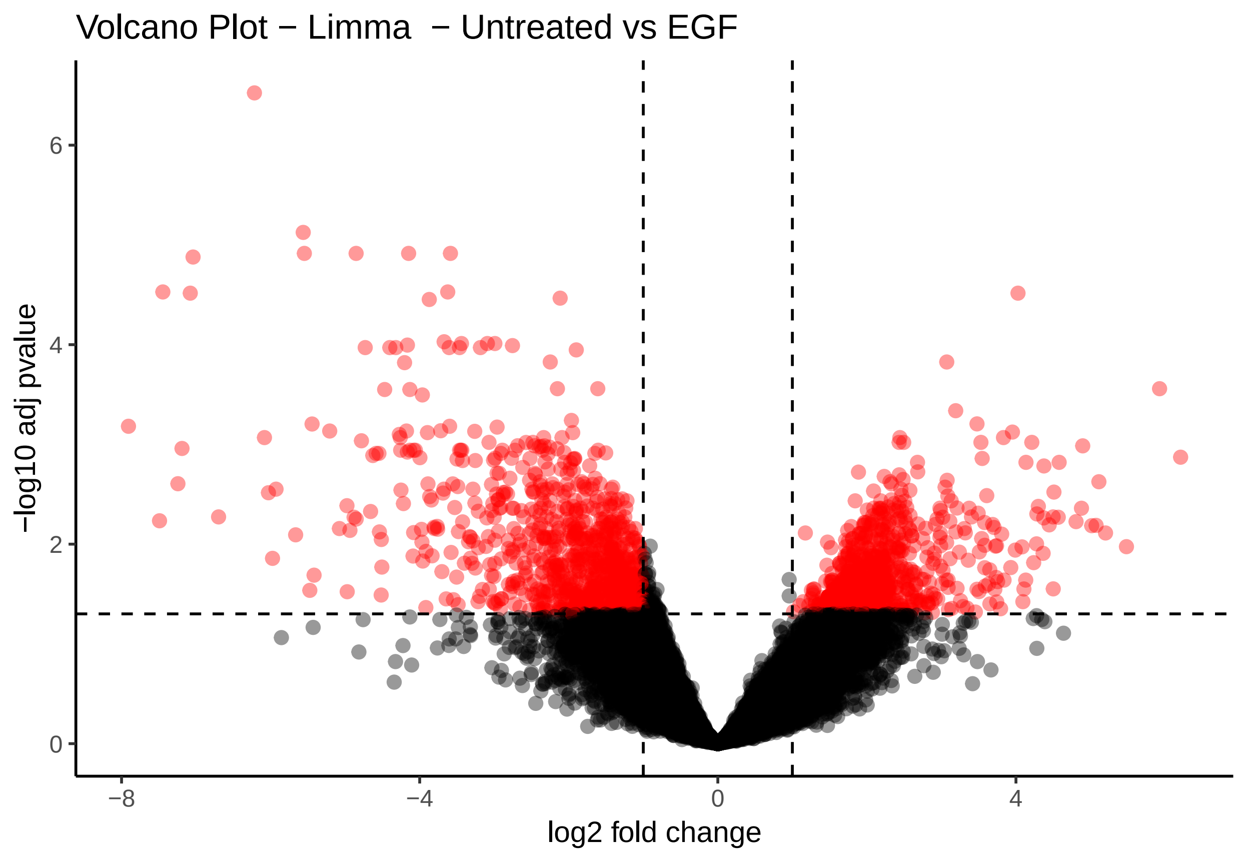
Advanced exercices with R

Making beautiful plots with ggplot

The previous VolcanoPlot is not very nice, and writting you own function can be useful to make beautiful plots ! As an exercice, you can write your own function based on the ggplot2 package to make a much better VolcanoPlot.

Solution

```
LimmaVolcano <- function(res, main="", fct=1.5, pt=0.05){
    stopifnot(require(ggplot2))
    res$sign <- 0
    res$sign[which(res$adj.P.Val < pt & abs(res$logFC) > fct)] <- 1

    p <- ggplot(data=res, aes(x=logFC, y=-log10(adj.P.Val), colour=as.factor(sign))) +
      theme_classic() + geom_point(alpha=0.4, size=2) +
      scale_color_manual(name="", values=c("1"="red", "0"="black")) +
      ggtitle(paste0("Volcano Plot - Limma ", main)) + theme(legend.position = "none") +
      xlab("log2 fold change") + ylab("-log10 adj pvalue") +
      geom_vline(xintercept=c(-fct, fct), linetype=2) +
      geom_hline(yintercept=-log10(pt), linetype=2)
    p
}
LimmaVolcano(res, fct=1, pt=0.05, main=" - Untreated vs EGF")
```



Dealing with gene annotation

There are many ways to deal with annotations in R. Several packages include databases that can be requested to convert a gene ID to another (see BioMart). You can look at this online tutorial for examples and details.

Here, our dataset is based on ENSEMBL annotation. Our goal is thus to convert the `ENSEMBL` IDs into `SYMBOL` which are already available from the annotation file we used during the data processing (gtf).

1. Write a function able to load a gtf file with both ENSEMBL and SYMBOL annotations (available in ./data/gencode.v19.annotation.gtf.gz), and to convert the gene annotation.

Solution

```r
## ensemble2symbol
## x: matrix with ENSEMBL Ids as rownames
## gtf.in: path to gtf file
## return: x with ENSEMBL|SYMBOL annotation
ensembl2symbol <- function(x, gtf.in){
  stopifnot(require(rtracklayer))
  message("Loading gtf file ...")
  dgtf <- rtracklayer::import(gtf.in)

  message("Subset only genes ...")
  my_genes <- dgtf[dgtf$type=="gene"]
  mcols(my_genes) <- mcols(my_genes)[c("gene_id","gene_type","gene_name")]

  message("Convert ENSEMBL to SYMBOL")
  m <- match(rownames(x), my_genes$gene_id)
  m.symb <- m[which(!is.na(m))]

  message("Loosing ", length(which(is.na(m))), " genes ...")
  x <- x[m.symb,]
  rownames(x) <- my_genes$gene_name[m.symb]
  return(x)
}

mygtf <- file.path(proj, "gencode.v19.annotation.gtf.gz")
d.ensembl <- read.csv(file.path(proj, "tablecounts_raw_ensembl.csv"),
                      row.names = 1)
d.ensembl <- as.matrix(d.ensembl)
d.annot <- ensembl2symbol(d.ensembl, mygtf)
```

Session Info

```
sessionInfo()
```

```
## R version 3.6.3 (2020-02-29)
## Platform: x86_64-pc-linux-gnu (64-bit)
## Running under: Ubuntu 18.04.5 LTS
##
## Matrix products: default
## BLAS:   /usr/lib/x86_64-linux-gnu/blas/libblas.so.3.7.1
## LAPACK: /usr/lib/x86_64-linux-gnu/lapack/liblapack.so.3.7.1
##
## locale:
##  [1] LC_CTYPE=fr_FR.UTF-8       LC_NUMERIC=C
##  [3] LC_TIME=fr_FR.UTF-8        LC_COLLATE=fr_FR.UTF-8
##  [5] LC_MONETARY=fr_FR.UTF-8    LC_MESSAGES=fr_FR.UTF-8
##  [7] LC_PAPER=fr_FR.UTF-8       LC_NAME=C
##  [9] LC_ADDRESS=C               LC_TELEPHONE=C
## [11] LC_MEASUREMENT=fr_FR.UTF-8 LC_IDENTIFICATION=C
##
## attached base packages:
## [1] parallel  stats4    stats     graphics  grDevices utils     datasets
## [8] methods   base
##
## other attached packages:
##  [1] rtracklayer_1.43.4          clusterProfiler_3.12.0
##  [3] org.Hs.eg.db_3.8.2          AnnotationDbi_1.45.1
##  [5] edgeR_3.25.9                limma_3.39.19
##  [7] RColorBrewer_1.1-2          pheatmap_1.0.12
##  [9] factoextra_1.0.6            ggplot2_3.1.1
## [11] FactoMineR_1.42             DESeq2_1.23.10
## [13] SummarizedExperiment_1.13.0 DelayedArray_0.8.0
## [15] BiocParallel_1.17.19        matrixStats_0.58.0
## [17] Biobase_2.43.1              GenomicRanges_1.35.1
## [19] GenomeInfoDb_1.19.3         IRanges_2.17.5
## [21] S4Vectors_0.21.24           BiocGenerics_0.29.2
##
## loaded via a namespace (and not attached):
```

```
##   [1] fgsea_1.10.0             colorspace_1.4-1
##   [3] ggridges_0.5.1           qvalue_2.16.0
##   [5] htmlTable_1.13.1         XVector_0.23.2
##   [7] base64enc_0.1-3          rstudioapi_0.10
##   [9] ggpubr_0.2               farver_1.1.0
##  [11] urltools_1.7.3           ggrepel_0.8.0
##  [13] bit64_0.9-7              xml2_1.2.0
##  [15] splines_3.6.3            leaps_3.0
##  [17] GOSemSim_2.10.0          geneplotter_1.60.0
##  [19] knitr_1.22               polyclip_1.10-0
##  [21] jsonlite_1.6             Formula_1.2-3
##  [23] Rsamtools_1.99.6         annotate_1.60.1
##  [25] cluster_2.1.0            GO.db_3.8.2
##  [27] ggforce_0.2.2            httr_1.4.0
##  [29] compiler_3.6.3           rvcheck_0.1.3
##  [31] backports_1.1.4          assertthat_0.2.1
##  [33] Matrix_1.3-2             lazyeval_0.2.2
##  [35] tweenr_1.0.1             prettyunits_1.0.2
##  [37] acepack_1.4.1            htmltools_0.3.6
##  [39] tools_3.6.3              igraph_1.2.4.1
##  [41] gtable_0.3.0             glue_1.3.1
##  [43] GenomeInfoDbData_1.2.1   reshape2_1.4.3
##  [45] DO.db_2.9                dplyr_0.8.0.1
##  [47] fastmatch_1.1-0          Rcpp_1.0.6
##  [49] enrichplot_1.4.0         Biostrings_2.51.5
##  [51] ggraph_1.0.2             xfun_0.6
##  [53] stringr_1.4.0            XML_3.98-1.19
##  [55] DOSE_3.10.2              europepmc_0.3
##  [57] zlibbioc_1.29.0          MASS_7.3-53.1
##  [59] scales_1.0.0             hms_0.4.2
##  [61] yaml_2.2.0               memoise_1.1.0
##  [63] gridExtra_2.3            UpSetR_1.4.0
##  [65] triebeard_0.3.0          rpart_4.1-15
##  [67] latticeExtra_0.6-28      stringi_1.4.3
##  [69] RSQLite_2.1.1            genefilter_1.65.0
##  [71] checkmate_1.9.1          rlang_0.4.10
##  [73] pkgconfig_2.0.2          bitops_1.0-6
##  [75] evaluate_0.13            lattice_0.20-41
##  [77] purrr_0.3.2              GenomicAlignments_1.19.1
##  [79] htmlwidgets_1.5.3        labeling_0.3
##  [81] cowplot_0.9.4            bit_4.0.4
##  [83] tidyselect_0.2.5         plyr_1.8.4
##  [85] magrittr_1.5             R6_2.4.0
##  [87] Hmisc_4.2-0              DBI_1.0.0
##  [89] pillar_1.3.1             foreign_0.8-76
##  [91] withr_2.1.2              survival_3.2-7
##  [93] scatterplot3d_0.3-41     RCurl_1.95-4.12
##  [95] nnet_7.3-15              tibble_2.1.1
##  [97] crayon_1.3.4             rmarkdown_1.12
##  [99] progress_1.2.0           viridis_0.5.1
## [101] locfit_1.5-9.1           grid_3.6.3
## [103] data.table_1.12.2        blob_1.1.1
## [105] digest_0.6.18            flashClust_1.01-2
## [107] xtable_1.8-4             tidyr_0.8.3
## [109] gridGraphics_0.4-1       munsell_0.5.0
## [111] ggplotify_0.0.3          viridisLite_0.3.0
```

Acknowledgments

This support has been prepared with the support of the Cancéropole Ile de France.

References